Gradle User Manual

Version 7.6.6-20260912032715+0000

Version 7.6.6-20260912032715+0000

Table of Contents

About Gradle . . . 4
- What is Gradle? . . . 4

Getting Started . . . 7
- Getting Started . . . 7
- Installing Gradle . . . 9
- Troubleshooting builds . . . 12
- Compatibility Matrix . . . 19

Upgrading and Migrating . . . 22
- Upgrading your build from Gradle 6.x to 7.0 . . . 22
- Upgrading your build from Gradle 5.x to 6.0 . . . 41
- Upgrading your build from Gradle 4.x to 5.0 . . . 60
- Migrating Builds From Apache Maven . . . 86
- Migrating Builds From Apache Ant . . . 106

Running Gradle Builds . . . 125
- Build Environment . . . 125
- The Gradle Daemon . . . 144
- Initialization Scripts . . . 149
- Executing Multi-Project Builds . . . 160
- Build Cache . . . 165

Authoring Gradle Builds . . . 183
- Build Script Basics . . . 183
- Authoring Tasks . . . 201
- Writing Build Scripts . . . 281
- Working With Files . . . 296
- Using Gradle Plugins . . . 360
- Build Lifecycle . . . 382
- Logging . . . 394
- Structuring and Building a Software Component with Gradle . . . 403
- Sharing Build Logic between Subprojects . . . 407
- Fine Tuning the Project Layout . . . 408
- Declaring Dependencies between Subprojects . . . 410
- Configuration time and execution time . . . 420
- Organizing Gradle Projects . . . 422
- Best practices for authoring maintainable builds . . . 429
- Lazy Configuration . . . 439
- Testing Build Logic with TestKit . . . 467
- Using Ant from Gradle . . . 481

Dependency Management . . . 500

Learning the Basics . . . 500
Declaring Versions . . . 611
Controlling Transitive Dependencies . . . 641
Producing and Consuming Variants of Libraries . . . 741
Working in a Multi-repo Environment . . . 818
Publishing Libraries . . . 827
Java & Other JVM Projects . . . 856
Building Java & JVM projects . . . 856
Testing in Java & JVM projects . . . 882
Managing Dependencies of JVM Projects . . . 920
C++ & Other Native Projects . . . 925
Building C++ projects . . . 925
Testing in C++ projects . . . 935
Building Swift projects . . . 936
Testing in Swift projects . . . 945
Native Projects using the Software Model . . . 955
Building native software . . . 955
Implementing model rules in a plugin . . . 994
Extending Gradle . . . 995
Developing Custom Gradle Task Types . . . 995
Developing Custom Gradle Plugins . . . 1026
Developing Custom Gradle Types . . . 1049
Gradle Plugin Development Plugin . . . 1059
Reference . . . 1063
A Groovy Build Script Primer . . . 1063
Gradle Kotlin DSL Primer . . . 1068
Gradle Plugin Reference . . . 1103
Command-Line Interface . . . 1106
Gradle & Third-party Tools . . . 1125
The Gradle Wrapper . . . 1129
The Directories and Files Gradle Uses . . . 1138
Plugins . . . 1141
The ANTLR Plugin . . . 1141
The Application Plugin . . . 1144
The Base Plugin . . . 1152
Build Init Plugin . . . 1155
The Checkstyle Plugin . . . 1163
The CodeNarc Plugin . . . 1166
The Distribution Plugin . . . 1167
The Ear Plugin . . . 1172
The Eclipse Plugins . . . 1178

The Groovy Plugin . . . 1188
The IDEA Plugin . . . 1199
Ivy Publish Plugin . . . 1208
The JaCoCo Plugin . . . 1222
The Java Plugin . . . 1234
The Java Library Plugin . . . 1253
The Java Library Distribution Plugin . . . 1268
The Java Platform Plugin . . . 1270
Maven Publish Plugin . . . 1277
The PMD Plugin . . . 1299
The Scala Plugin . . . 1302
The Signing Plugin . . . 1313
The War Plugin . . . 1323
License Information . . . 1329
License Information . . . 1330
Gradle Documentation . . . 1330
Gradle Build Scan Plugin . . . 1330

About Gradle

What is Gradle?

Gradle is an open-source build automation tool flexible enough to build almost any type of software. Gradle makes few assumptions about what you're trying to build or how to build it. This makes Gradle particularly flexible.

Design

Gradle bases its design on the following fundamentals:

High performance

Gradle avoids unnecessary work by only running tasks that need to do work because inputs or outputs have changed. Gradle uses various caches to reuse outputs from previous builds. With a shared build cache, you can even reuse outputs from other machines.

JVM foundation

Gradle runs on the JVM. This is a bonus for users familiar with Java, since build logic can use the standard Java APIs. It also makes it easy to run Gradle on different platforms.

Conventions

Gradle makes common types of projects easy to build through conventions. Plugins set sensible defaults to keep build scripts minimal. But these conventions don't limit you: you can configure settings, add your own tasks, and make many other customizations in your builds.

Extensibility

Most builds have special requirements that require custom build logic. You can readily extend Gradle to provide your own build logic with custom tasks and plugins. See Android builds for an example: they add many new build concepts such as flavors and build types.

IDE support

Several major IDEs provide interaction with Gradle builds, including Android Studio, IntelliJ IDEA, Eclipse, VSCode, and NetBeans. Gradle can also generate the solution files required to load a project into Visual Studio.

Insight

Build Scan™ provides extensive information about a build that you can use to identify issues. You can use Build Scans to identify problems with a build's performance and even share them for debugging help.

Terminology

It's helpful to know the following terminology before you dive into the details of Gradle.

Projects

Projects are the things that Gradle builds. Projects contain a build script, which is a file located in the project's root directory usually named `build.gradle` or `build.gradle.kts`. Builds scripts define tasks, dependencies, plugins, and other configuration for that project. A single build can contain one or more projects and each project can contain their own subprojects.

Tasks

Tasks contain the logic for executing some work—compiling code, running tests or deploying software. In most use cases, you'll use existing tasks. Gradle provides tasks that implement many common build system needs, like the built-in Java `Test` task that can run tests. Plugins provide even more types of tasks.

Tasks themselves consist of:

- Actions: pieces of work that do something, like copy files or compile source
- Inputs: values, files and directories that the actions use or operate on
- Outputs: files and directories that the actions modify or generate

Plugins

Plugins allow you to introduce new concepts into a build beyond tasks, files and dependency configurations. For example, most language plugins add the concept of source sets to a build.

Plugins provide a means of reusing logic and configuration across multiple projects. With plugins, you can write a task once and use it in multiple builds. Or you can store common configuration, like logging, dependencies, and version management, in one place. This reduces duplication in build scripts. Appropriately modeling build processes with plugins can greatly improve ease of use and efficiency.

Build Phases

Gradle evaluates and executes build scripts in three **build phases** of the Build Lifecycle:

Initialization
: Sets up the environment for the build and determine which projects will take part in it.

Configuration
: Constructs and configures the task graph for the build. Determines which tasks need to run and in which order, based on the task the user wants to run.

Execution
: Runs the tasks selected at the end of the configuration phase.

Builds

A build is an execution of a collection of tasks in a Gradle project. You run a build via the command line interface (CLI) or an IDE by specifying task selectors. Gradle configures the build and selects

the tasks to run. Gradle runs the smallest complete set of tasks based on the requested tasks and their dependencies.

Getting Started

Getting Started

Everyone has to start somewhere and if you're new to Gradle, this is where to begin.

Before you start

In order to use Gradle effectively, you need to know what it is and understand some of its fundamental concepts. So before you start using Gradle in earnest, we highly recommend you read What is Gradle?.

Installation

If all you want to do is run an existing Gradle build, then you don't need to install Gradle if the build has a Gradle Wrapper, identifiable via the *gradlew* and/or *gradlew.bat* files in the root of the build. You just need to make sure your system satisfies Gradle's prerequisites.

Android Studio comes with a working installation of Gradle, so you don't need to install Gradle separately in that case.

In order to create a new build or add a Wrapper to an existing build, you will need to install Gradle according to these instructions. Note that there may be other ways to install Gradle in addition to those described on that page, since it's nearly impossible to keep track of all the package managers out there.

Try Gradle

Actively using Gradle is a great way to learn about it, so once you've installed Gradle, try one of the introductory hands-on tutorials:

- Building Android apps
- Building Java applications
- Building Java libraries
- Building Groovy applications
- Building Groovy libraries
- Building Scala applications
- Building Scala libraries
- Building Kotlin JVM applications
- Building Kotlin JVM libraries
- Building C++ applications
- Building C++ libraries
- Building Swift applications

- Building Swift libraries
- Creating build scans

There are more samples available on the samples pages.

Command line vs IDEs

Some folks are hard-core command-line users, while others prefer to never leave the comfort of their IDE. Many people happily use both and Gradle endeavors not to discriminate. Gradle is supported by several major IDEs and everything that can be done from the command line is available to IDEs via the Tooling API.

Android Studio and IntelliJ IDEA users should consider using Kotlin DSL build scripts for the superior IDE support when editing them.

Executing Gradle builds

If you follow any of the tutorials linked above, you will execute a Gradle build. But what do you do if you're given a Gradle build without any instructions?

Here are some useful steps to follow:

1. Determine whether the project has a Gradle wrapper and use it if it's there — the main IDEs default to using the wrapper when it's available.
2. Discover the project structure.

   Either import the build with an IDE or run `gradle projects` from the command line. If only the root project is listed, it's a single-project build. Otherwise it's a multi-project build.

3. Find out what tasks you can run.

   If you have imported the build into an IDE, you should have access to a view that displays all the available tasks. From the command line, run `gradle tasks`.

4. Learn more about the tasks via `gradle help --task <taskname>`.

   The `help` task can display extra information about a task, including which projects contain that task and what options the task supports.

5. Run the task that you are interested in.

   Many convention-based builds integrate with Gradle's lifecycle tasks, so use those when you don't have something more specific you want to do with the build. For example, most builds have `clean`, `check`, `assemble` and `build` tasks.

   From the command line, just run `gradle <taskname>` to execute a particular task. You can learn more about command-line execution in the corresponding user manual chapter. If you're using an IDE, check its documentation to find out how to run a task.

Gradle builds often follow standard conventions on project structure and tasks, so if you're familiar

with other builds of the same type — such as Java, Android or native builds — then the file and directory structure of the build should be familiar, as well as many of the tasks and project properties.

For more specialized builds or those with significant customizations, you should ideally have access to documentation on how to run the build and what build properties you can configure.

Authoring Gradle builds

Learning to create and maintain Gradle builds is a process, and one that takes a little time. We recommend that you start with the appropriate core plugins and their conventions for your project, and then gradually incorporate customizations as you learn more about the tool.

Here are some useful first steps on your journey to mastering Gradle:

1. Try one or two basic tutorials to see what a Gradle build looks like, particularly the ones that match the type of project you work with (Java, native, Android, etc.).
2. Make sure you've read What is Gradle?
3. Learn about the fundamental elements of a Gradle build: projects, tasks, and the file API.
4. If you are building software for the JVM, be sure to read about the specifics of those types of projects in Building Java & JVM projects and Testing in Java & JVM projects.
5. Familiarize yourself with the core plugins that come packaged with Gradle, as they provide a lot of useful functionality out of the box.
6. Learn how to author maintainable build scripts and best organize your Gradle projects.

The user manual contains a lot of other useful information and you can find samples demonstrating various Gradle features on the samples pages.

Integrating 3rd-party tools with Gradle

Gradle's flexibility means that it readily works with other tools, such as those listed on our Gradle & Third-party Tools page.

There are two main modes of integration:

- A tool drives Gradle — uses it to extract information about a build and run it — via the Tooling API
- Gradle invokes or generates information for a tool via the 3rd-party tool's APIs — this is usually done via plugins and custom task types

Tools that have existing Java-based APIs are generally straightforward to integrate. You can find many such integrations on Gradle's plugin portal.

Installing Gradle

You can install the Gradle build tool on Linux, macOS, or Windows. This document covers installing using a package manager like SDKMAN! or Homebrew, as well as manual installation.

Use of the Gradle Wrapper is the recommended way to upgrade Gradle.

You can find all releases and their checksums on the releases page.

Prerequisites

Gradle runs on all major operating systems and requires only a Java Development Kit version 8 or higher to run. To check, run `java -version`. You should see something like this:

```
 java -version
java version "1.8.0_151"
Java(TM) SE Runtime Environment (build 1.8.0_151-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.151-b12, mixed mode)
```

Gradle ships with its own Groovy library, therefore Groovy does not need to be installed. Any existing Groovy installation is ignored by Gradle.

Gradle uses whatever JDK it finds in your path. Alternatively, you can set the `JAVA_HOME` environment variable to point to the installation directory of the desired JDK.

See the full compatibility notes for Java, Groovy, Kotlin and Android.

Installing with a package manager

SDKMAN! is a tool for managing parallel versions of multiple Software Development Kits on most Unix-like systems (macOS, Linux, Cygwin, Solaris and FreeBSD). We deploy and maintain the versions available from SDKMAN!.

```
 sdk install gradle
```

Homebrew is "the missing package manager for macOS".

```
 brew install gradle
```

MacPorts is another package manager for macOS.

```
 sudo port install gradle
```

Other package managers are available, but the version of Gradle distributed by them is not controlled by Gradle, Inc. Linux package managers may distribute a modified version of Gradle that is incompatible or incomplete when compared to the official version (available from SDKMAN! or below).

Proceed to next steps

Installing manually

Step 1. Download the latest Gradle distribution

The distribution ZIP file comes in two flavors:

- Binary-only (bin)
- Complete (all) with docs and sources

Need to work with an older version? See the releases page.

Step 2. Unpack the distribution

Linux & MacOS users

Unzip the distribution zip file in the directory of your choosing, e.g.:

```
 mkdir /opt/gradle
 unzip -d /opt/gradle gradle-7.6.6-20260912032715+0000-bin.zip
 ls /opt/gradle/gradle-7.6.6-20260912032715+0000
LICENSE  NOTICE  bin  README  init.d  lib  media
```

Microsoft Windows users

Create a new directory `C:\Gradle` with **File Explorer**.

Open a second **File Explorer** window and go to the directory where the Gradle distribution was downloaded. Double-click the ZIP archive to expose the content. Drag the content folder `gradle-7.6.6-20260912032715+0000` to your newly created `C:\Gradle` folder.

Alternatively, you can unpack the Gradle distribution ZIP into `C:\Gradle` using an archiver tool of your choice.

Step 3. Configure your system environment

To run Gradle, the path to the unpacked files from the Gradle website need to be on your terminal's path. The steps to do this are different for each operating system.

Linux & MacOS users

Configure your `PATH` environment variable to include the `bin` directory of the unzipped distribution, e.g.:

```
 export PATH=$PATH:/opt/gradle/gradle-7.6.6-20260912032715+0000/bin
```

Alternatively, you could also add the environment variable `GRADLE_HOME` and point this to the unzipped distribution. Instead of adding a specific version of Gradle to your `PATH`, you can add `$GRADLE_HOME/bin` to your `PATH`. When upgrading to a different version of Gradle, just change the `GRADLE_HOME` environment variable.

Microsoft Windows users

In **File Explorer** right-click on the `This PC` (or `Computer`) icon, then click `Properties` → `Advanced System Settings` → `Environmental Variables`.

Under `System Variables` select `Path`, then click `Edit`. Add an entry for `C:\Gradle\gradle-7.6.6-20260912032715+0000\bin`. Click OK to save.

Alternatively, you could also add the environment variable `GRADLE_HOME` and point this to the unzipped distribution. Instead of adding a specific version of Gradle to your `Path`, you can add `%GRADLE_HOME%/bin` to your `Path`. When upgrading to a different version of Gradle, just change the `GRADLE_HOME` environment variable.

 Proceed to next steps

Verifying installation

Open a console (or a Windows command prompt) and run `gradle -v` to run gradle and display the version, e.g.:

```
 gradle -v

------------------------------------------------------------
Gradle 7.6.6-20260912032715+0000
------------------------------------------------------------

(environment specific information)
```

If you run into any trouble, see the section on troubleshooting installation.

You can verify the integrity of the Gradle distribution by downloading the SHA-256 file (available from the releases page) and following these verification instructions.

Next steps

Now that you have Gradle installed, use these resources for getting started:

- Create your first Gradle project by following one of our step-by-step samples.
- Sign up for a live introductory Gradle training with a core engineer.
- Learn how to achieve common tasks through the command-line interface.
- Configure Gradle execution, such as use of an HTTP proxy for downloading dependencies.
- Subscribe to the Gradle Newsletter for monthly release and community updates.

Troubleshooting builds

The following is a collection of common issues and suggestions for addressing them. You can get other tips and search the Gradle forums and StackOverflow #gradle answers, as well as Gradle

documentation from help.gradle.org.

Troubleshooting Gradle installation

If you followed the installation instructions, and aren't able to execute your Gradle build, here are some tips that may help.

If you installed Gradle outside of just invoking the Gradle Wrapper, you can check your Gradle installation by running `gradle --version` in a terminal.

You should see something like this:

```
 gradle --version

------------------------------------------------------------
Gradle 6.5
------------------------------------------------------------

Build time:   2020-06-02 20:46:21 UTC
Revision:     a27f41e4ae5e8a41ab9b19f8dd6d86d7b384dad4

Kotlin:       1.3.72
Groovy:       2.5.11
Ant:          Apache Ant(TM) version 1.10.7 compiled on September 1 2019
JVM:          14 (AdoptOpenJDK 14+36)
OS:           Mac OS X 10.15.2 x86_64
```

If not, here are some things you might see instead.

Command not found: gradle

If you get "command not found: gradle", you need to ensure that Gradle is properly added to your `PATH`.

JAVA_HOME is set to an invalid directory

If you get something like:

```
ERROR: JAVA_HOME is set to an invalid directory

Please set the JAVA_HOME variable in your environment to match the location of your
Java installation.
```

You'll need to ensure that a Java Development Kit version 8 or higher is properly installed, the `JAVA_HOME` environment variable is set, and Java is added to your `PATH`.

Permission denied

If you get "permission denied", that means that Gradle likely exists in the correct place, but it is not executable. You can fix this using `chmod +x path/to/executable` on *nix-based systems.

Other installation failures

If `gradle --version` works, but all of your builds fail with the same error, it is possible there is a problem with one of your Gradle build configuration scripts.

You can verify the problem is with Gradle scripts by running `gradle help` which executes configuration scripts, but no Gradle tasks. If the error persists, build configuration is problematic. If not, then the problem exists within the execution of one or more of the requested tasks (Gradle executes configuration scripts first, and then executes build steps).

Debugging dependency resolution

Common dependency resolution issues such as resolving version conflicts are covered in Troubleshooting Dependency Resolution.

You can see a dependency tree and see which resolved dependency versions differed from what was requested by clicking the *Dependencies* view and using the search functionality, specifying the resolution reason.

Figure 1. Debugging dependency conflicts with build scans

The actual build scan with filtering criteria is available for exploration.

Troubleshooting slow Gradle builds

For build performance issues (including “slow sync time”), see improving the Performance of Gradle Builds.

Android developers should watch a presentation by the Android SDK Tools team about Speeding Up Your Android Gradle Builds. Many tips are also covered in the Android Studio user guide on optimizing build speed.

Debugging build logic

Attaching a debugger to your build

You can set breakpoints and debug buildSrc and standalone plugins in your Gradle build itself by setting the `org.gradle.debug` property to "true" and then attaching a remote debugger to port 5005. You can change the port number by setting the `org.gradle.debug.port` property to the desired port number.

To attach the debugger remotely via network, you need to set the `org.gradle.debug.host` property to the machine's IP address or `*` (listen on all interfaces).

```
 gradle help -Dorg.gradle.debug=true
```

In addition, if you've adopted the Kotlin DSL, you can also debug build scripts themselves.

The following video demonstrates how to debug an example build using IntelliJ IDEA.

[remote debug gradle] | *remote-debug-gradle.gif*

Figure 2. Interactive debugging of a build script

Adding and changing logging

In addition to controlling logging verbosity, you can also control display of task outcomes (e.g. "UP-TO-DATE") in lifecycle logging using the `--console=verbose` flag.

You can also replace much of Gradle's logging with your own by registering various event listeners. One example of a custom event logger is explained in the logging documentation. You can also control logging from external tools, making them more verbose in order to debug their execution.

NOTE	Additional logs from the Gradle Daemon can be found under `GRADLE_USER_HOME/daemon/<gradle-version>/`.

Task executed when it should have been UP-TO-DATE

`--info` logs explain why a task was executed, though build scans do this in a searchable, visual way by going to the *Timeline* view and clicking on the task you want to inspect.

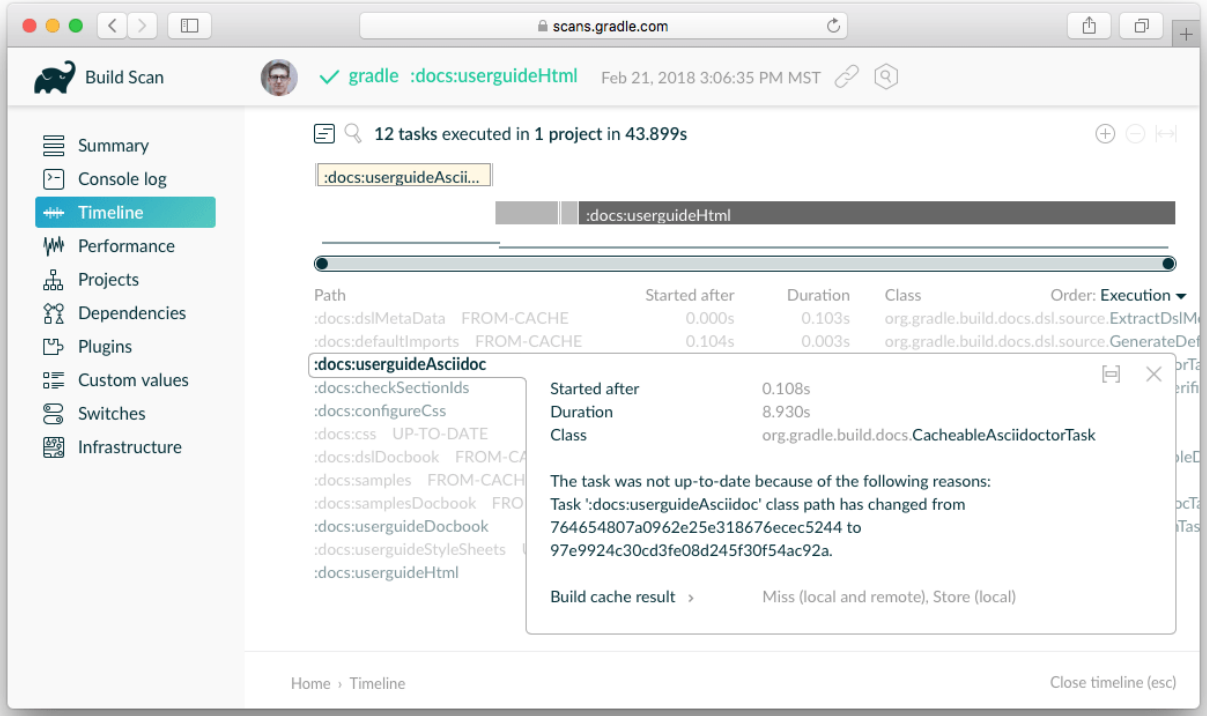


Figure 3. Debugging incremental build with a build scan

You can learn what the task outcomes mean from this listing.

Debugging IDE integration

Many infrequent errors within IDEs can be solved by "refreshing" Gradle. See also more documentation on working with Gradle in IntelliJ IDEA and in Eclipse.

Refreshing IntelliJ IDEA

NOTE: This only works for Gradle projects linked to IntelliJ.

From the main menu, go to `View` > `Tool Windows` > `Gradle`. Then click on the *Refresh* icon.

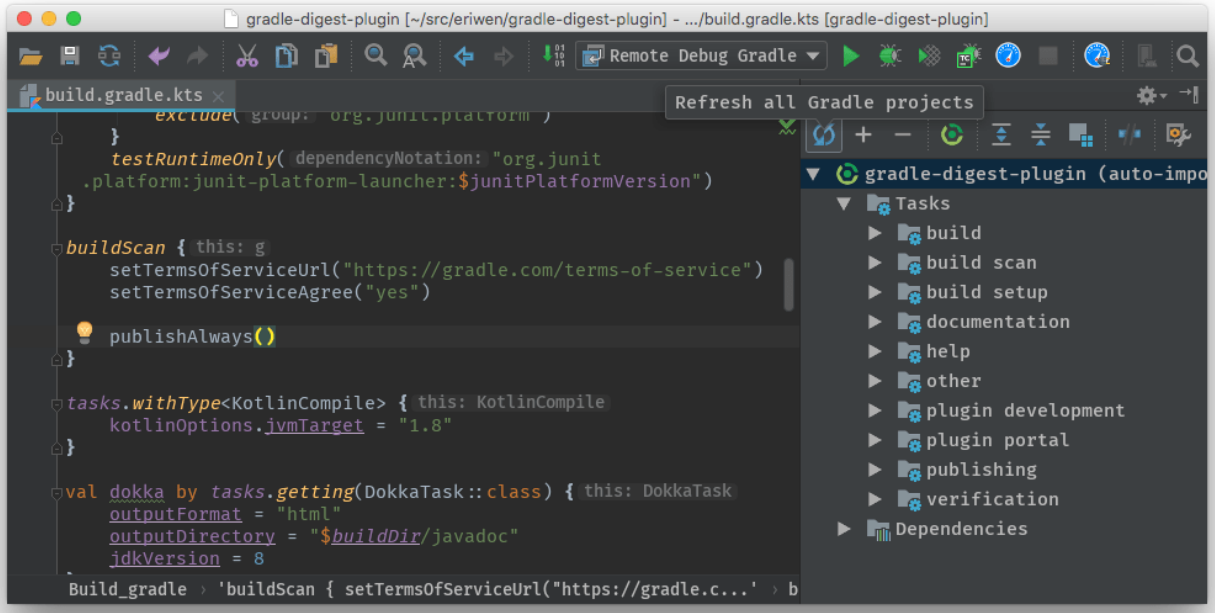


Figure 4. Refreshing a Gradle project in IntelliJ IDEA

Refreshing Eclipse (using Buildship)

If you're using Buildship for the Eclipse IDE, you can re-synchronize your Gradle build by opening the "Gradle Tasks" view and clicking the "Refresh" icon, or by executing the `Gradle` > `Refresh Gradle Project` command from the context menu while editing a Gradle script.

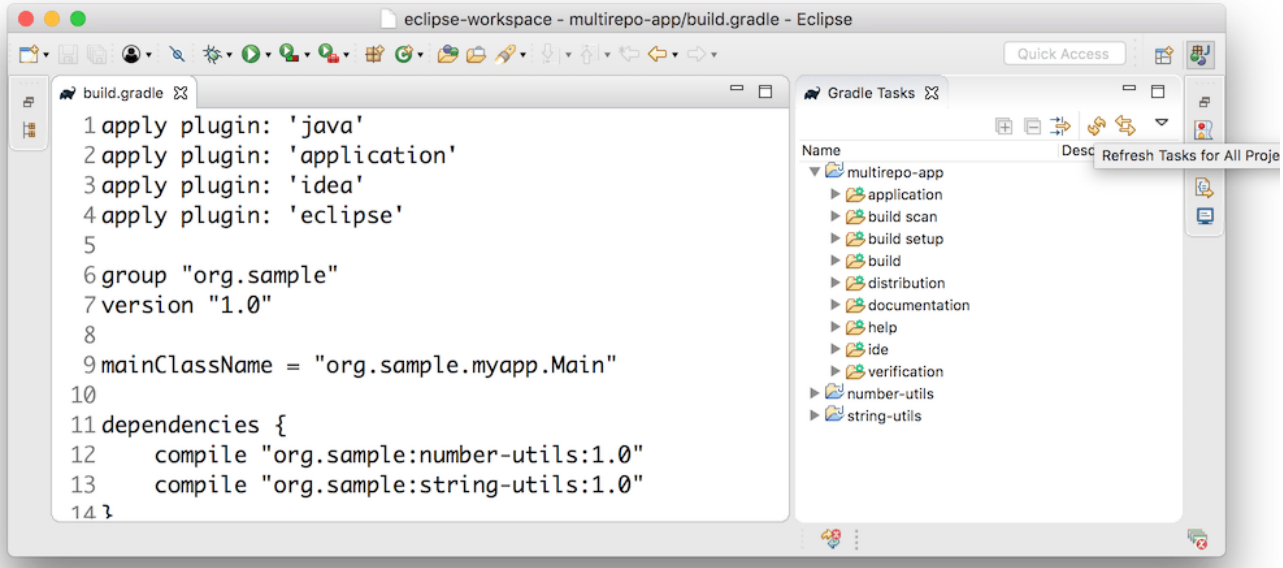


Figure 5. Refreshing a Gradle project in Eclipse Buildship

Troubleshooting daemon connection issues

If your Gradle build fails before running any tasks, you may be encountering problems with your network configuration. When Gradle is unable to communicate with the Gradle daemon process, the build will immediately fail with a message similar to this:

```
$ gradle help

Starting a Gradle Daemon, 1 stopped Daemon could not be reused, use --status for
details

FAILURE: Build failed with an exception.

* What went wrong:
A new daemon was started but could not be connected to: pid=DaemonInfo{pid=55913,
address=[7fb34c82-1907-4c32-afda-888c9b6e2279 port:42751, addresses:[/127.0.0.1]],
state=Busy, ...
```

We have observed this can occur when network address translation (NAT) masquerade is used. When NAT masquerade is enabled, connections that should be considered local to the machine are masked to appear from external IP addresses. Gradle refuses to connect to any external IP address as a security precaution.

The solution to this problem is to adjust your network configuration such that local connections are not modified to appear as from external addresses.

You can monitor the detected network setup and the connection requests in the daemon log file (`<GRADLE_USER_HOME>/daemon/<Gradle version>/daemon-<PID>.out.log`).

```
2021-08-12T12:01:50.755+0200 [DEBUG]
[org.gradle.internal.remote.internal.inet.InetAddresses] Adding IP addresses for
network interface enp0s3
2021-08-12T12:01:50.759+0200 [DEBUG]
[org.gradle.internal.remote.internal.inet.InetAddresses] Is this a loopback interface?
false
2021-08-12T12:01:50.769+0200 [DEBUG]
[org.gradle.internal.remote.internal.inet.InetAddresses] Adding remote address
/fe80:0:0:0:85ba:3f3e:1b88:c0e1%enp0s3
2021-08-12T12:01:50.770+0200 [DEBUG]
[org.gradle.internal.remote.internal.inet.InetAddresses] Adding remote address
/10.0.2.15
2021-08-12T12:01:50.770+0200 [DEBUG]
[org.gradle.internal.remote.internal.inet.InetAddresses] Adding IP addresses for
network interface lo
2021-08-12T12:01:50.771+0200 [DEBUG]
[org.gradle.internal.remote.internal.inet.InetAddresses] Is this a loopback interface?
true
2021-08-12T12:01:50.771+0200 [DEBUG]
[org.gradle.internal.remote.internal.inet.InetAddresses] Adding loopback address
/0:0:0:0:0:0:0:1%lo
2021-08-12T12:01:50.771+0200 [DEBUG]
[org.gradle.internal.remote.internal.inet.InetAddresses] Adding loopback address
/127.0.0.1
2021-08-12T12:01:50.775+0200 [DEBUG]
[org.gradle.internal.remote.internal.inet.TcpIncomingConnector] Listening on
[7fb34c82-1907-4c32-afda-888c9b6e2279 port:42751, addresses:[localhost/127.0.0.1]].
...
2021-08-12T12:01:50.797+0200 [INFO]
[org.gradle.launcher.daemon.server.DaemonRegistryUpdater] Advertising the daemon
address to the clients: [7fb34c82-1907-4c32-afda-888c9b6e2279 port:42751,
addresses:[localhost/127.0.0.1]]
...
2021-08-12T12:01:50.923+0200 [ERROR]
[org.gradle.internal.remote.internal.inet.TcpIncomingConnector] Cannot accept
connection from remote address /10.0.2.15.
```

Getting additional help

If you didn't find a fix for your issue here, please reach out to the Gradle community on the help forum or search relevant developer resources using help.gradle.org.

If you believe you've found a bug in Gradle, please file an issue on GitHub.

Compatibility Matrix

The sections below describe Gradle's compatibility with several integrations. Other versions not listed here may or may not work.

Java

A Java version between 8 and 19 is required to execute Gradle. Java 20 and later versions are not yet supported.

Java 6 and 7 can still be used for compilation and forked test execution.

Any supported version of Java can be used for compile or test.

For older Gradle versions, please see the table below which Java version is supported by which Gradle release.

Table 1. Java Compatibility

Java version	First Gradle version to support it
8	2.0
9	4.3
10	4.7
11	5.0
12	5.4
13	6.0
14	6.3
15	6.7
16	7.0
17	7.3
18	7.5
19	7.6

Kotlin

Gradle is tested with Kotlin 1.3.72 through 1.7.10.

Gradle plugins written in Kotlin target Kotlin 1.4 for compatibility with Gradle and Kotlin DSL build scripts, even though the embedded Kotlin runtime is Kotlin 1.7.

Table 2. Embedded Kotlin version

Gradle version	Embedded Kotlin version	Kotlin Language version
5.0	1.3.10	1.3
5.1	1.3.11	1.3
5.2	1.3.20	1.3
5.3	1.3.21	1.3
5.5	1.3.31	1.3

5.6	1.3.41	1.3
6.0	1.3.50	1.3
6.1	1.3.61	1.3
6.3	1.3.70	1.3
6.4	1.3.71	1.3
6.5	1.3.72	1.3
6.8	1.4.20	1.3
7.0	1.4.31	1.4
7.2	1.5.21	1.4
7.3	1.5.31	1.4
7.5	1.6.21	1.4
7.6	1.7.10	1.4

Groovy

Gradle is tested with Groovy 1.5.8 through 4.0.0.

Gradle plugins written in Groovy must use Groovy 3.x for compatibility with Gradle and Groovy DSL build scripts.

Android

Gradle is tested with Android Gradle Plugin 4.1, 4.2, 7.0, 7.1, 7.2, 7.3 and 7.4. Alpha and beta versions may or may not work.

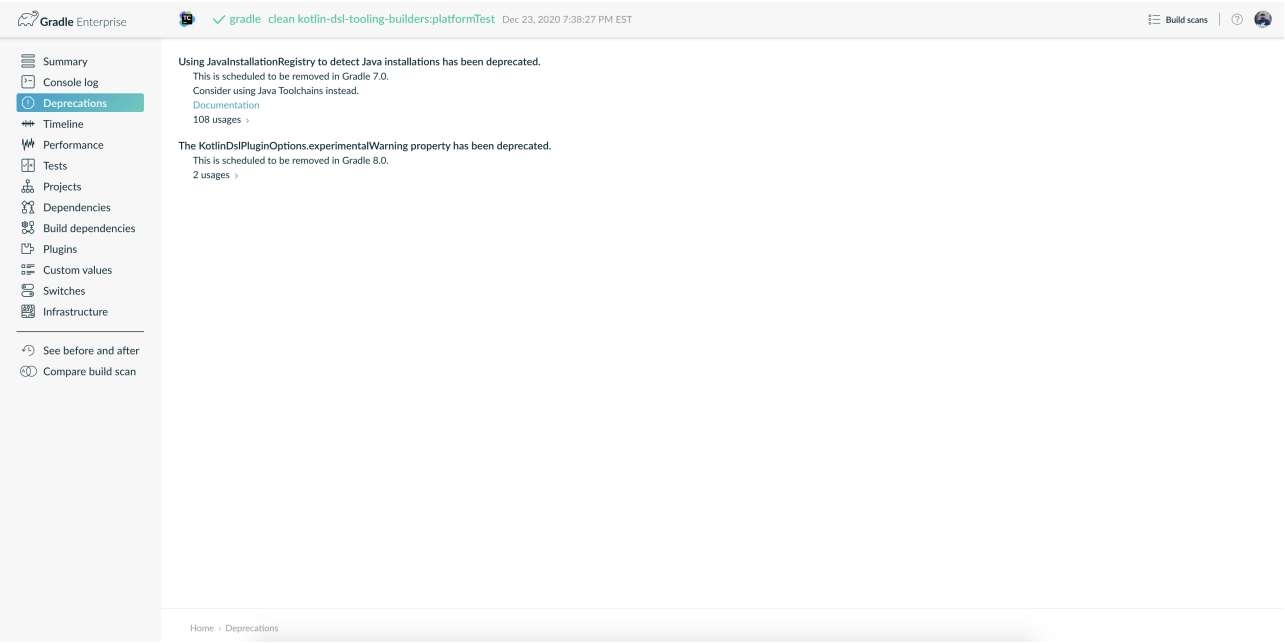
Upgrading and Migrating

Upgrading your build from Gradle 6.x to 7.0

This chapter provides the information you need to migrate your Gradle 6.x builds to the latest Gradle release. For migrating from Gradle 4.x or 5.x, see the older migration guide first.

We recommend the following steps for all users:

1. Try running `gradle help --scan` and view the deprecations view of the generated build scan.



   This is so that you can see any deprecation warnings that apply to your build.

   Alternatively, you could run `gradle help --warning-mode=all` to see the deprecations in the console, though it may not report as much detailed information.

2. Update your plugins.

   Some plugins will break with this new version of Gradle, for example because they use internal APIs that have been removed or changed. The previous step will help you identify potential problems by issuing deprecation warnings when a plugin does try to use a deprecated part of the API.

3. Run `gradle wrapper --gradle-version 7.0` to update the project to 7.0.
4. Try to run the project and debug any errors using the Troubleshooting Guide.

Upgrading from 6.9

Changes in the IDE integration

Changes in the IDEA model

The `getGeneratedSourceDirectories()` and `getGeneratedTestDirectories()` methods are removed from the `IdeaContentRoot` interface. Clients should replace these invocations with `getSourceDirectories()` and `getTestDirectories()` and use the `isGenerated()` method on the returned instances.

Dependency locking now defaults to a single file per project

The format of the dependency lockfile has been changed and as a consequence there is only one file per project instead of one file per configuration per project. This change only affects *writing* lock files. Gradle remains capable of *loading* lock state saved in the older format.

Head over to the documentation to learn how to migrate to the new format. The migration can be performed per configuration and does not have to be done in a single step. Gradle will automatically clean up previous lock files when migrating them over to the new file format.

Gradle Module Metadata is now reproducible by default

The `buildId` field will not be populated by default to ensure that the produced metadata file remains unchanged when no build inputs are changed. Users can still opt in to have this unique identifier part of the produced metadata if they want to, see the documentation.

The `jcenter()` convenience method is now deprecated

JFrog announced the sunset of the JCenter repository in February 2021. Many Gradle builds rely on JCenter for project dependencies.

No new packages or versions are published to JCenter, but JFrog says they will keep JCenter running in a read-only state indefinitely. We recommend that you consider using `mavenCentral()`, `google()` or a private `maven` repository instead.

Gradle emits a deprecation warning when `jcenter()` is used as a repository and this method is scheduled to be removed in Gradle 8.0.

Potential breaking changes

Updates to bundled Gradle dependencies

- Kotlin has been updated to Kotlin 1.4.31.
- Groovy has been updated to Groovy 3.0.7.

Changes to Groovy and Groovy DSL

Due to the update to the next major version of Groovy, you may experience minor issues when upgrading to Gradle 7.0.

The new version of Groovy has a stricter parser that fails to compile code that may have been accepted in previous Groovy versions. If you encounter syntax errors, check the Groovy issue tracker and Groovy 3 release highlights.

Some very specific regressions have already been fixed in the next minor version of Groovy.

Groovy modularization

Gradle no longer embeds a copy of `groovy-all` that bundles all Groovy modules into a single jar—only the most important modules are distributed in the Gradle distribution.

The `localGroovy()` dependency will include these Groovy modules:

- `groovy`
- `groovy-ant`
- `groovy-astbuilder`
- `groovy-console`
- `groovy-datetime`
- `groovy-dateutil`
- `groovy-groovydoc`
- `groovy-json`
- `groovy-nio`
- `groovy-sql`
- `groovy-templates`
- `groovy-test`
- `groovy-xml`

But the following Groovy modules are **not** included:

- `groovy-cli-picocli`
- `groovy-docgenerator`
- `groovy-groovysh`
- `groovy-jmx`
- `groovy-jsr223`
- `groovy-macro`
- `groovy-servlet`
- `groovy-swing`
- `groovy-test-junit5`
- `groovy-testng`

You can pull these dependencies into your build like any other external dependency.

Building Gradle plugins with Groovy 3

Plugins built with Gradle 7.0 will now have Groovy 3 on their classpath when using `gradleApi()` or `localGroovy()`.

NOTE	If you use Spock to test your plugins, you will need to use Spock 2.x. There are no compatible versions of Spock 1.x and Groovy 3.

```
dependencies {
    // Ensure you use the Groovy 3.x variant
    testImplementation('org.spockframework:spock-core:2.0-groovy-3.0') {
        exclude group: 'org.codehaus.groovy'
    }
}

// Spock 2 is based on JUnit Platform which needs to be enabled explicitly.
tasks.withType(Test).configureEach {
    useJUnitPlatform()
}
```

Performance

Depending on the number of subprojects and Groovy DSL build scripts, you may notice a performance regression when compiling build scripts for the first time or when changes are made to the build script's classpath. This is due to the slower performance of the Groovy 3 parser, but the Groovy team is aware of the issue and trying to mitigate the regression.

In general, we are also looking at how we can improve the performance of build script compilation for both Groovy DSL and Kotlin DSL.

Encountering 'Could not find method X for arguments Y on DefaultDependencyHandler'

While the following error initially looks like a compile error, it is actually due to the fact that specific `Configuration`s have been removed. Please refer to Removal of `compile` and `runtime` configurations for more details.

```
Could not find method testCompile() for arguments
[DefaultExternalModuleDependency{group='org.junit', name='junit-bom', version='5.7.0',
configuration='default'}] on object of type
org.gradle.api.internal.artifacts.dsl.dependencies.DefaultDependencyHandler.
```

Updates to default tool integration versions

- PMD has been updated to PMD 6.31.0.
- Groovy and GroovyDoc have been updated to Groovy 3.0.7.

Removal of `compile` and `runtime` configurations

Since its inception, Gradle provided the `compile` and `runtime` configurations to declare dependencies. These however did not support a fine grained scoping of dependencies. Hence, better replacements were introduced in Gradle 3.4:

- The `implementation` configuration should be used to declare dependencies which are *implementation details* of a library: they are not visible to consumers of the library during compilation time.
- The `api` configuration, available only if you apply the `java-library` plugin, should be used to declare dependencies which are part of the API of a library, that need to be exposed to consumers at compilation time.

In Gradle 7, both the `compile` and `runtime` configurations are removed. Therefore, you have to migrate to the `implementation` and `api` configurations above. If you are still using the `java` plugin for a Java library, you will need to apply the `java-library` plugin instead.

Table 3. Common configuration upgrades

Removed Configuration	New Configuration
`compile`	`api` or `implementation`
`runtime`	`runtimeOnly`
`testRuntime`	`testRuntimeOnly`
`testCompile`	`testImplementation`
`<sourceSet>Runtime`	`<sourceSet>RuntimeOnly`
`<sourceSet>Compile`	`<sourceSet>Implementation`

You can find more details about the benefits of the new configurations and which one to use in place of `compile` and `runtime` by reading the Java Library plugin documentation.

WARNING

When using the Groovy DSL, you need to watch out for a particular upgrade problem when dealing with the removed configurations.

If you were creating custom configurations that extend one of the removed configurations, Gradle may silently create configurations that do not exist.

This looks something like:

```
configurations {
  // This silently creates a configuration called "runtime"
  myConf extendsFrom runtime
}
```

The result of dependency resolution for your custom configuration may not be the same as Gradle 6.x or before. You may notice missing dependencies or artifacts.

Location of temporary project files for `ProjectBuilder`

The `ProjectBuilder` API is used for inspecting Gradle builds in unit tests. This API used to create temporary project files under the system temporary directory as defined by `java.io.tmpdir`.

The API now creates temporary project files under the `Test` task's temporary directory. This path is

usually under the project build directory. This may cause test failures when the test expects particular file paths.

If the test uses `ProjectBuilder.withProjectDir(…)`, it is unaffected.

Location of temporary files for TestKit tests

Tests that use the TestKit API used to create temporary files under the system temporary directory as defined by `java.io.tmpdir`. These files were used to store copies of Gradle distributions or another test-only Gradle User Home.

TestKit tests will now create temporary files under the `Test` task's temporary directory. This path is usually under the project build directory. This may cause test failures when the test expects particular file paths.

If the test uses `GradleRunner.withTestKitDir(…)`, it is unaffected.

File system watching with TestKit on Windows

The file system watching implementation on Windows adds a lock to the root project directory in order to watch for changes. This may cause errors when you try to delete the root project directory after running a build with TestKit. For example, tests that use TestKit together with JUnit's `@TempDir` extension, or the `TemporaryFolder` rule can run into this problem. To avoid problems with these file locks, `TestKit` disables file system watching for builds executed on Windows via `GradleRunner`. If you'd like to override the default behavior, you can enable file system watching by passing `--watch-fs` to `GradleRunner.withArguments()`.

Removal of the legacy `maven` plugin

The `maven` plugin has been removed. You should use the `maven-publish` plugin instead.

Please refer to the documentation of the Maven Publish plugin for more details.

Removal of the `uploadArchives` task

The `uploadArchives` task was used in combination with the legacy Ivy or Maven publishing mechanisms. It has been removed in Gradle 7. You should migrate to the `maven-publish` or `ivy-publish` plugin instead.

Please refer to the documentation of the Maven Publish plugin for publishing on Maven repositories. Please refer to the documentation of the Ivy Publish plugin for publishing on Ivy repositories.

Changes in dependency version sorting

In the context of dependency version sorting, a `-SNAPSHOT` version is now considered to be right before a final release but after any `-RC` version. More special version suffixes are also taken into account. This brings the Gradle algorithm closer to the Maven one for well-known version suffixes.

Have a look at the documentation for all the rules Gradle applies.

Removal of Play Framework plugins

The deprecated Play plugins have been removed. An external replacement, the Play Framework plugin, is available from the plugin portal.

Removal of deprecated JVM plugins

These unmaintained alternative JVM plugins have been removed: `java-lang`, `scala-lang`, `junit-test-suite`, `jvm-component`, `jvm-resources`.

Please use the stable Java Library and Scala plugins instead.

Removal of experimental JavaScript plugins

The following plugins for experimental JavaScript integration are now removed from the distribution: `coffeescript-base`, `envjs`, `javascript-base`, `jshint`, `rhino`.

If you used these plugins despite their experimental nature, you may find suitable replacements in the Plugin Portal.

Configuring the layout of an Ivy repository

The `layout` method taking a configuration block has been removed and is replaced by patternLayout.

Executing a Gradle build without a settings file is now an error

A Gradle build is defined by its `settings.gradle(.kts)` file found in the current or parent directory. Without a settings file, a Gradle build is undefined and Gradle produces an error when attempting to execute tasks.

To fix this error, create a `settings.gradle(.kts)` file for the build.

Exceptions to this are invoking Gradle with the `init` task or using diagnostic command line flags, such as `--version`.

Calling Project.afterEvaluate() after project evaluation is now an error

Gradle 6.x warns users about the wrong behavior and ignores the target action in this scenario. Starting from 7.0 the same case will produce an error. Plugins and build scripts should be adjusted to call `afterEvaluate` only at configuration time. If you have such a build failure and the related `afterEvaluate` statement is declared in your build sources then you can simply delete it. If `afterEvaluate` is declared in a plugin then report the issue to the plugin maintainers.

Modifying file collections after values finalized is now an error

Calling any mutator methods (i.e. `clear()`, `add()`, `remove()`, etc.) on `ConfigurableFileCollection` after the stored value calculated throws an exception. Users and plugin authors should adjust their code such that all configuration on `ConfigurableFileCollection` happens during configuration time, before the values are read.

Removal of `ProjectLayout#configurableFiles`

Please use `ObjectFactory#fileCollection()` instead.

Removal of `BasePluginConvention.libsDir` and `BasePluginConvention.distsDir`

Please use the `libsDirectory` and `distsDirectory` properties instead.

Removal of `UnableToDeleteFileException`

Existing usages should be replaced with `RuntimeException`.

Properties removed in Checkstyle and PMD plugins

- The `configDir` getters and setters have been removed from the Checkstle task and extension. Use the `configDirectory` property instead.
- The `rulePriority` getter and setter have been removed from the Pmd task and extension. Use the `rulesMinimumPriority` property instead.

Removal of `baseName` property in `distribution` plugin

The `getBaseName()` and `setBaseName()` methods were removed from the `Distribution` class. Clients should replace the usages with the `distributionBaseName` property.

Using `AbstractTask`

Registering a task with the `AbstractTask` type or with a type extending `AbstractTask` was deprecated in Gradle 6.5 and is now an error in Gradle 7.0. You can use DefaultTask instead.

Removal of `BuildListener.buildStarted(Gradle)`

`BuildListener.buildStarted(Gradle)` was deprecated in Gradle 6.0 and is now removed in Gradle 7.0. Please use BuildListener.beforeSettings(Settings) instead.

Removal of unused `StartParameter` APIs

The following APIs, which were not usable via command line options anymore since Gradle 5.0, are now removed: `StartParameter.useEmptySettings()`, `StartParameter.isUseEmptySettings()`, `StartParameter.setSearchUpwards(boolean)` and `StartParameter.isSearchUpwards()`.

Removal of searching for settings files in 'master' directories

Gradle no longer supports discovering the settings file in a directory named `master` in a sibling directory. If your build still uses this deprecated feature, consider refactoring the build to have the root directory match the physical root of the project hierarchy. You can find more information about how to structure a Gradle build or a composition of builds in the user manual. Alternatively, you can still run tasks in builds like this by invoking the build from the `master` directory only using a fully qualified path to the task.

`modularity.inferModulePath` defaults to 'true'

Compiling, testing and executing now works automatically for any source set that defines a module

by containing a `module-info.java` file. Usually, this is the behavior you need. If this is causing issues in cases you manually configure the module path, or use a 3rd party plugin for it, you can still opt out of this by setting `modularity.inferModulePath` to `false` on the java extension or individual tasks.

Removal of `ValidateTaskProperties`

The `ValidateTaskProperties` task has been removed and replaced by the ValidatePlugins task.

Removal of `ImmutableFileCollection`

The `ImmutableFileCollection` type has been removed. Use the factory method instead. A handle to the project layout can be obtained via Project.layout.

Removal of `ComponentSelectionReason.getDescription`

The method `ComponentSelectionReason.getDescription` has been removed. It is replaced by `ComponentSelectionReason.getDescriptions` which returns a list of `ComponentSelectionDescriptor`, each having a `getDescription`.

Removal of domain object collection constructors

The following deprecated constructors were removed:

- DefaultNamedDomainObjectList(Class, Instantiator, Namer)
- DefaultNamedDomainObjectSet(Class, Instantiator)
- DefaultPolymorphicDomainObjectContainer(Class, Instantiator)
- FactoryNamedDomainObjectContainer(Class, Instantiator, NamedDomainObjectFactory)

Removal of arbitrary local cache configuration

The local build cache configuration now needs to be done via BuildCacheConfiguration.local().

Removal of DefaultVersionSelectorScheme constructor

This internal API was used in plugins, amongst other the Nebula plugins, and was deprecated in the Gradle 5.x timeline and is now removed. Latest plugins version should no longer reference it.

Setting the `config_loc` config property on the `checkstyle` plugin is now an error

The `checkstyle` plugin now fails for the following configuration

```
checkstyle {
    configProperties['config_loc'] = file("path/to/checkstyle-config-dir")
}
```

Builds should declare the checkstyle configuration with the `checkstyle` block:

```
checkstyle {
    configDirectory = file("path/to/checkstyle-config-dir")
}
```

Querying the mapped value of a provider before the producer has completed is now an error

Gradle 6.x warns users about the wrong behavior and then returns a possibly incorrect provider value. Starting with 7.0 the same case will produce an error. Plugins and build scripts should be adjusted to query the mapped value of a provider, for example a task output property, after the task has completed.

Task validation problems are now errors

Gradle 6.0 started warning about problems with task definitions (such as incorrectly defined inputs or outputs). For Gradle 7.0, those warnings are now errors and will fail the build.

Change in behavior when there's a strict version conflict with a local project

Previous Gradle releases had an inconsistent behavior in regard to conflict resolution in a particular configuration: - your project declares a strict dependency on a published module (for example, `com.mycompany:some-module:1.2!!`, where `1.2!!` is the short hand notation for a strict dependency on 1.2) - your build actually provides `com.mycompany:some-module` in a higher version

Previous Gradle releases would succeed, selecting the project dependency despite the strict constraint. Starting from Gradle 7, this will trigger a dependency resolution failure.

See this issue for more context.

Deprecations

Missing dependencies between tasks

Having a task which produces an output in a location and another task consuming that location by referring to it as an input without the consumer task depending on the producer task has been deprecated. A fix for this problem is to add a dependency from the consumer to the producer.

Duplicates strategy

Gradle 7 now fails when a copy operation (or any operation which uses a `org.gradle.api.file.CopySpec`) encounters a duplicate entry, and that the duplicates strategy isn't set. Please look at the CopySpec docs for details.

Upgrading from 6.8

No upgrade notes from 6.8 to 6.9, as 6.9 only contains bug fixes.

Upgrading from 6.7

Potential breaking changes

Toolchain API is now marked as @NonNull

The API supporting the Java Toolchain feature in `org.gradle.jvm.toolchain` is now marked as `@NonNull`.

This may impact Kotlin consumers where the return types of APIs are no longer nullable.

Updates to default tool integration versions

- JaCoCo has been updated to 0.8.6.
- Checkstyle has been updated to Checkstyle 8.37.
- CodeNarc has been updated to CodeNarc 2.0.0.

Updates to bundled Gradle dependencies

- Kotlin has been updated to Kotlin 1.4.20. Note that Gradle scripts are still using the Kotlin 1.3 language.
- Apache Ant has been updated to 1.10.9 to fix CVE-2020-11979

Projects imported into Eclipse now include custom source set classpaths

Previously, projects imported by Eclipse only included dependencies for the main and test source sets. The compile and runtime classpaths of custom source sets were ignored.

Since Gradle 6.8, projects imported into Eclipse include the compile and runtime classpath for every source set defined by the build.

SourceTask is no longer sensitive to empty directories

Previously, empty directories would be taken into account during up-to-date checks and build cache key calculations for the sources declared in `SourceTask`. This meant that a source tree that contained an empty directory and an otherwise identical source tree that did not contain the empty directory would be considered different sources, even if the task would produce the same outputs. In Gradle 6.8, `SourceTask` now ignores empty directories during doing up-to-date checks and build cache key calculations. In the vast majority of cases, this is the desired behavior, but it is possible that a task may extend `SourceTask` but also produce different outputs when empty directories are present in the sources. For tasks where this is a concern, you can expose a separate property without the `@IgnoreEmptyDirectories` annotation in order to capture those changes:

```
@InputFiles
@SkipWhenEmpty
@PathSensitive(PathSensitivity.ABSOLUTE)
public FileTree getSourcesWithEmptyDirectories() {
    return super.getSource()
}
```

Changes to publications

Publishing a component which has a dependency on an enforced platform now triggers a validation error, preventing accidental publishing of bad metadata: enforced platforms use cases should be limited to applications, not things which can be consumed from another library or an application.

If, for some reason, you still want to publish components with dependencies on enforced platforms, you can disable the validation following the documentation.

Changing default excludes during the execution phase

Gradle's file trees apply some default exclude patterns for convenience — the same defaults as Ant in fact. See the user manual for more information. Sometimes, Ant's default excludes prove problematic, for example when you want to include the `.gitignore` in an archive file.

Changing Gradle's default excludes during the execution phase can lead to correctness problems with up-to-date checks. As a consequence, you are only allowed to change Gradle's default excludes in the settings script, see the user manual for an example.

Deprecations

Referencing tasks from included builds

Direct references to tasks from included builds in `mustRunAfter`, `shouldRunAfter` and `finalizedBy` task methods have been deprecated. Task ordering using `mustRunAfter` and `shouldRunAfter` as well as finalizers specified by `finalizedBy` should be used for task ordering within a build. If you happen to have cross-build task ordering defined using above mentioned methods, consider restructuring such builds and decoupling them from one another.

Searching for settings files in 'master' directories

Gradle will emit a deprecation warning when your build relies on finding the settings file in a directory named `master` in a sibling directory.

If your build uses this feature, consider refactoring the build to have the root directory match the physical root of the project hierarchy.

Alternatively, you can still run tasks in builds like this by invoking the build from the `master` directory only using a fully qualified path to the task.

Using method `NamedDomainObjectContainer<T>.invoke(kotlin.Function1)`

Gradle Kotlin DSL extensions have been changed to favor Gradle's `Action<T>` type over Kotlin function types.

While the change should be transparent to Kotlin clients, Java clients calling Kotlin DSL extensions need to be updated to use the `Action<T>` APIs.

Upgrading from 6.6

Potential breaking changes

buildSrc can now see included builds from the root

Previously, `buildSrc` was built in such a way that included builds were ignored from the root build.

Since Gradle 6.7, `buildSrc` can see any included build from the root build. This may cause dependencies to be substituted from an included build in `buildSrc`. This may also change the order in which some builds are executed if an included build is needed by `buildSrc`.

Updates to default tool integration versions

- PMD has been updated to PMD 6.26.0.
- Checkstyle has been updated to Checkstyle 8.35.
- CodeNarc has been updated to CodeNarc 1.6.1.

Deprecations

Changing default excludes during the execution phase

Gradle's file trees apply some default exclude patterns for convenience — the same defaults as Ant in fact. See the user manual for more information. Sometimes, Ant's default excludes prove problematic, for example when you want to include the `.gitignore` in an archive file.

Changing Gradle's default excludes during the execution phase can lead to correctness problems with up-to-date checks, and is deprecated. You are only allowed to change Gradle's default excludes in the settings script, see the user manual for an example.

Using a Configuration directly as a dependency

Gradle allowed instances of `Configuration` to be used directly as dependencies:

```
dependencies {
    implementation(configurations.myConfiguration)
}
```

This behavior is now deprecated as it is confusing: one could expect the "dependent configuration" to be resolved first and add the result of resolution as dependencies to the including configuration, which is not the case. The deprecated version can be replaced with the actual behavior, which is configuration inheritance:

```
configurations.implementation.extendsFrom(configurations.myConfiguration)
```

Upgrading from 6.5

Potential breaking changes

Updates to bundled Gradle dependencies

- Ant has been updated to 1.10.8.
- Groovy has been updated to Groovy 2.5.12.

Dependency substitutions and variant aware dependency resolution

While adding support for expressing variant support in dependency substitutions, a bug fix introduced a behaviour change that some builds may rely upon. Previously a substituted dependency would still use the attributes of the original selector instead of the ones from the replacement selector.

With that change, existing substitutions around dependencies with richer selectors, such as for platform dependencies, will no longer work as they did. It becomes mandatory to define the variant aware part in the target selector.

You can be affected by this change if you:

- have dependencies on platforms, like `implementation platform("org:platform:1.0")`
- *or* if you specify attributes on dependencies,
- *and* you use resolution rules on these dependencies.

See the documentation for resolving issues if you are impacted.

Deprecations

No deprecations were made in Gradle 6.6.

Upgrading from 6.4

Potential breaking changes

Updates to bundled Gradle dependencies

- Kotlin has been updated to Kotlin 1.3.72.
- Groovy has been updated to Groovy 2.5.11.

Updates to default tool integration versions

- PMD has been updated to PMD 6.23.0.

Deprecations

Internal class AbstractTask is deprecated

`AbstractTask` is an internal class which is visible on the public API, as a superclass of public type

`DefaultTask`. `AbstractTask` will be removed in Gradle 7.0, and the following are deprecated in Gradle 6.5:

- Registering a task whose type is `AbstractTask` or `TaskInternal`. You can remove the task type from the task registration and Gradle will use `DefaultTask` instead.
- Registering a task whose type is a subclass of `AbstractTask` but not a subclass of `DefaultTask`. You can change the task type to extend `DefaultTask` instead.
- Using the class `AbstractTask` from plugin code or build scripts. You can change the code to use `DefaultTask` instead.

Upgrading from 6.3

Potential breaking changes

PMD plugin expects PMD 6.0.0 or higher by default

Gradle 6.4 enabled incremental analysis by default. Incremental analysis is only available in PMD 6.0.0 or higher. If you want to use an older PMD version, you need to disable incremental analysis:

```
pmd {
    incrementalAnalysis = false
}
```

Changes in dependency locking

With Gradle 6.4, the incubating API for dependency locking `LockMode` has changed. The value is now set via a `Property<LockMode>` instead of a direct setter. This means that the notation to set the value has to be updated for the Kotlin DSL:

```
dependencyLocking {
    lockMode.set(LockMode.STRICT)
}
```

Users of the Groovy DSL should not be impacted as the notation `lockMode = LockMode.STRICT` remains valid.

Java versions in published metadata

If a Java library is published with Gradle Module Metadata, the information which Java version it supports is encoded in the `org.gradle.jvm.version` attribute. By default, this attribute was set to what you configured in `java.targetCompatibility`. If that was not configured, it was set to the current Java version running Gradle. Changing the version of a particular compile task, e.g. `javaCompile.targetCompatibility` had no effect on that attribute, leading to wrong information if the attribute was not adjusted manually. This is now fixed and the attribute defaults to the setting of the compile task that is associated with the sources from which the published jar is built.

Ivy repositories with custom layouts

Gradle versions from 6.0 to 6.3.x included could generate bad Gradle Module Metadata when publishing on an Ivy repository which had a custom repository layout. Starting from 6.4, Gradle will no longer publish Gradle Module Metadata if it detects that you are using a custom repository layout.

New properties may shadow variables in build scripts

This release introduces some new properties — `mainClass`, `mainModule`, `modularity` — in different places. Since these are very generic names, there is a chance that you use one of them in your build scripts as variable name. A new property might then shadow one of your variables in an undesired way, leading to a build failure where the property is accessed instead of the local variable with the same name. You can fix it by renaming the corresponding variable in the build script.

Affected is configuration code inside the `application {}` and `java {}` configuration blocks, inside a java execution setup with `project.javaexec {}`, and inside various task configurations (`JavaExec`, `CreateStartScripts`, `JavaCompile`, `Test`, `Javadoc`).

Updates to bundled Gradle dependencies

- Kotlin has been updated to Kotlin 1.3.71.

Deprecations

There were no deprecations between Gradle 6.3 and 6.4.

Upgrading from 6.2

Potential breaking changes

Fewer dependencies available in IDEA

Gradle no longer includes the annotation processor classpath as provided dependencies in IDEA. The dependencies IDEA sees at compile time are the same as what Gradle sees after resolving the compile classpath (configuration named `compileClasspath`). This prevents the leakage of annotation processor dependencies into the project's code.

Before Gradle introduced incremental annotation processing support, IDEA required all annotation processors to be on the compilation classpath to be able to run annotation processing when compiling in IDEA. This is no longer necessary because Gradle has a separate annotation processor classpath. The dependencies for annotation processors are not added to an IDEA module's classpath when a Gradle project with annotation processors is imported.

Updates to bundled Gradle dependencies

- Kotlin has been updated to Kotlin 1.3.70.
- Groovy has been updated to Groovy 2.5.10.

Updates to default tool integration versions

- PMD has been updated to PMD 6.21.0.
- CodeNarc has been updated to CodeNarc 1.5.

Rich console support removed for some 32-bit operating systems

Gradle 6.3 does not support the rich console for 32-bit Unix systems and for old FreeBSD versions (older than FreeBSD 10). Microsoft Windows 32-bit is unaffected.

Gradle will continue building projects on 32-bit systems but will no longer show the rich console.

Deprecations

Using default and archives configurations

Almost every Gradle project has the *default* and *archives* configurations which are added by the *base* plugin. These configurations are no longer used in modern Gradle builds that use variant aware dependency management and the new publishing plugins.

While the configurations will stay in Gradle for backwards compatibility for now, using them to declare dependencies or to resolve dependencies is now deprecated.

Resolving these configurations was never an intended use case and only possible because in earlier Gradle versions *every* configuration was resolvable. For declaring dependencies, please use the configurations provided by the plugins you use, for example by the Java Library plugin.

Upgrading from 6.1

Potential breaking changes

Compile and runtime classpath now request library variants by default

A classpath in a JVM project now explicitly requests the `org.gradle.category=library` attribute. This leads to clearer error messages if a certain library cannot be used. For example, when the library does not support the required Java version. The practical effect is that now all platform dependencies have to be declared as such. Before, platform dependencies also worked, accidentally, when the `platform()` keyword was omitted for local platforms or platforms published with Gradle Module Metadata.

Properties from project root `gradle.properties` leaking into `buildSrc` and included builds

There was a regression in Gradle 6.2 and Gradle 6.2.1 that caused Gradle properties set in the project root `gradle.properties` file to leak into the `buildSrc` build and any builds included by the root.

This could cause your build to start failing if the `buildSrc` build or an included build suddenly found an unexpected or incompatible value for a property coming from the project root `gradle.properties` file.

The regression has been fixed in Gradle 6.2.2.

Deprecations

There were no deprecations between Gradle 6.1 and 6.2.

Upgrading from 6.0 and earlier

Deprecations

Querying a mapped output property of a task before the task has completed

Querying the value of a mapped output property before the task has completed can cause strange build failures because it indicates stale or non-existent outputs may be used by mistake. This behavior is deprecated and will emit a deprecation warning. This will become an error in Gradle 7.0.

The following example demonstrates this problem where the Producer's output file is parsed before the Producer executes:

```
class Consumer extends DefaultTask {
    @Input
    final Property<Integer> threadPoolSize = ...
}

class Producer extends DefaultTask {
    @OutputFile
    final RegularFileProperty outputFile = ...
}

// threadPoolSize is read from the producer's outputFile
consumer.threadPoolSize = producer.outputFile.map { it.text.toInteger() }

// Emits deprecation warning
println("thread pool size = " + consumer.threadPoolSize.get())
```

Querying the value of `consumer.threadPoolSize` will produce a deprecation warning if done prior to `producer` completing, as the output file has not yet been generated.

Discontinued methods

The following methods have been discontinued and should no longer be used. They will be removed in Gradle 7.0.

- `BasePluginConvention.setProject(ProjectInternal)`
- `BasePluginConvention.getProject()`
- `StartParameter.useEmptySettings()`
- `StartParameter.isUseEmptySettings()`

Alternative JVM plugins (a.k.a "Software Model")

A set of alternative plugins for Java and Scala development were introduced in Gradle 2.x as an experiment based on the "software model". These plugins are now deprecated and will eventually be removed. If you are still using one of these old plugins (`java-lang`, `scala-lang`, `jvm-component`, `jvm-resources`, `junit-test-suite`) please consult the documentation on Building Java & JVM projects to determine which of the stable JVM plugins are appropriate for your project.

Potential breaking changes

`ProjectLayout` is no longer available to worker actions as a service

In Gradle 6.0, the `ProjectLayout` service was made available to worker actions via service injection. This service allowed for mutable state to leak into a worker action and introduced a way for dependencies to go undeclared in the worker action.

`ProjectLayout` has been removed from the available services. Worker actions that were using `ProjectLayout` should switch to injecting the `projectDirectory` or `buildDirectory` as a parameter instead.

Updates to bundled Gradle dependencies

- Kotlin has been updated to Kotlin 1.3.61.

Updates to default tool integration versions

- Checkstyle has been updated to Checkstyle 8.27.
- PMD has been updated to PMD 6.20.0.

Publishing Spring Boot applications

Starting from Gradle 6.2, Gradle performs a sanity check before uploading, to make sure you don't upload stale files (files produced by another build). This introduces a problem with Spring Boot applications which are uploaded using the `components.java` component:

```
Artifact my-application-0.0.1-SNAPSHOT.jar wasn't produced by this build.
```

This is caused by the fact that the main `jar` task is disabled by the Spring Boot application, and the component expects it to be present. Because the `bootJar` task uses the *same file* as the main `jar` task by default, previous releases of Gradle would either:

- publish a stale `bootJar` artifact
- or fail if the `bootJar` task hasn't been called previously

A workaround is to tell Gradle what to upload. If you want to upload the `bootJar`, then you need to configure the outgoing configurations to do this:

```
configurations {
   [apiElements, runtimeElements].each {
       it.outgoing.artifacts.removeIf {
it.buildDependencies.getDependencies(null).contains(jar) }
       it.outgoing.artifact(bootJar)
   }
}
```

Alternatively, you might want to re-enable the `jar` task, and add the `bootJar` with a different classifier.

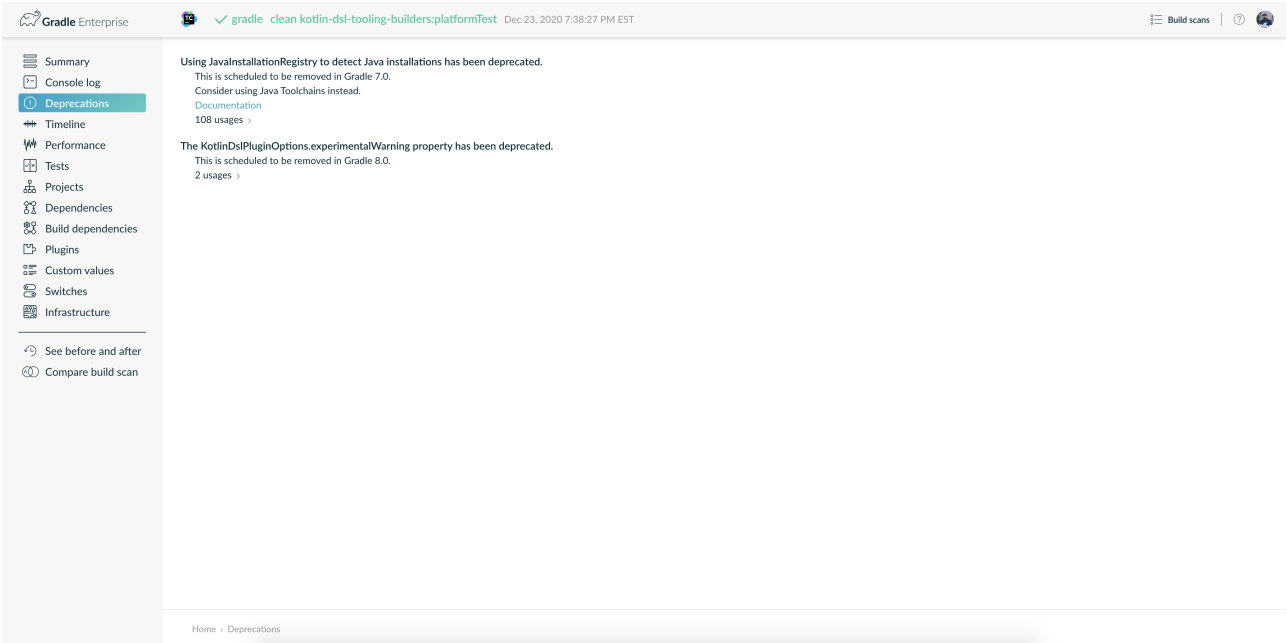
```
jar {
   enabled = true
}

bootJar {
   classifier = 'application'
}
```

Upgrading your build from Gradle 5.x to 6.0

This chapter provides the information you need to migrate your Gradle 5.x builds to Gradle 6.0. For migrating from Gradle 4.x, complete the 4.x to 5.0 guide first.

We recommend the following steps for all users:

1. Try running `gradle help --scan` and view the deprecations view of the generated build scan.



   This is so that you can see any deprecation warnings that apply to your build.

   Alternatively, you could run `gradle help --warning-mode=all` to see the deprecations in the

console, though it may not report as much detailed information.

2. Update your plugins.

   Some plugins will break with this new version of Gradle, for example because they use internal APIs that have been removed or changed. The previous step will help you identify potential problems by issuing deprecation warnings when a plugin does try to use a deprecated part of the API.

3. Run `gradle wrapper --gradle-version 6.0` to update the project to 6.0.
4. Try to run the project and debug any errors using the Troubleshooting Guide.

Upgrading from 5.6 and earlier

Deprecations

Dependencies should no longer be declared using the `compile` and `runtime` configurations

The usage of the `compile` and `runtime` configurations in the Java ecosystem plugins has been discouraged since Gradle 3.4.

These configurations are used for compiling and running code from the `main` source set. Other sources sets create similar configurations (e.g. `testCompile` and `testRuntime` for the `test` source set), should not be used either. The `implementation`, `api`, `compileOnly` and `runtimeOnly` configurations should be used to declare dependencies and the `compileClasspath` and `runtimeClasspath` configurations to resolve dependencies. See the relationship of these configurations.

Legacy publication system is deprecated and replaced with the `*-publish` plugins

The `uploadArchives` task and the `maven` plugin are deprecated.

Users should migrate to the publishing system of Gradle by using either the `maven-publish` or `ivy-publish` plugins. These plugins have been stable since Gradle 4.8.

The publishing system is also the only way to ensure the publication of Gradle Module Metadata.

Problems with tasks emit deprecation warnings

When Gradle detects problems with task definitions (such as incorrectly defined inputs or outputs) it will show the following message on the console:

```
Deprecated Gradle features were used in this build, making it incompatible with Gradle
7.0.
Use '--warning-mode all' to show the individual deprecation warnings.
See
https://docs.gradle.org/6.0/userguide/command_line_interface.html#sec:command_line_war
nings
```

The deprecation warnings show up in build scans for every build, regardless of the command-line switches used.

When the build is executed with `--warning-mode all`, the individual warnings will be shown:

```
> Task :myTask
Property 'inputDirectory' is declared without normalization specified. Properties of
cacheable work must declare their normalization via @PathSensitive, @Classpath or
@CompileClasspath. Defaulting to PathSensitivity.ABSOLUTE. This behaviour has been
deprecated and is scheduled to be removed in Gradle 7.0.
Property 'outputFile' is not annotated with an input or output annotation. This
behaviour has been deprecated and is scheduled to be removed in Gradle 7.0.
```

If you own the code of the tasks in question, you can fix them by following the suggestions. You can also use `--stacktrace` to see where in the code each warning originates from.

Otherwise, you'll need to report the problems to the maintainer of the relevant task or plugin.

Old API for incremental tasks, `IncrementalTaskInputs`, has been deprecated

In Gradle 5.4 we introduced a new API for implementing incremental tasks: InputChanges. The old API based on `IncrementalTaskInputs` has been deprecated.

Forced dependencies

Forcing dependency versions using `force = true` on a first-level dependency has been deprecated.

Force has both a semantic and ordering issue which can be avoided by using a strict version constraint.

Search upwards related APIs in `StartParameter` have been deprecated

In Gradle 5.0, we removed the `--no-search-upward` CLI parameter.

The related APIs in `StartParameter` (like `isSearchUpwards()`) are now deprecated.

APIs `BuildListener.buildStarted` and `Gradle.buildStarted` have been deprecated

These methods currently do not work as expected since the callbacks will never be called after the build has started.

The methods are being deprecated to avoid confusion.

Implicit duplicate strategy for `Copy` or archive tasks has been deprecated

Archive tasks `Tar` and `Zip` by default allow multiple entries for the same path to exist in the created archive. This can cause "grossly invalid zip files" that can trigger zip bomb detection.

To prevent this from happening accidentally, encountering duplicates while creating an archive now produces a deprecation message and will fail the build starting with Gradle 7.0.

`Copy` tasks also happily copy multiple sources with the same relative path to the destination directory. This behavior has also been deprecated.

If you want to allow duplicates, you can specify that explicitly:

```
task archive(type: Zip) {
    duplicatesStrategy = DuplicatesStrategy.INCLUDE // allow duplicates
    ...
}
```

Executing Gradle without a settings file has been deprecated

A Gradle build is defined by a `settings.gradle[.kts]` file in the current or parent directory. Without a settings file, a Gradle build is undefined and will emit a deprecation warning.

In Gradle 7.0, Gradle will only allow you to invoke the `init` task or diagnostic command line flags, such as `--version`, with undefined builds.

Calling `Project.afterEvaluate` on an evaluated project has been deprecated

Once a project is evaluated, Gradle ignores all configuration passed to `Project#afterEvaluate` and emits a deprecation warning. This scenario will become an error in Gradle 7.0.

Deprecated plugins

The following bundled plugins were never announced and will be removed in the next major release of Gradle:

- `org.gradle.coffeescript-base`
- `org.gradle.envjs`
- `org.gradle.javascript-base`
- `org.gradle.jshint`
- `org.gradle.rhino`

Some of these plugins may have replacements on the Plugin Portal.

Potential breaking changes

Android Gradle Plugin 3.3 and earlier is no longer supported

Gradle 6.0 supports Android Gradle Plugin versions 3.4 and later.

Build scan plugin 2.x is no longer supported

For Gradle 6, usage of the build scan plugin must be replaced with the Gradle Enterprise plugin. This also requires changing how the plugin is applied. Please see https://gradle.com/help/gradle-6-build-scan-plugin for more information.

Updates to bundled Gradle dependencies

- Groovy has been updated to Groovy 2.5.8.

- Kotlin has been updated to Kotlin 1.3.50.
- Ant has been updated to Ant 1.10.7.

Updates to default integration versions

- Checkstyle has been updated to Checkstyle 8.24.
- CodeNarc has been updated to CodeNarc 1.4.
- PMD has been updated to PMD 6.17.0.
- JaCoCo has been updated to 0.8.5. Contributed by Evgeny Mandrikov

Changes to build and task names in composite builds

Previously, Gradle used the name of the root project as the build name for an included build. Now, the name of the build's root directory is used and the root project name is not considered if different. A different name for the build can be specified if the build is being included via a settings file.

```
includeBuild("some-other-build") {
    name = "another-name"
}
```

The previous behavior was problematic as it caused different names to be used at different times during the build.

buildSrc is now reserved as a project and subproject build name

Previously, Gradle did not prevent using the name "buildSrc" for a subproject of a multi-project build or as the name of an included build. Now, this is not allowed. The name "buildSrc" is now reserved for the conventional buildSrc project that builds extra build logic.

Typical use of buildSrc is unaffected by this change. You will only be affected if your settings file specifies `include("buildSrc")` or `includeBuild("buildSrc")`.

Scala Zinc compiler

The Zinc compiler has been upgraded to version 1.3.0. Gradle no longer supports building for Scala 2.9.

The minimum Zinc compiler supported by Gradle is 1.2.0 and the maximum tested version is 1.3.0.

To make it easier to select the version of the Zinc compiler, you can now configure a `zincVersion` property:

```
scala {
    zincVersion = "1.2.1"
}
```

Please remove any explicit dependencies you've added to the `zinc` configuration and use this property instead. If you try to use the `com.typesafe.zinc:zinc` dependency, Gradle will switch to the new Zinc implementation.

Changes to Build Cache

Local build cache is always a directory cache

In the past, it was possible to use any build cache implementation as the `local` cache. This is no longer allowed as the local cache must always be a `DirectoryBuildCache`.

Calls to `BuildCacheConfiguration.local(Class)` with anything other than `DirectoryBuildCache` as the type will fail the build. Calling these methods with the `DirectoryBuildCache` type will produce a deprecation warning.

Use `getLocal()` and `local(Action)` instead.

Failing to pack or unpack cached results will now fail the build

In the past, when Gradle encountered a problem while packing the results of a cached task, Gradle would ignore the problem and continue running the build.

When encountering a corrupt cached artifact, Gradle would remove whatever was already unpacked and re-execute the task to make sure the build had a chance to succeed.

While this behavior was intended to make a build successful, this had the adverse effect of hiding problems and led to reduced cache performance.

In Gradle 6.0, both pack and unpack errors will cause the build to fail, so that these problems will be surfaced more easily.

buildSrc projects automatically use build cache configuration

Previously, in order to to use the build cache for the buildSrc build you needed to duplicate your build cache config in the buildSrc build. Now, it automatically uses the build cache configuration defined by the top level settings script.

Changes to Dependency Management

Gradle Module Metadata is always published

Officially introduced in Gradle 5.3, Gradle Module Metadata was created to solve many of the problems that have plagued dependency management for years, in particular, but not exclusively, in the Java ecosystem.

With Gradle 6.0, Gradle Module Metadata is enabled by default.

This means, if you are publishing libraries with Gradle and using the maven-publish or ivy-publish plugin, the Gradle Module Metadata file is always published **in addition** to traditional metadata.

The traditional metadata file will contain a marker so that Gradle knows that there is additional metadata to consume.

Gradle Module Metadata has stricter validation

The following rules are verified when publishing Gradle Module Metadata:

- Variant names must be unique,
- Each variant must have at least one attribute,
- Two variants cannot have the exact same attributes and capabilities,
- If there are dependencies, at least one, across all variants, must carry version information.

These are documented in the specification as well.

Maven or Ivy repositories are no longer queried for artifacts without metadata by default

If Gradle fails to locate the metadata file (`.pom` or `ivy.xml`) of a module in a repository defined in the `repositories { }` section, it now assumes that the module does not exist in that repository.

For dynamic versions, the `maven-metadata.xml` for the corresponding module needs to be present in a Maven repository.

Previously, Gradle would also look for a default artifact (`.jar`). This behavior often caused a large number of unnecessary requests when using multiple repositories that slowed builds down.

You can opt into the old behavior for selected repositories by adding the `artifact()` metadata source.

Changing the pom `packaging` property no longer changes the artifact extension

Previously, if the pom packaging was not *jar*, *ejb*, *bundle* or *maven-plugin*, the extension of the main artifact published to a Maven repository was changed during publishing to match the pom packaging.

This behavior led to broken Gradle Module Metadata and was difficult to understand due to handling of different packaging types.

Build authors can change the artifact name when the artifact is created to obtain the same result as before — e.g. by setting `jar.archiveExtension.set(pomPackaging)` explicitly.

An `ivy.xml` published for Java libraries contains more information

A number of fixes were made to produce more correct `ivy.xml` metadata in the `ivy-publish` plugin.

As a consequence, the internal structure of the `ivy.xml` file has changed. The `runtime` configuration now contains more information, which corresponds to the *runtimeElements* variant of a Java library. The `default` configuration should yield the same result as before.

In general, users are advised to migrate from `ivy.xml` to the new Gradle Module Metadata format.

Changes to Plugins and Build scripts

Classes from `buildSrc` are no longer visible to settings scripts

Previously, the buildSrc project was built before applying the project's settings script and its classes were visible within the script. Now, buildSrc is built after the settings script and its classes are not visible to it. The buildSrc classes remain visible to project build scripts and script plugins.

Custom logic can be used from a settings script by declaring external dependencies.

The `pluginManagement` block in settings scripts is now isolated

Previously, any `pluginManagement {}` blocks inside a settings script were executed during the normal execution of the script.

Now, they are executed earlier in a similar manner to `buildscript {}` or `plugins {}`. This means that code inside such a block cannot reference anything declared elsewhere in the script.

This change has been made so that `pluginManagement` configuration can also be applied when resolving plugins for the settings script itself.

Plugins and classes loaded in settings scripts are visible to project scripts and `buildSrc`

Previously, any classes added to the a settings script by using `buildscript {}` were not visible outside of the script. Now, they they are visible to all of the project build scripts.

They are also visible to the `buildSrc` build script and its settings script.

This change has been made so that plugins applied to the settings script can contribute logic to the entire build.

Plugin validation changes

- The `validateTaskProperties` task is now deprecated, use `validatePlugins` instead. The new name better reflects the fact that it also validates artifact transform parameters and other non-property definitions.
- The `ValidateTaskProperties` type is replaced by `ValidatePlugins`.
- The `setClasses()` method is now removed. Use `getClasses().setFrom()` instead.
- The `setClasspath()` method is also removed. use `getClasspath().setFrom()` instead.
- The failOnWarning option is now enabled by default.
- The following task validation errors now fail the build at runtime and are promoted to errors for ValidatePlugins:
  - A task property is annotated with a property annotation not allowed for tasks, like `@InputArtifact`.

Changes to Kotlin DSL

Using the `embedded-kotlin` plugin now requires a repository

Just like when using the `kotlin-dsl` plugin, it is now required to declare a repository where Kotlin dependencies can be found if you apply the `embedded-kotlin` plugin.

```
plugins {
    `embedded-kotlin`
}

repositories {
    mavenCentral()
}
```

Kotlin DSL IDE support now requires Kotlin IntelliJ Plugin >= 1.3.50

With Kotlin IntelliJ plugin versions prior to 1.3.50, Kotlin DSL scripts will be wrongly highlighted when the *Gradle JVM* is set to a version different from the one in *Project SDK*. Simply upgrade your IDE plugin to a version >= 1.3.50 to restore the correct Kotlin DSL script highlighting behavior.

Kotlin DSL script base types no longer extend `Project`, `Settings` or `Gradle`

In previous versions, Kotlin DSL scripts were compiled to classes that implemented one of the three core Gradle configuration interfaces in order to implicitly expose their APIs to scripts. `org.gradle.api.Project` for project scripts, `org.gradle.api.initialization.Settings` for settings scripts and `org.gradle.api.invocation.Gradle` for init scripts.

Having the script instance implement the core Gradle interface of the model object it was supposed to configure was convenient because it made the model object API immediately available to the body of the script but it was also a lie that could cause all sorts of trouble whenever the script itself was used in place of the model object, a project script **was not** a proper `Project` instance just because it implemented the core `Project` interface and the same was true for settings and init scripts.

In 6.0 all Kotlin DSL scripts are compiled to classes that implement the newly introduced `org.gradle.kotlin.dsl.KotlinScript` interface and the corresponding model objects are now available as *implicit receivers* in the body of the scripts. In other words, a project script behaves as if the body of the script is enclosed within a `with(project) { ⋯ }` block, a settings script as if the body of the script is enclosed within a `with(settings) { ⋯ }` block and an init script as if the body of the script is enclosed within a `with(gradle) { ⋯ }` block. This implies the corresponding model object is also available as a property in the body of the script, the `project` property for project scripts, the `settings` property for settings scripts and the `gradle` property for init scripts.

As part of the change, the `SettingsScriptApi` interface is no longer implemented by settings scripts and the `InitScriptApi` interface is no longer implemented by init scripts. They should be replaced with the corresponding model object interfaces, `Settings` and `Gradle`.

Miscellaneous

Javadoc and Groovydoc don't include timestamps by default

Timestamps in the generated documentation have very limited practical use, however they make it impossible to have repeatable documentation builds. Therefore, the `Javadoc` and `Groovydoc` tasks are now configured to not include timestamps by default any more.

User provided 'config_loc' properties are ignored by Checkstyle

Gradle always uses `configDirectory` as the value for 'config_loc' when running Checkstyle.

New Tooling API progress event

In Gradle 6.0, we introduced a new progress event (org.gradle.tooling.events.test.TestOutputEvent) to expose the output of test execution. This new event breaks the convention of having a `StartEvent`-`FinisEvent` pair to express progress. `TaskOutputEvent` is a simple `ProgressEvent`.

Changes to the task container behavior

The following deprecated methods on the task container now result in errors:

- `TaskContainer.add()`
- `TaskContainer.addAll()`
- `TaskContainer.remove()`
- `TaskContainer.removeAll()`
- `TaskContainer.retainAll()`
- `TaskContainer.clear()`
- `TaskContainer.iterator().remove()`

Additionally, the following deprecated functionality now results in an error:

- Replacing a task that has already been realized.
- Replacing a registered (unrealized) task with an incompatible type. A compatible type is the same type or a sub-type of the registered type.
- Replacing a task that has never been registered.

Replaced and Removed APIs

Methods on `DefaultTask` and `ProjectLayout` replaced with `ObjectFactory`

Use `ObjectFactory.fileProperty()` instead of the following methods that are now removed:

- `DefaultTask.newInputFile()`
- `DefaultTask.newOutputFile()`
- `ProjectLayout.fileProperty()`

Use `ObjectFactory.directoryProperty()` instead of the following methods that are now removed:

- `DefaultTask.newInputDirectory()`
- `DefaultTask.newOutputDirectory()`
- `ProjectLayout.directoryProperty()`

Annotation `@Nullable` has been removed

The `org.gradle.api.Nullable` annotation type has been removed. Use `javax.annotation.Nullable` from JSR-305 instead.

The FindBugs plugin has been removed

The deprecated FindBugs plugin has been removed. As an alternative, you can use the SpotBugs plugin from the Gradle Plugin Portal.

The JDepend plugin has been removed

The deprecated JDepend plugin has been removed. There are a number of community-provided plugins for code and architecture analysis available on the Gradle Plugin Portal.

The OSGI plugin has been removed

The deprecated OSGI plugin has been removed. There are a number of community-provided OSGI plugins available on the Gradle Plugin Portal.

The announce and build-announcements plugins have been removed

The deprecated announce and build-announcements plugins have been removed. There are a number of community-provided plugins for sending out notifications available on the Gradle Plugin Portal.

The Compare Gradle Builds plugin has been removed

The deprecated Compare Gradle Builds plugin has been removed. Please use build scans for build analysis and comparison.

The Play plugins have been removed

The deprecated Play plugin has been removed. An external replacement, the Play Framework plugin, is available from the plugin portal.

Method `AbstractCompile.compile()` method has been removed

The abstract method `compile()` is no longer declared by `AbstractCompile`.

Tasks extending `AbstractCompile` can implement their own `@TaskAction` method with the name of their choosing.

They are also free to add a method annotated with `@TaskAction` using an `InputChanges` parameter without having to implement a parameter-less one as well.

Other Deprecated Behaviors and APIs

- The `org.gradle.util.internal.GUtil.savePropertiesNoDateComment` has been removed. There is no public replacement for this internal method.
- The deprecated class `org.gradle.api.tasks.compile.CompilerArgumentProvider` has been removed. Use org.gradle.process.CommandLineArgumentProvider instead.

- The deprecated class `org.gradle.api.ConventionProperty` has been removed. Use Providers instead of convention properties.
- The deprecated class `org.gradle.reporting.DurationFormatter` has been removed.
- The bridge method `org.gradle.api.tasks.TaskInputs.property(String name, @Nullable Object value)` returning `TaskInputs` has been removed. A plugin using the method must be compiled with Gradle 4.3 to work on Gradle 6.0.
- The following setters have been removed from `JacocoReportBase`:
  - executionData - use `getExecutionData().setFrom()` instead.
  - sourceDirectories - use `getSourceDirectories().setFrom()` instead.
  - classDirectories - use `getClassDirectories().setFrom()` instead.
  - additionalClassDirs - use `getAdditionalClassDirs().setFrom()` instead.
  - additionalSourceDirs - use `getAdditionalSourceDirs().setFrom()` instead.
- The `append` property on `JacocoTaskExtension` has been removed. `append` is now always configured to be true for the Jacoco agent.
- The `configureDefaultOutputPathForJacocoMerge` method on `JacocoPlugin` has been removed. The method was never meant to be public.
- File paths in deployment descriptor file name for the ear plugin are not allowed any more. Use a simple name, like `application.xml`, instead.
- The `org.gradle.testfixtures.ProjectBuilder` constructor has been removed. Please use `ProjectBuilder.builder()` instead.
- When incremental Groovy compilation is enabled, a wrong configuration of the source roots or enabling Java annotation for Groovy now fails the build. Disable incremental Groovy compilation when you want to compile in those cases.
- `ComponentSelectionRule` no longer can inject the metadata or ivy descriptor. Use the methods on the `ComponentSelection` parameter instead.
- Declaring an incremental task without declaring outputs is now an error. Declare file outputs or use TaskOutputs.upToDateWhen() instead.
- The `getEffectiveAnnotationProcessorPath()` method is removed from the `JavaCompile` and `ScalaCompile` tasks.
- Changing the value of a task property with type `Property<T>` after the task has started execution now results in an error.
- The `isLegacyLayout()` method is removed from `SourceSetOutput`.
- The map returned by `TaskInputs.getProperties()` is now unmodifiable. Trying to modify it will result in an `UnsupportedOperationException` being thrown.
- There are slight changes in the incubating capabilities resolution API, which has been introduced in 5.6, to also allow variant selection based on variant name

Upgrading from 5.5 and earlier

Deprecations

Changing the contents of `ConfigurableFileCollection` task properties after task starts execution

When a task property has type `ConfigurableFileCollection`, then the file collection referenced by the property will ignore changes made to the contents of the collection once the task starts execution. This has two benefits. Firstly, this prevents accidental changes to the property value during task execution which can cause Gradle up-to-date checks and build cache lookup using different values to those used by the task action. Secondly, this improves performance as Gradle can calculate the value once and cache the result.

This will become an error in Gradle 6.0.

Creating `SignOperation` instances

Creating `SignOperation` instances directly is now deprecated. Instead, the methods of `SigningExtension` should be used to create these instances.

This will become an error in Gradle 6.0.

Declaring an incremental task without outputs

Declaring an incremental task without declaring outputs is now deprecated. Declare file outputs or use TaskOutputs.upToDateWhen() instead.

This will become an error in Gradle 6.0.

Method `WorkerExecutor.submit()` is deprecated

The `WorkerExecutor.submit()` method is now deprecated. The new `noIsolation()`, `classLoaderIsolation()` and `processIsolation()` methods should now be used to submit work. See the section on the Worker API for more information on using these methods.

`WorkerExecutor.submit()` will be removed in Gradle 8.0.

Potential breaking changes

Task dependencies are honored for task `@Input` properties whose value is a `Property`

Previously, task dependencies would be ignored for task `@Input` properties of type `Property<T>`. These are now honored, so that it is possible to attach a task output property to a task `@Input` property.

This may introduce unexpected cycles in the task dependency graph, where the value of an output property is mapped to produce a value for an input property.

Declaring task dependencies using a file `Provider` that does not represent a task output

Previously, it was possible to pass `Task.dependsOn()` a `Provider<File>`, `Provider<RegularFile>` or `Provider<Directory>` instance that did not represent a task output. These providers would be silently ignored.

This is now an error because Gradle does not know how to build files that are not task outputs.

Note that it is still possible to to pass `Task.dependsOn()` a `Provider` that returns a file and that represents a task output, for example `myTask.dependsOn(jar.archiveFile)` or `myTask.dependsOn(taskProvider.flatMap { it.outputDirectory })`, when the `Provider` is an annotated `@OutputFile` or `@OutputDirectory` property of a task.

Setting `Property` value to `null` uses the property convention

Previously, calling `Property.set(null)` would always reset the value of the property to 'not defined'. Now, the convention that is associated with the property using the `convention()` method will be used to determine the value of the property.

Enhanced validation of names for `publishing.publications` and `publishing.repositories`

The repository and publication names are used to construct task names for publishing. It was possible to supply a name that would result in an invalid task name. Names for publications and repositories are now restricted to `[A-Za-z0-9_\\-.]+`.

Restricted Worker API classloader and process classpath

Gradle now prevents internal dependencies (like Guava) from leaking into the classpath used by Worker API actions. This fixes an issue where a worker needs to use a dependency that is also used by Gradle internally.

In previous releases, it was possible to rely on these leaked classes. Plugins relying on this behavior will now fail. To fix the plugin, the worker should explicitly include all required dependencies in its classpath.

Default PMD version upgraded to 6.15.0

The PMD plugin has been upgraded to use PMD version 6.15.0 instead of 6.8.0 by default.

Contributed by wreulicke

Configuration copies have unique names

Previously, all copies of a configuration always had the name `<OriginConfigurationName>Copy`. Now when creating multiple copies, each will have a unique name by adding an index starting from the second copy. (e.g. `CompileOnlyCopy2`)

Changed classpath filtering for Eclipse

Gradle 5.6 no longer supplies custom classpath attributes in the Eclipse model. Instead, it provides the attributes for Eclipse test sources. This change requires Buildship version 3.1.1 or later.

Embedded Kotlin upgraded to 1.3.41

Gradle Kotlin DSL scripts and Gradle Plugins authored using the `kotlin-dsl` plugin are now compiled using Kotlin 1.3.41.

Please see the Kotlin blog post and changelog for more information about the included changes.

The minimum supported Kotlin Gradle Plugin version is now 1.2.31. Previously it was 1.2.21.

Automatic capability conflict resolution

Previous versions of Gradle would automatically select, in case of capability conflicts, the module which has the highest capability version. Starting from 5.6, this is an opt-in behavior that can be activated using:

```
configurations.all {
   resolutionStrategy.capabilitiesResolution.all { selectHighestVersion() }
}
```

See the capabilities section of the documentation for more options.

File removal operations don't follow symlinked directories

When Gradle has to remove the output files of a task for various reasons, it will not follow symlinked directories. The symlink itself will be deleted, but the contents of the linked directory will stay intact.

Disabled debug argument parsing in JavaExec

Gradle 5.6 introduced a new DSL element (`JavaForkOptions.debugOptions(Action<JavaDebugOptions>)`) to configure debug properties for forked Java processes. Due to this change, Gradle no longer parses debug-related JVM arguments. Consequently, `JavaForkOptions.getDebu()` no longer returns `true` if the `-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005` or the `-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005` argument is specified to the process.

Scala 2.9 and Zinc compiler

Gradle no longer supports building applications using Scala 2.9.

Upgrading from 5.4 and earlier

Deprecations

Play

The built-in Play plugin has been deprecated and will be replaced by a new Play Framework plugin available from the plugin portal.

Build Comparison

The *build comparison* plugin has been deprecated and will be removed in the next major version of Gradle.

Build scans show much deeper insights into your build and you can use Gradle Enterprise to directly compare two build's build-scans.

Potential breaking changes

User supplied Eclipse project names may be ignored on conflict

Project names configured via `EclipseProject.setName(…)` were honored by Gradle and Buildship in all cases, even when the names caused conflicts and import/synchronization errors.

Gradle can now deduplicate these names if they conflict with other project names in an Eclipse workspace. This may lead to different Eclipse project names for projects with user-specified names.

The upcoming 3.1.1 version of Buildship is required to take advantage of this behavior.

Contributed by Christian Fränkel

Default JaCoCo version upgraded to 0.8.4

The JaCoCo plugin has been upgraded to use JaCoCo version 0.8.4 instead of 0.8.3 by default.

Contributed by Evgeny Mandrikov

Embedded Ant version upgraded to 1.9.14

The version of Ant distributed with Gradle has been upgraded to 1.9.14 from 1.9.13.

Type `DependencyHandler` now statically exposes `ExtensionAware`

This affects Kotlin DSL build scripts that make use of `ExtensionAware` extension members such as the `extra` properties accessor inside the `dependencies {}` block. The receiver for those members will no longer be the enclosing `Project` instance but the `dependencies` object itself, the innermost `ExtensionAware` conforming receiver. In order to address `Project` extra properties inside `dependencies {}` the receiver must be explicitly qualified i.e. `project.extra` instead of just `extra`. Affected extensions also include `the<T>()` and `configure<T>(T.() → Unit)`.

Improved processing of dependency excludes

Previous versions of Gradle could, in some complex dependency graphs, have a wrong result or a randomized dependency order when lots of excludes were present. To mitigate this, the algorithm that computes exclusions has been rewritten. In some rare cases this may cause some differences in resolution, due to the correctness changes.

Improved classpath separation for worker processes

The system classpath for worker daemons started by the Worker API when using `PROCESS` isolation has been reduced to a minimum set of Gradle infrastructure. User code is still segregated into a separate classloader to isolate it from the Gradle runtime. This should be a transparent change for tasks using the worker API, but previous versions of Gradle mixed user code and Gradle internals in the worker process. Worker actions that rely on things like the `java.class.path` system property may be affected, since `java.class.path` now represents only the classpath of the Gradle internals.

Upgrading from 5.3 and earlier

Deprecations

Using custom local build cache implementations

Using a custom build cache implementation for the local build cache is now deprecated. The only allowed type will be `DirectoryBuildCache` going forward. There is no change in the support for using custom build cache implementations as the remote build cache.

Potential breaking changes

Use HTTPS when configuring Google Hosted Libraries via `googleApis()`

The Google Hosted Libraries URL accessible via `JavaScriptRepositoriesExtension#GOOGLE_APIS_REPO_URL` was changed to use the HTTPS protocol. The change also affect the Ivy repository configured via `googleApis()`.

Upgrading from 5.2 and earlier

Potential breaking changes

Bug fixes in platform resolution

There was a bug from Gradle 5.0 to 5.2.1 (included) where enforced platforms would potentially include dependencies instead of constraints. This would happen whenever a POM file defined both dependencies and "constraints" (via `<dependencyManagement>`) and that you used `enforcedPlatform`. Gradle 5.3 fixes this bug, meaning that you might have differences in the resolution result if you relied on this broken behavior. Similarly, Gradle 5.3 will no longer try to download jars for `platform` and `enforcedPlatform` dependencies (as they should only bring in constraints).

Automatic target JVM version

If you apply any of the Java plugins, Gradle will now do its best to select dependencies which match the target compatibility of the module being compiled. What it means, in practice, is that if you have module A built for Java 8, and module B built for Java 8, then there's no change. However if B is built for Java 9+, then it's not binary compatible anymore, and Gradle would complain with an error message like the following:

```
Unable to find a matching variant of project :producer:
  - Variant 'apiElements' capability test:producer:unspecified:
      - Provides org.gradle.dependency.bundling 'external'
      - Required org.gradle.jvm.version '8' and found incompatible value '9'.
      - Required org.gradle.usage 'java-api' and found value 'java-api-jars'.
  - Variant 'runtimeElements' capability test:producer:unspecified:
      - Provides org.gradle.dependency.bundling 'external'
      - Required org.gradle.jvm.version '8' and found incompatible value '9'.
      - Required org.gradle.usage 'java-api' and found value 'java-runtime-jars'.
```

In general, this is a sign that your project is misconfigured and that your dependencies are not compatible. However, there are cases where you still may want to do this, for example when only a *subset* of classes of your module actually need the Java 9 dependencies, and are not intended to be used on earlier releases. Java in general doesn't encourage you to do this (you should split your module instead), but if you face this problem, you can workaround by disabling this new behavior on the consumer side:

```
java {
   disableAutoTargetJvm()
}
```

Bug fix in Maven / Ivy interoperability with dependency substitution

If you have a Maven dependency pointing to an Ivy dependency where the `default` configuration dependencies do not match the `compile` + `runtime` + `master` ones *and* that Ivy dependency was substituted (using a `resolutionStrategy.force`, `resolutionStrategy.eachDependency` or `resolutionStrategy.dependencySubstitution`) then this fix will impact you. The legacy behaviour of Gradle, prior to 5.0, was still in place instead of being replaced by the changes introduced by improved pom support.

Delete operations correctly handle symbolic links on Windows

Gradle no longer ignores the `followSymlink` option on Windows for the `clean` task, all `Delete` tasks, and `project.delete {}` operations in the presence of junction points and symbolic links.

Fix in publication of additional artifacts

In previous Gradle versions, additional artifacts registered at the project level were not published by `maven-publish` or `ivy-publish` unless they were also added as artifacts in the publication configuration.

With Gradle 5.3, these artifacts are now properly accounted for and published.

This means that artifacts that are registered both on the project *and* the publication, Ivy or Maven, will cause publication to fail since it will create duplicate entries. The fix is to remove these artifacts from the publication configuration.

Upgrading from 5.1 and earlier

Potential breaking changes

none

Upgrading from 5.0 and earlier

Deprecations

Follow the API links to learn how to deal with these deprecations (if no extra information is provided here):

- Setters for `classes` and `classpath` on `ValidateTaskProperties`
- There should not be setters for lazy properties like `ConfigurableFileCollection`. Use `setFrom` instead. For example,

```
validateTaskProperties.getClasses().setFrom(fileCollection)
validateTaskProperties.getClasspath().setFrom(fileCollection)
```

Potential breaking changes

The following changes were not previously deprecated:

Signing API changes

Input and output files of `Sign` tasks are now tracked via `Signature.getToSign()` and `Signature.getFile()`, respectively.

Collection properties default to empty collection

In Gradle 5.0, the collection property instances created using `ObjectFactory` would have no value defined, requiring plugin authors to explicitly set an initial value. This proved to be awkward and error prone so `ObjectFactory` now returns instances with an empty collection as their initial value.

Worker API: working directory of a worker can no longer be set

Since JDK 11 no longer supports changing the working directory of a running process, setting the working directory of a worker via its fork options is now prohibited. All workers now use the same working directory to enable reuse. Please pass files and directories as arguments instead. See examples in the Worker API documentation.

Changes to native linking tasks

To expand our idiomatic Provider API practices, the install name property from `org.gradle.nativeplatform.tasks.LinkSharedLibrary` is affected by this change.

- `getInstallName()` was changed to return a `Property`.
- `setInstallName(String)` was removed. Use `Property.set()` instead.

Passing arguments to Windows Resource Compiler

To expand our idiomatic Provider API practices, the `WindowsResourceCompile` task has been converted to use the Provider API.

Passing additional compiler arguments now follow the same pattern as the `CppCompile` and other tasks.

Copied configuration no longer shares a list of `beforeResolve` actions with original

The list of `beforeResolve` actions are no longer shared between a copied configuration and the original. Instead, a copied configuration receives a copy of the `beforeResolve` actions at the time the

copy is made. Any `beforeResolve` actions added after copying (to either configuration) will not be shared between the original and the copy. This may break plugins that relied on the previous behaviour.

Changes to incubating POM customization types

- The type of `MavenPomDeveloper.properties` has changed from `Property<Map<String, String>>` to `MapProperty<String, String>`.
- The type of `MavenPomContributor.properties` has changed from `Property<Map<String, String>>` to `MapProperty<String, String>`.

Changes to specifying operating system for native projects

The incubating `operatingSystems` property on native components has been replaced with the targetMachines property.

Changes for archive tasks (`Zip`, `Jar`, `War`, `Ear`, `Tar`)

Change in behavior for tasks extending `AbstractArchiveTask`

The `AbstractArchiveTask` has several new properties using the Provider API. Plugins that extend these types and override methods from the base class may no longer behave the same way. Internally, `AbstractArchiveTask` prefers the new properties and methods like `getArchiveName()` are façades over the new properties.

If your plugin/build only uses these types (and does not extend them), nothing has changed.

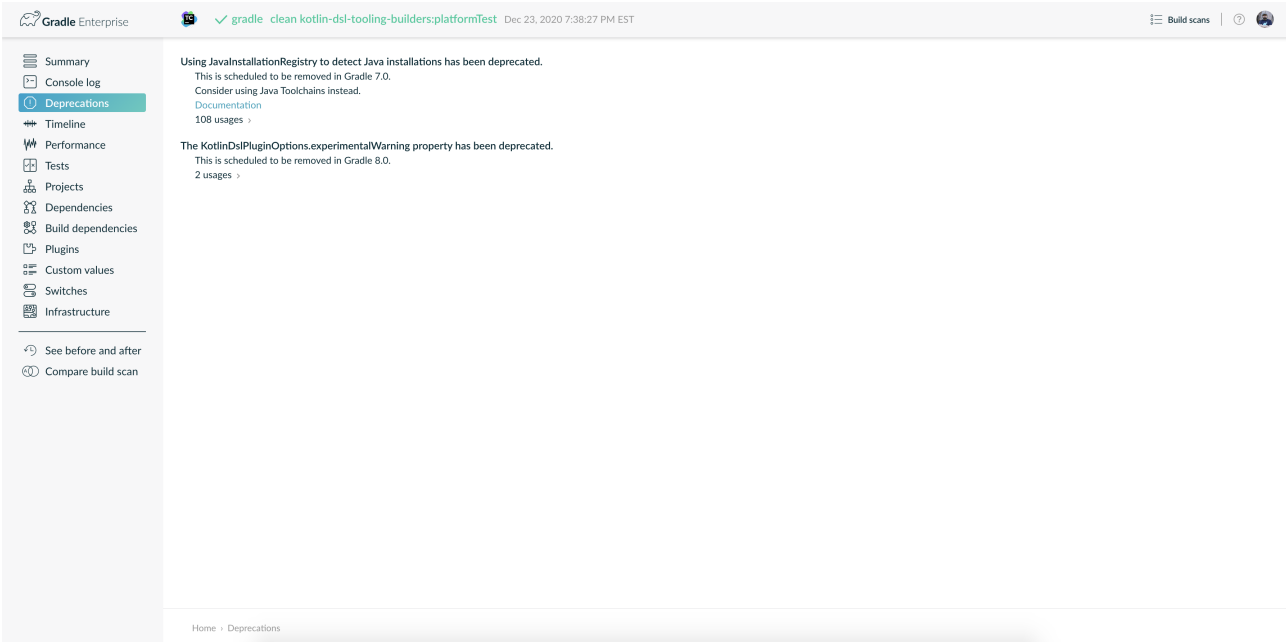
Upgrading your build from Gradle 4.x to 5.0

This chapter provides the information you need to migrate your older Gradle 4.x builds to Gradle 5.0. In most cases, you will need to apply the changes from all versions that come after the one you're upgrading from. For example, if you're upgrading from Gradle 4.3 to 5.0, you will also need to apply the changes since 4.4, 4.5, etc up to 5.0.

TIP	If you are using Gradle for Android, you need to move to version 3.3 or higher of both the Android Gradle Plugin and Android Studio.

For all users

1. If you are not already on the latest 4.10.x release, read the sections below for help upgrading your project to the latest 4.10.x release. We recommend upgrading to the latest 4.10.x release to get the most useful warnings and deprecations information before moving to 5.0. Avoid upgrading Gradle and migrating to Kotlin DSL at the same time in order to ease troubleshooting in case of potential issues.
2. Try running `gradle help --scan` and view the deprecations view of the generated build scan. If there are no warnings, the Deprecations tab will not appear.



This is so that you can see any deprecation warnings that apply to your build. Gradle 5.x will generate (potentially less obvious) errors if you try to upgrade directly to it.

Alternatively, you could run `gradle help --warning-mode=all` to see the deprecations in the console, though it may not report as much detailed information.

3. Update your plugins.

   Some plugins will break with this new version of Gradle, for example because they use internal APIs that have been removed or changed. The previous step will help you identify potential problems by issuing deprecation warnings when a plugin does try to use a deprecated part of the API.

   In particular, you will need to use at least a 2.x version of the **Shadow Plugin**.

4. Run `gradle wrapper --gradle-version 5.0` to update the project to 5.0
5. Move to Java 8 or higher if you haven't already. Whereas Gradle 4.x requires Java 7, Gradle 5 requires Java 8 to run.
6. Read the Upgrading from 4.10 section and make any necessary changes.
7. Try to run the project and debug any errors using the Troubleshooting Guide.

In addition, Gradle has added several significant new and improved features that you should consider using in your builds:

- Maven Publish and Ivy Publish Plugins that now support digital signatures with the Signing Plugin.
- Use native BOM import in your builds.
- The Worker API for enabling units of work to run in parallel.
- A new API for creating and configuring tasks lazily that can significantly improve your build's configuration time.

Other notable changes to be aware of that may break your build include:

- Separation of compile and runtime dependencies when consuming POMs
- A change that means you should configure existing `wrapper` and `init` tasks rather than defining your own.
- The honoring of implicit wildcards in Maven POM exclusions, which may result in dependencies being excluded that weren't before.
- A change to the way you add Java annotation processors to a project.
- The default memory settings for the command-line client, the Gradle daemon, and all workers including compilers and test executors, have been greatly reduced.
- The default versions of several code quality plugins have been updated.
- Several library versions used by Gradle have been upgraded.

Upgrading from 4.10 and earlier

If you are not already on version 4.10, skip down to the section that applies to your current Gradle version and work your way up until you reach here. Then, apply these changes when moving from Gradle 4.10 to 5.0.

Other changes

- The `enableFeaturePreview('IMPROVED_POM_SUPPORT')` and `enableFeaturePreview('STABLE_PUBLISHING')` flags are no longer necessary. These features are now enabled by default.
- Gradle now bundles JAXB for Java 9 and above. You can remove the `--add-modules java.xml.bind` option from `org.gradle.jvmargs`, if set.

Potential breaking changes

The changes in this section have the potential to break your build, but the vast majority have been deprecated for quite some time and few builds will be affected by a large number of them. We strongly recommend upgrading to Gradle 4.10 first to get a report on what deprecations affect your build.

The following breaking changes are not from deprecations, but the result of changes in behavior:

- Separation of compile and runtime dependencies when consuming POMs
- The evaluation of the `publishing {}` block is no longer deferred until needed but behaves like any other block. Please use `afterEvaluate {}` if you need to defer evaluation.
- The `Javadoc` and `Groovydoc` tasks now delete the destination dir for the documentation before executing. This has been added to remove stale output files from the last task execution.
- The Java Library Distribution Plugin is now based on the Java Library Plugin instead of the Java Plugin.

  While it applies the Java Plugin, it behaves slightly different (e.g. it adds the `api` configuration). Thus, make sure to check whether your build behaves as expected after upgrading.

- The `html` property on `CheckstyleReport` and `FindBugsReport` now returns a `CustomizableHtmlReport` instance that is easier to configure from statically typed languages like Java and Kotlin.
- The Configuration Avoidance API has been updated to prevent the creation and configuration of tasks that are never used.
- The default memory settings for the command-line client, the Gradle daemon, and all workers including compilers and test executors, have been greatly reduced.
- The default versions of several code quality plugins have been updated.
- Several library versions used by Gradle have been upgraded.

The following breaking changes will appear as deprecation warnings with Gradle 4.10:

General

- `<<` for task definitions no longer works. In other words, you can not use the syntax `task myTask << { ⋯ }`.

  Use the Task.doLast() method instead, like this:

  ```
  task myTask {
      doLast {
          ...
      }
  }
  ```

- You can no longer use any of the following characters in domain object names, such as project and task names: <space> `/ \ : < > " ? * |` . You should also not use `.` as a leading or trailing character.

Running Gradle & build environment

- As mentioned before, Gradle can no longer be run on Java 7. However, you can still use forked compilation and testing to build and test software for Java 6 and above.
- The `-Dtest.single` command-line option has been removed — use test filtering instead.
- The `-Dtest.debug` command-line option has been removed — use the `--debug-jvm` option instead.
- The `-u`/`--no-search-upward` command-line option has been removed — make sure all your builds have a *settings.gradle* file.
- The `--recompile-scripts` command-line option has been removed.
- You can no longer have a Gradle build nested in a subdirectory of another Gradle build unless the nested build has a *settings.gradle* file.
- The `DirectoryBuildCache.setTargetSizeInMB(long)` method has been removed — use DirectoryBuildCache.removeUnusedEntriesAfterDays instead.
- The `org.gradle.readLoggingConfigFile` system property no longer does anything — update affected tests to work with your `java.util.logging` settings.

Working with files

- You can no longer cast `FileCollection` objects to other types using the `as` keyword or the `asType()` method.
- You can no longer pass `null` as the configuration action of CopySpec.from(Object, Action).
- For better compatibility with the Kotlin DSL, CopySpec.duplicatesStrategy is no longer nullable. The property setter no longer accepts `null` as a way to reset the property back to its default value. Use `DuplicatesStrategy.INHERIT` instead.
- The `FileCollection.stopExecutionIfEmpty()` method has been removed — use the @SkipWhenEmpty annotation on `FileCollection` task properties instead.
- The `FileCollection.add()` method has been removed — use Project.files() and Project.fileTree() to create configurable file collections/file trees and add to them via ConfigurableFileCollection.from().
- `SimpleFileCollection` has been removed — use Project.files(Object...) instead.
- Don't have your own classes extend `AbstractFileCollection` — use the Project.files() method instead. This problem may exhibit as a missing `getBuildDependencies()` method.

Java builds

- The `CompileOptions.bootClasspath` property has been removed — use CompileOptions.bootstrapClasspath instead.
- You can no longer use `-source-path` as a generic compiler argument — use CompileOptions.sourcepath instead.
- You can no longer use `-processorpath` as a generic compiler argument — use CompileOptions.annotationProcessorPath instead.
- Gradle will no longer automatically apply annotation processors that are on the compile classpath — use CompileOptions.annotationProcessorPath instead.
- The `testClassesDir` property has been removed from the Test task — use testClassesDirs instead.
- The `classesDir` property has been removed from both the *JDepend* task and SourceSetOutput. Use the *JDepend.classesDirs* and SourceSetOutput.classesDirs properties instead.
- The `JavaLibrary(PublishArtifact, DependencySet)` constructor has been removed — this was used by the Shadow Plugin, so make sure you upgrade to at least version 2.x of that plugin.
- The `JavaBasePlugin.configureForSourceSet()` method has been removed.
- You can no longer create your own instances of JavaPluginConvention, ApplicationPluginConvention, WarPluginConvention, EarPluginConvention, BasePluginConvention, and ProjectReportsPluginConvention.
- The `Maven` Plugin used to publish the highly outdated Maven 2 metadata format. This has been changed and it will now publish Maven 3 metadata, just like the `Maven Publish` Plugin.

  With the removal of Maven 2 support, the methods that configure unique snapshot behavior have also been removed. Maven 3 only supports unique snapshots, so we decided to remove them.

Tasks & properties

- The following legacy classes and methods related to lazy properties have been removed — use ObjectFactory.property() to create `Property` instances:
  - `PropertyState`
  - `DirectoryVar`
  - `RegularFileVar`
  - `ProjectLayout.newDirectoryVar()`
  - `ProjectLayout.newFileVar()`
  - `Project.property(Class)`
  - `Script.property(Class)`
  - `ProviderFactory.property(Class)`
- Tasks configured and registered with the task configuration avoidance APIs have more restrictions on the other methods that can be called from a configuration action.
- The internal `@Option` and `@OptionValues` annotations — package `org.gradle.api.internal.tasks.options` — have been removed. Use the public @Option and @OptionValues annotations instead.
- The `Task.deleteAllActions()` method has been removed with no replacement.
- The `Task.dependsOnTaskDidWork()` method has been removed — use declared inputs and outputs instead.
- The following properties and methods of `TaskInternal` have been removed — use task dependencies, task rules, reusable utility methods, or the Worker API in place of executing a task directly.
  - `execute()`
  - `executer`
  - `getValidators()`
  - `addValidator()`
- The TaskInputs.file(Object) method can no longer be called with an argument that resolves to anything other than a single regular file.
- The TaskInputs.dir(Object) method can no longer be called with an argument that resolves to anything other than a single directory.
- You can no longer register invalid inputs and outputs via TaskInputs and TaskOutputs.
- The `TaskDestroyables.file()` and `TaskDestroyables.files()` methods have been removed — use TaskDestroyables.register() instead.
- `SimpleWorkResult` has been removed — use WorkResult.didWork.
- Overriding built-in tasks deprecated in 4.8 now produces an error.

  Attempting to replace a built-in task will produce an error similar to the following:

```
> Cannot add task 'wrapper' as a task with that name already exists.
```

Scala & Play

- Play 2.2 is no longer supported — please upgrade the version of Play you are using.
- The `ScalaDocOptions.styleSheet` property has been removed — the Scaladoc Ant task in Scala 2.11.8 and later no longer supports this property.

Kotlin DSL

- Artifact configuration accessors now have the type `NamedDomainObjectProvider<Configuration>` instead of `Configuration`
- `PluginAware.apply<T>(to)` was renamed `PluginAware.applyTo<T>(target)`.

Both changes could cause script compilation errors. See the Gradle Kotlin DSL release notes for more information and how to fix builds broken by the changes described above.

Miscellaneous

- The `ConfigurableReport.setDestination(Object)` method has been removed — use ConfigurableReport.setDestination(File) instead.
- The `Signature.setFile(File)` method has been removed — Gradle does not support changing the output file for the generated signature.
- The read-only `Signature.toSignArtifact` property has been removed — it should never have been part of the public API.
- The `@DeferredConfigurable` annotation has been removed.
- The method `isDeferredConfigurable()` was removed from `ExtensionSchema`.
- `IdeaPlugin.performPostEvaluationActions()` and `EclipsePlugin.performPostEvaluationActions()` have been removed.
- `The `BroadcastingCollectionEventRegister.getAddAction()` method has been removed with no replacement.
- The internal `org.gradle.util` package is no longer imported by default.

  Ideally you shouldn't use classes from this package, but, as a quick fix, you can add explicit imports to your build scripts for those classes.

- The `gradlePluginPortal()` repository no longer looks for JARs without a POM by default.
- The Tooling API can no longer connect to builds using a Gradle version below Gradle 2.6. The same applies to builds run through TestKit.
- Gradle 5.0 requires a minimum Tooling API client version of 3.0. Older client libraries can no longer run builds with Gradle 5.0.
- The IdeaModule Tooling API model element contains methods to retrieve resources and test resources so those elements were removed from the result of `IdeaModule.getSourceDirs()` and `IdeaModule.getTestSourceDirs()`.
- In previous Gradle versions, the `source` field in `SourceTask` was accessible from subclasses.

This is not the case anymore as the `source` field is now declared as `private`.

- In the Worker API, the working directory of a worker can no longer be set.
- A change in behavior related to dependency and version constraints may impact a small number of users.
- There have been several changes to property factory methods on DefaultTask that may impact the creation of custom tasks.

Upgrading from 4.9 and earlier

If you are not already on version 4.9, skip down to the section that applies to your current Gradle version and work your way up until you reach here. Then, apply these changes when upgrading to Gradle 4.10.

Deprecated classes, methods and properties

Follow the API links to learn how to deal with these deprecations (if no extra information is provided here):

- `TaskContainer.add()` and `TaskContainer.addAll()` — use TaskContainer.create() or TaskContainer.register() instead

Potential breaking changes

- There have been several potentially breaking changes in Kotlin DSL — see the *Breaking changes* section of that project's release notes.
- You can no longer use any of the Project.beforeEvaluate() or Project.afterEvaluate() methods with lazy task configuration, for example inside a TaskContainer.register() block.
- Publishing to AWS S3 requires new permissions.
- Both PluginUnderTestMetadata and GeneratePluginDescriptors — classes used by the Java Gradle Plugin Development Plugin — have been updated to use the Provider API.

  Use the Property.set() method to modify their values rather than using standard property assignment syntax, unless you are doing so in a Groovy build script. Standard property assignment still works in that one case.

Upgrading from 4.8 and earlier

- Consider trying the lazy API for task creation and configuration

Potential breaking changes

- You can no longer use GPath syntax with tasks.withType().

  Use Groovy's spread operator instead. For example, you would replace `tasks.withType(JavaCompile).name` with `tasks.withType(JavaCompile)*.name`.

Upgrading from 4.7 and earlier

- Switch to the Maven Publish and Ivy Publish plugins
- Use deferred configuration with the publishing plugins
- Configure existing `wrapper` and `init` tasks rather than defining your own
- Consider migrating to the built-in dependency locking mechanism if you are currently using a plugin or custom solution for this

Potential breaking changes

- Build will now fail if a specified init script is not found.
- `TaskContainer.remove()` now actually removes the given task — some plugins may have accidentally relied on the old behavior.
- Gradle now honors implicit wildcards in Maven POM exclusions.
- The Kotlin DSL now respects JSR-305 package annotations.

  This will lead to some types annotated according to JSR-305 being treated as nullable where they were treated as non-nullable before. This may lead to compilation errors in the build script. See the relevant Kotlin DSL release notes for details.

- Error messages will be directed to standard error rather than standard output now, unless a console is attached to both standard output and standard error. This may affect tools that scrape a build's plain console output. Ignore this change if you're upgrading from an earlier version of Gradle.

Deprecations

Prior to this release, builds were allowed to replace built-in tasks. This feature has been deprecated.

The full list of built-in tasks that should not be replaced is: `wrapper`, `init`, `help`, `tasks`, `projects`, `buildEnvironment`, `components`, `dependencies`, `dependencyInsight`, `dependentComponents`, `model`, `properties`.

Upgrading from 4.6 and earlier

Potential breaking changes

- Gradle will now, by convention, look for Checkstyle configuration files in the root project's *config/checkstyle* directory.

  Checkstyle configuration files in subprojects — the old by-convention location — will be ignored unless you explicitly configure their path via checkstyle.configDir or checkstyle.config.

- The structure of Gradle's plain console output has changed, which may break tools that scrape that output.
- The APIs of many native tasks related to compilation, linking and installation have changed in breaking ways.

- [Kotlin DSL] Delegated properties used to access Gradle's build properties — defined in *gradle.properties* for example — must now be explicitly typed.
- [Kotlin DSL] Declaring a `plugins {}` block inside a nested scope now throws an exception.
- [Kotlin DSL] Only one `pluginManagement {}` block is allowed now.
- The cache control DSL provided by the `org.gradle.api.artifacts.cache.*` interfaces are no longer available.
- `getEnabledDirectoryReportDestinations()`, `getEnabledFileReportDestinations()` and `getEnabledReportNames()` have all been removed from `org.gradle.api.reporting.ReportContainer`.
- StartParameter.projectProperties and StartParameter.systemPropertiesArgs now return immutable maps.

Upgrading from 4.5 and earlier

Deprecations

- You should not put annotation processors on the compile classpath or declare them with the `-processorpath` compiler argument.

  They should be added to the `annotationProcessor` configuration instead. If you don't want any processing, but your compile classpath contains a processor unintentionally (e.g. as part of a library you depend on), use the `-proc:none` compiler argument to ignore it.

- Use CommandLineArgumentProvider in place of CompilerArgumentProvider.

Potential breaking changes

- The Java plugins now add a `sourceSetAnnotationProcessor` configuration for each source set, which might break if any of them match existing configurations you have. We recommend you remove your conflicting configuration declarations.
- The `StartParameter.taskOutputCacheEnabled` property has been replaced by StartParameter.setBuildCacheEnabled(boolean).
- The Visual Studio integration now only configures a single solution for all components in a build.
- Gradle has replaced HttpClient 4.4.1 with version 4.5.5.
- Gradle now bundles the `kotlin-stdlib-jdk8` artifact instead of `kotlin-stdlib-jre8`. This may affect your build. Please see the Kotlin documentation for more details.

Upgrading from 4.4 and earlier

- Make sure you have a *settings.gradle* file: it avoids a performance penalty and allows you to set the root project's name.
- Gradle now ignores the build cache configuration of included builds (composite builds) and instead uses the root build's configuration for all the builds.

Potential breaking changes

- Two overloaded `ValidateTaskProperties.setOutputFile()` methods were removed. They are replaced with auto-generated setters when the task is accessed from a build script, but that won't be the case from plugins and other code outside of the build script.
- The Maven Publish Plugin now produces more complete maven-metadata.xml files, including maintaining a list of `<snapshotVersion>` elements. Some older versions of Maven may not be able to consume this metadata.
- `HttpBuildCache` no longer follows redirects.
- The `Depend` task type has been removed.
- Project.file(Object) no longer normalizes case for file paths on case-insensitive file systems. It now ignores case in such circumstances and does not touch the file system.
- ListProperty no longer extends Property.

Upgrading from 4.3 and earlier

Potential breaking changes

- AbstractTestTask is now extended by non-JVM test tasks as well as Test. Plugins should beware configuring all tasks of type `AbstractTestTask` because of this.
- The default output location for EclipseClasspath.defaultOutputDir has changed from `$projectDir`*/bin* to `$projectDir`*/bin/default*.
- The deprecated `InstallExecutable.setDestinationDir(Provider)` was removed — use InstallExecutable.installDirectory instead.
- The deprecated `InstallExecutable.setExecutable(Provider)` was removed — use InstallExecutable.executableFile instead.
- Gradle will no longer prefer a version of Visual Studio found on the path over other locations. It is now a last resort.

  You can bypass the toolchain discovery by specifying the installation directory of the version of Visual Studio you want via VisualCpp.setInstallDir(Object).

- `pluginManagement.repositories` is now of type RepositoryHandler rather than `PluginRepositoriesSpec`, which has been removed.
- 5xx HTTP errors during dependency resolution will now trigger exceptions in the build.
- The embedded Apache Ant has been upgraded from 1.9.6 to 1.9.9.
- Several third-party libraries used by Gradle have been upgraded to fix security issues.

Upgrading from 4.2 and earlier

- The `plugins {}` block can now be used in subprojects and for plugins in the *buildSrc* directory.

Other deprecations

- You should no longer run Gradle versions older than 2.6 via the Tooling API.
- You should no longer run any version of Gradle via an older version of the Tooling API than 3.0.
- You should no longer chain TaskInputs.property(String,Object) and TaskInputs.properties(Map) methods.

Potential breaking changes

- DefaultTask.newOutputDirectory() now returns a `DirectoryProperty` instead of a `DirectoryVar`.
- DefaultTask.newOutputFile() now returns a `RegularFileProperty` instead of a `RegularFileVar`.
- DefaultTask.newInputFile() now returns a `RegularFileProperty` instead of a `RegularFileVar`.
- ProjectLayout.buildDirectory now returns a `DirectoryProperty` instead of a `DirectoryVar`.
- AbstractNativeCompileTask.compilerArgs is now of type `ListProperty<String>` instead of `List<String>`.
- AbstractNativeCompileTask.objectFileDir is now of type `DirectoryProperty` instead of `File`.
- AbstractLinkTask.linkerArgs is now of type `ListProperty<String>` instead of `List<String>`.
- `TaskDestroyables.getFiles()` is no longer part of the public API.
- Overlapping version ranges for a dependency now result in Gradle picking a version that satisfies all declared ranges.

  For example, if a dependency on `some-module` is found with a version range of `[3,6]` and also transitively with a range of `[4,8]`, Gradle now selects version 6 instead of 8. The prior behavior was to select 8.

- The order of elements in `Iterable` properties marked with either `@OutputFiles` or `@OutputDirectories` now matters. If the order changes, the property is no longer considered up to date.

  Prefer using separate properties with `@OutputFile`/`@OutputDirectory` annotations or use `Map` properties with `@OutputFiles`/`@OutputDirectories` instead.

- Gradle will no longer ignore dependency resolution errors from a repository when there is another repository it can check. Dependency resolution will fail instead. This results in more deterministic behavior with respect to resolution results.

Upgrading from 4.1 and earlier

Potential breaking changes

- The `withPathSensitivity()` methods on TaskFilePropertyBuilder and TaskOutputFilePropertyBuilder have been removed.
- The bundled `bndlib` has been upgraded from 3.2.0 to 3.4.0.
- The FindBugs Plugin no longer renders progress information from its analysis. If you rely on that output in any way, you can enable it with *FindBugs.showProgress*.

Upgrading from 4.0

- Consider using the new Worker API to enable units of work within your build to run in parallel.

Deprecated classes, methods and properties

Follow the API links to learn how to deal with these deprecations (if no extra information is provided here):

- Nullable

Potential breaking changes

- Non-Java projects that have a project dependency on a Java project now consume the `runtimeElements` configuration by default instead of the `default` configuration.

  To override this behavior, you can explicitly declare the configuration to use in the project dependency. For example: `project(path: ':myJavaProject', configuration: 'default')`.

- Default Zinc compiler upgraded from 0.3.13 to 0.3.15.
- [Kotlin DSL] Base package renamed from `org.gradle.script.lang.kotlin` to `org.gradle.kotlin.dsl`.

Changes in detail

[5.0] Default memory settings changed

The command line client now starts with 64MB of heap instead of 1GB. This may affect builds running directly inside the client VM using `--no-daemon` mode. We discourage the use of `--no-daemon`, but if you must use it, you can increase the available memory using the `GRADLE_OPTS` environment variable.

The Gradle daemon now starts with 512MB of heap instead of 1GB. Large projects may have to increase this setting using the `org.gradle.jvmargs` property.

All workers, including compilers and test executors, now start with 512MB of heap. The previous default was 1/4th of physical memory. Large projects may have to increase this setting on the relevant tasks, e.g. `JavaCompile` or `Test`.

[5.0] New default versions for code quality plugins

The default tool versions of the following code quality plugins have been updated:

- The Checkstyle Plugin now uses 8.12 instead of 6.19 by default.
- The CodeNarc Plugin now uses 1.2.1 instead of 1.1 by default.
- The JaCoCo Plugin now uses 0.8.2 instead of 0.8.1 by default.
- The PMD Plugin now uses 6.8.0 instead of 5.6.1 by default.

  In addition, the default ruleset was changed from the now deprecated `java-basic` to

`category/java/errorprone.xml`.

We recommend configuring a ruleset explicitly, though.

[5.0] Library upgrades

Several libraries that are used by Gradle have been upgraded:

- Groovy was upgraded from 2.4.15 to 2.5.4.
- Ant has been upgraded from 1.9.11 to 1.9.13.
- The AWS SDK used to access S3-backed Maven/Ivy repositories has been upgraded from 1.11.267 to 1.11.407.
- The BND library used by the OSGi Plugin has been upgraded from 3.4.0 to 4.0.0.
- The Google Cloud Storage JSON API Client Library used to access Google Cloud Storage backed Maven/Ivy repositories has been upgraded from v1-rev116-1.23.0 to v1-rev136-1.25.0.
- Ivy has been upgraded from 2.2.0 to 2.3.0.
- The JUnit Platform libraries used by the `Test` task have been upgraded from 1.0.3 to 1.3.1.
- The Maven Wagon libraries used to access Maven repositories have been upgraded from 2.4 to 3.0.0.
- SLF4J has been upgraded from 1.7.16 to 1.7.25.

[5.0] Improved support for dependency and version constraints

Through the Gradle 4.x release stream, new `@Incubating` features were added to the dependency resolution engine. These include sophisticated version constraints (`prefer`, `strictly`, `reject`), dependency constraints, and `platform` dependencies.

If you have been using the `IMPROVED_POM_SUPPORT` feature preview, playing with constraints or prefer, reject and other specific version indications, then make sure to take a good look at your dependency resolution results.

[5.0] BOM import

Gradle now provides support for importing bill of materials (BOM) files, which are effectively POM files that use `<dependencyManagement>` sections to control the versions of direct and transitive dependencies. All you need to do is declare the POM as a `platform` dependency.

The following example picks the versions of the `gson` and `dom4j` dependencies from the declared Spring Boot BOM:

```
dependencies {
    // import a BOM
    implementation platform('org.springframework.boot:spring-boot-
dependencies:1.5.8.RELEASE')

    // define dependencies without versions
    implementation 'com.google.code.gson:gson'
    implementation 'dom4j:dom4j'
}
```

[5.0] Separation of compile and runtime dependencies when consuming POMs

Since Gradle 1.0, runtime-scoped dependencies have been included in the Java compilation classpath, which has some drawbacks:

- The compilation classpath is much larger than it needs to be, slowing down compilation.
- The compilation classpath includes runtime-scoped files that do not impact compilation, resulting in unnecessary re-compilation when those files change.

With this new behavior, the Java and Java Library plugins both honor the separation of compile and runtime scopes. This means that the compilation classpath only includes compile-scoped dependencies, while the runtime classpath adds the runtime-scoped dependencies as well. This is particularly useful if you develop and publish Java libraries with Gradle where the separation between `api` and `implementation` dependencies is reflected in the published scopes.

[5.0] Changes to property factory methods on `DefaultTask`

Property factory methods on `DefaultTask` are now final

The property factory methods such as `newInputFile()` are intended to be called from the constructor of a type that extends `DefaultTask`. These methods are now final to avoid subclasses overriding these methods and using state that is not initialized.

Inputs and outputs are not automatically registered

The Property instances that are returned by these methods are no longer automatically registered as inputs or outputs of the task. The Property instances need to be declared as inputs or outputs in the usual ways, such as attaching annotations such as `@OutputFile` or using the runtime API to register the property.

For example, you could previously use the following syntax and have both outputFile instances registered as declared outputs:

build.gradle

```
class MyTask extends DefaultTask {
    // note: no annotation here
    final RegularFileProperty outputFile = newOutputFile()
}

task myOtherTask {
    def outputFile = newOutputFile()
    doLast { ... }
}
```

build.gradle.kts

```
open class MyTask : DefaultTask() {
    // note: no annotation here
    val outputFile: RegularFileProperty = newOutputFile()
}

task("myOtherTask") {
    val outputFile = newOutputFile()
    doLast { ... }
}
```

Now you have to explicitly register `outputFile`, like this:

build.gradle

```
class MyTask extends DefaultTask {
    @OutputFile // property needs an annotation
    final RegularFileProperty outputFile = project.objects.fileProperty()
}

task myOtherTask {
    def outputFile = project.objects.fileProperty()
    outputs.file(outputFile) // or to be registered using the runtime API
    doLast { ... }
}
```

build.gradle.kts

```
open class MyTask : DefaultTask() {
    @OutputFile // property needs an annotation
    val outputFile: RegularFileProperty = project.objects.fileProperty()
}

task("myOtherTask") {
    val outputFile = project.objects.fileProperty()
    outputs.file(outputFile) // or to be registered using the runtime API
    doLast { ... }
}
```

[5.0] Gradle now bundles JAXB for Java 9 and above

In order to use S3 backed artifact repositories, you previously had to add `--add-modules java.xml.bind` to `org.gradle.jvmargs` when running on Java 9 and above.

Since Java 11 no longer contains the `java.xml.bind` module, Gradle now bundles JAXB 2.3.1 (`com.sun.xml.bind:jaxb-impl`) and uses it on Java 9 and above.

Please remove the `--add-modules java.xml.bind` option from `org.gradle.jvmargs`, if set.

[5.0] The `gradlePluginPortal()` repository no longer looks for JARs without a POM by default

With this new behavior, if a plugin or a transitive dependency of a plugin found in the `gradlePluginPortal()` repository has no Maven POM it will fail to resolve.

Artifacts published to a Maven repository without a POM should be fixed. If you encounter such artifacts, please ask the plugin or library author to publish a new version with proper metadata.

If you are stuck with a bad plugin, you can work around by re-enabling JARs as metadata source for

the `gradlePluginPortal()` repository:

settings.gradle

```
pluginManagement {
    repositories {
        gradlePluginPortal().tap {
            metadataSources {
                mavenPom()
                artifact()
            }
        }
    }
}
```

settings.gradle.kts

```
pluginManagement {
    repositories {
        gradlePluginPortal().apply {
            (this as MavenArtifactRepository).metadataSources {
                mavenPom()
                artifact()
            }
        }
    }
}
```

Java Library Distribution Plugin utilizes Java Library Plugin

The Java Library Distribution Plugin is now based on the Java Library Plugin instead of the Java Plugin.

Additionally, the default distribution created by the plugin will contain all artifacts of the `runtimeClasspath` configuration instead of the deprecated `runtime` configuration.

Configuration Avoidance API disallows common configuration errors

The configuration avoidance API introduced in Gradle 4.9 allows you to avoid creating and configuring tasks that are never used.

With the existing API, this example adds two tasks (`foo` and `bar`):

build.gradle

```
tasks.create("foo") {
    tasks.create("bar")
}
```

build.gradle.kts

```
tasks.create("foo") {
    tasks.create("bar")
}
```

When converting this to use the new API, something surprising happens: `bar` doesn't exist. The new API only executes configuration actions when necessary, so the `register()` for task `bar` only executes when `foo` is configured.

build.gradle

```
tasks.register("foo") {
    tasks.register("bar") // WRONG
}
```

build.gradle.kts

```
tasks.register("foo") {
    tasks.register("bar") // WRONG
}
```

To avoid this, Gradle now detects this and prevents modification to the underlying container (through `create()` or `register()`) when using the new API.

[5.0] Worker API: working directory of a worker can no longer be set

Since JDK 11 no longer supports changing the working directory of a running process, setting the working directory of a worker via its fork options is now prohibited.

All workers now use the same working directory to enable reuse.

Please pass files and directories as arguments instead.

[4.10] Publishing to AWS S3 requires new permissions

The S3 repository transport protocol allows Gradle to publish artifacts to AWS S3 buckets. Starting with this release, every artifact uploaded to an S3 bucket will be equipped with the `bucket-owner-full-control` canned ACL. Make sure that the AWS account used to publish artifacts has the `s3:PutObjectAcl` and `s3:PutObjectVersionAcl` permissions, otherwise the upload will fail.

```
{
    "Version":"2012-10-17",
    "Statement":[
        // ...
        {
            "Effect":"Allow",
            "Action":[
                "s3:PutObject", // necessary for uploading objects
                "s3:PutObjectAcl", // required starting with this release
                "s3:PutObjectVersionAcl" // if S3 bucket versioning is enabled
            ],
            "Resource":"arn:aws:s3:::myCompanyBucket/*"
        }
    ]
}
```

See AWS S3 Cross Account Access for more information.

[4.9] Consider trying the lazy API for task creation and configuration

Gradle 4.9 introduced a new way to create and configure tasks that works lazily. When you use this approach for tasks that are expensive to configure, or when you have many, many tasks, your build configuration time can drop significantly when those tasks don't run.

You can learn more about lazily creating tasks in the Task Configuration Avoidance chapter. You can also read about the background to this new feature in this blog post.

[4.8] Switch to the Maven Publish and Ivy Publish Plugins

Now that the publishing plugins are stable, we recommend that you migrate from the legacy publishing mechanism for standard Java projects, i.e. those based on the Java Plugin. That includes projects that use any one of: Java Library Plugin, Application Plugin or War Plugin.

To use the new approach, simply replace any `upload<Conf>` configuration with a `publishing {}` block. See the publishing overview chapter for more information.

[4.8] Use deferred configuration for publishing plugins

Prior to Gradle 4.8, the `publishing {}` block was implicitly treated as if all the logic inside it was executed after the project was evaluated. This was confusing, because it was the only block that behaved that way. As part of the stabilization effort in Gradle 4.8, we are deprecating this behavior and asking all users to migrate their build.

The new, stable behavior can be switched on by adding the following to your settings file:

settings.gradle

```
enableFeaturePreview('STABLE_PUBLISHING')
```

settings.gradle.kts

```
enableFeaturePreview("STABLE_PUBLISHING")
```

We recommend doing a test run with a local repository to see whether all artifacts still have the expected coordinates. In most cases everything should work as before and you are done. However, your publishing block may rely on the implicit deferred configuration, particularly if it relies on values that may change during the configuration phase of the build.

For example, under the new behavior, the following logic assumes that `jar.archiveBaseName` doesn't change after `artifactId` is set:

build.gradle

```
subprojects {
    publishing {
        publications {
            mavenJava {
                from components.java
                artifactId = jar.archiveBaseName
            }
        }
    }
}
```

build.gradle.kts

```
subprojects {
    publishing {
        publications {
            named<MavenPublication>("mavenJava") {
                from(components["java"])
                artifactId = tasks.jar.get().archiveBaseName.get()
            }
        }
    }
}
```

If that assumption is incorrect or might possibly be incorrect in the future, the `artifactId` must be set within an `afterEvaluate {}` block, like so:

build.gradle

```
subprojects {
    publishing {
        publications {
            mavenJava {
                from components.java
                afterEvaluate {
                    artifactId = jar.archiveBaseName
                }
            }
        }
    }
}
```

build.gradle.kts

```
subprojects {
    publishing {
        publications {
            named<MavenPublication>("mavenJava") {
                from(components["java"])
                afterEvaluate {
                    artifactId = tasks.jar.get().archiveBbaseName.get()
                }
            }
        }
    }
}
```

[4.8] Configure existing `wrapper` and `init` tasks

You should no longer define your own `wrapper` and `init` tasks. Configure the existing tasks instead, for example by converting this:

build.gradle

```
task wrapper(type: Wrapper) {
    ...
}
```

build.gradle.kts

```
task<Wrapper>("wrapper") {
    ...
}
```

to this:

build.gradle

```
wrapper {
    ...
}
```

build.gradle.kts

```
tasks.wrapper {
    ...
}
```

[4.8] Gradle now honors implicit wildcards in Maven POM exclusions

If an exclusion in a Maven POM was missing either a `groupId` or `artifactId`, Gradle used to ignore the exclusion. Now the missing elements are treated as implicit wildcards — e.g. `<groupId>*</groupId>` — which means that some of your dependencies may now be excluded where they weren't before.

You will need to explicitly declare any missing dependencies that you need.

[4.7] Changes to the structure of Gradle's plain console output

The plain console mode now formats output consistently with the rich console, which means that the output format has changed. For example:

- The output produced by a given task is now grouped together, even when other tasks execute in parallel with it.
- Task execution headers are printed with a "> Task" prefix.
- All output produced during build execution is written to the standard output file handle. This includes messages written to System.err unless you are redirecting standard error to a file or any other non-console destination.

This may break tools that scrape details from the plain console output.

[4.6] Changes to the APIs of native tasks related to compilation, linking and installation

Many tasks related to compiling, linking and installing native libraries and applications have been converted to the Provider API so that they support lazy configuration. This conversion has introduced some breaking changes to the APIs of the tasks so that they match the conventions of the Provider API.

The following tasks have been changed:

AbstractLinkTask and its subclasses

- `getDestinationDir()` was replaced by `getDestinationDirectory()`.
- `getBinaryFile()`, `getOutputFile()` was replaced by `getLinkedFile()`.
- `setOutputFile(File)` was removed. Use `Property.set()` instead.
- `setOutputFile(Provider)` was removed. Use `Property.set()` instead.
- `getTargetPlatform()` was changed to return a `Property`.
- `setTargetPlatform(NativePlatform)` was removed. Use `Property.set()` instead.
- `getToolChain()` was changed to return a `Property`.
- `setToolChain(NativeToolChain)` was removed. Use `Property.set()` instead.

CreateStaticLibrary

- `getOutputFile()` was changed to return a `Property`.
- `setOutputFile(File)` was removed. Use `Property.set()` instead.
- `setOutputFile(Provider)` was removed. Use `Property.set()` instead.
- `getTargetPlatform()` was changed to return a `Property`.
- `setTargetPlatform(NativePlatform)` was removed. Use `Property.set()` instead.
- `getToolChain()` was changed to return a `Property`.
- `setToolChain(NativeToolChain)` was removed. Use `Property.set()` instead.
- `getStaticLibArgs()` was changed to return a `ListProperty`.
- `setStaticLibArgs(List)` was removed. Use `ListProperty.set()` instead.

InstallExecutable

- `getSourceFile()` was replaced by `getExecutableFile()`.
- `getPlatform()` was replaced by `getTargetPlatform()`.

- `setTargetPlatform(NativePlatform)` was removed. Use `Property.set()` instead.
- `getToolChain()` was changed to return a `Property`.
- `setToolChain(NativeToolChain)` was removed. Use `Property.set()` instead.

The following have also seen similar changes:

- Assemble
- WindowsResourceCompile
- StripSymbols
- ExtractSymbols
- SwiftCompile
- LinkMachOBundle

[4.6] Visual Studio integration only supports a single solution file for all components of a build

VisualStudioExtension no longer has a `solutions` property. Instead, you configure a single solution via VisualStudioRootExtension in the root project, like so:

build.gradle

```
model {
    visualStudio {
        solution {
            solutionFile.location = "vs/${name}.sln"
        }
    }
}
```

In addition, there are no longer individual tasks to generate the solution files for each component, but rather a single `visualStudio` task that generates a solution file that encompasses all components in the build.

[4.5] `HttpBuildCache` no longer follows redirects

When connecting to an HTTP build cache backend via `HttpBuildCache`, Gradle does not follow redirects any more, treating them as errors instead. Getting a redirect from the build cache backend is mostly a configuration error — using an "http" URL instead of "https" for example — and has negative effects on performance.

[4.4] Third-party dependency upgrades

This version includes several upgrades of third-party dependencies:

- jackson: 2.6.6 → 2.8.9
- plexus-utils: 2.0.6 → 2.1
- xercesImpl: 2.9.1 → 2.11.0
- bsh: 2.0b4 → 2.0b6
- bouncycastle: 1.57 → 1.58

This fix the following security issues:

- CVE-2017-7525 (critical)
- SONATYPE-2017-0359 (critical)
- SONATYPE-2017-0355 (critical)
- SONATYPE-2017-0398 (critical)
- CVE-2013-4002 (critical)
- CVE-2016-2510 (severe)
- SONATYPE-2016-0397 (severe)
- CVE-2009-2625 (severe)
- SONATYPE-2017-0348 (severe)

Gradle does not expose public APIs for these 3rd-party dependencies, but those who customize Gradle will want to be aware.

Migrating Builds From Apache Maven

TIP	Suffering from slow Maven builds? Register here for our Build Cache training session to learn how Gradle Enterprise can speed up Maven builds by up to 90%.

Apache Maven is a build tool for Java and other JVM-based projects that's in widespread use, and so people that want to use Gradle often have to migrate an existing Maven build. This guide will help with such a migration by explaining the differences and similarities between the two tools' models and providing steps that you can follow to ease the process.

Converting a build can be scary, but you don't have to do it alone. You can search docs, forums, and StackOverflow from help.gradle.org or reach out to the Gradle community on the forums if you get stuck.

Making a case for migration

The primary differences between Gradle and Maven are flexibility, performance, user experience, and dependency management. A visual overview of these aspects is available in the Maven vs Gradle feature comparison.

Since Gradle 3.0, Gradle has invested heavily in making Gradle builds much faster, with features such as build caching, compile avoidance, and an improved incremental Java compiler. Gradle is now 2-10x faster than Maven for the vast majority of projects, even without using a build cache. In-

depth performance comparison and business cases for switching from Maven to Gradle can be found here.

General guidelines

Gradle and Maven have fundamentally different views on how to build a project. Gradle provides a flexible and extensible build model that delegates the actual work to the execution of a graph of tasks. Maven uses a model of fixed, linear phases to which you can attach goals (the things that do the work). This may make migrating between the two seem intimidating, but migrations can be surprisingly easy because Gradle follows many of the same conventions as Maven — such as the standard project structure — and its dependency management works in a similar way.

Here we lay out a series of steps for you to follow that will help facilitate the migration of any Maven build to Gradle:

TIP	Keep the old Maven build and new Gradle build side by side. You know the Maven build works, so you should keep it until you are confident that the Gradle build produces all the same artifacts and otherwise does what you need. This also means that users can try the Gradle build without getting a new copy of the source tree.

1. Create a build scan for the Maven build.

   A build scan will make it easier to visualize what's happening in your existing Maven build. For Maven builds, you'll be able to see the project structure, what plugins are being used, a timeline of the build steps, and more. Keep this handy so you can compare it to the Gradle build scans you get while converting the project.

2. Develop a mechanism to verify that the two builds produce the same artifacts

   This is a vitally important step to ensure that your deployments and tests don't break. Even small changes, such as the contents of a manifest file in a JAR, can cause problems. If your Gradle build produces the same output as the Maven build, this will give you and others confidence in switching over and make it easier to implement the big changes that will provide the greatest benefits.

   This doesn't mean that you need to verify every artifact at every stage, although doing so can help you quickly identify the source of a problem. You can just focus on the critical output such as final reports and the artifacts that are published or deployed.

   You will need to factor in some inherent differences in the build output that Gradle produces compared to Maven. Generated POMs will contain only the information needed for consumption and they will use `<compile>` and `<runtime>` scopes correctly for that scenario. You might also see differences in the order of files in archives and of files on classpaths. Most differences will be benign, but it's worth identifying them and verifying that they are OK.

3. Run an automatic conversion

   This will create all the Gradle build files you need, even for multi-module builds. For simpler Maven projects, the Gradle build will be ready to run!

4. Create a build scan for the Gradle build.

   A build scan will make it easier to visualize what's happening in the build. For Gradle builds, you'll be able to see the project structure, the dependencies (regular and inter-project ones), what plugins are being used and the console output of the build.

   Your build may fail at this point, but that's ok; the scan will still run. Compare the build scan for the Gradle build to the one for the Maven build and continue down this list to troubleshoot the failures.

   We recommend that you regularly generate build scans during the migration to help you identify and troubleshoot problems. If you want, you can also use a Gradle build scan to identify opportunities to improve the performance of the build, after all performance is a big reason for switching to Gradle in the first place.

5. Verify your dependencies and fix any problems
6. Configure integration and functional tests

   Many tests can simply be migrated by configuring an extra source set. If you are using a third-party library, such as FitNesse, look to see whether there is a suitable community plugin available on the Gradle Plugin Portal.

7. Replace Maven plugins with Gradle equivalents

   In the case of popular plugins, Gradle often has an equivalent plugin that you can use. You might also find that you can replace a plugin with built-in Gradle functionality. As a last resort, you may need to reimplement a Maven plugin via your own custom plugins and task types.

The rest of this chapter looks in more detail at specific aspects of migrating a build from Maven to Gradle.

Understanding the build lifecycle

Maven builds are based around the concept of *build lifecycles* that consist of a set of fixed phases. This can prove an impediment for users migrating to Gradle because its build lifecycle is something different, although it's important to understand how Gradle builds fit into the structure of initialization, configuration, and execution phases. Fortunately, Gradle has a feature that can mimic Maven's phases: *lifecycle tasks*.

These allow you to define your own "lifecycles" by creating no-action tasks that simply depend on the tasks you're interested in. And to make the transition to Gradle easier for Maven users, the Base Plugin — applied by all the JVM language plugins like the Java Library Plugin — provides a set of lifecycle tasks that correspond to the main Maven phases.

Here is a list of some of the main Maven phases and the Gradle tasks that they map to:

`clean`

Use the `clean` task provided by the Base Plugin.

`compile`

Use the `classes` task provided by the Java Plugin and other JVM language plugins. This compiles all classes for all source files of all languages and also performs resource filtering via the `processResources` task.

`test`

Use the `test` task provided by the Java Plugin. It runs just the unit tests, or more specifically, the tests that make up the `test` source set.

`package`

Use the `assemble` task provided by the Base Plugin. This builds whatever is the appropriate package for the project, for example a JAR for Java libraries or a WAR for traditional Java webapps.

`verify`

Use the `check` task provided by the Base Plugin. This runs all verification tasks that are attached to it, which typically includes the unit tests, any static analysis tasks — such as Checkstyle — and others. If you want to include integration tests, you will have to configure these manually, which is a simple process.

`install`

Use the `publishToMavenLocal` task provided by the Maven Publish Plugin.

Note that Gradle builds don't require you to "install" artifacts as you have access to more appropriate features like inter-project dependencies and composite builds. You should only use `publishToMavenLocal` for interoperating with Maven builds.

Gradle also allows you to resolve dependencies against the local Maven cache, as described in the Declaring repositories section.

`deploy`

Use the `publish` task provided by the Maven Publish Plugin — making sure you switch from the older Maven Plugin (ID: `maven`) if your build is using that one. This will publish your package to all configured publication repositories. There are also other tasks that allow you to publish to a single repository even when multiple ones are defined.

Note that the Maven Publish Plugin does not publish **source and Javadoc JARs** *by default*, but this can easily be activated as explained in the guide for building java projects.

Performing an automatic conversion

Gradle's `init` task is typically used to create a new skeleton project, but you can also use it to convert an existing Maven build to Gradle automatically. Once Gradle is installed on your system, all you have to do is run the command

```
> gradle init
```

from the root project directory and let Gradle do its thing. That basically consists of parsing the

existing POMs and generating the corresponding Gradle build scripts. Gradle will also create a settings script if you're migrating a multi-project build.

You'll find that the new Gradle build includes the following:

- All the custom repositories that are specified in the POM
- Your external and inter-project dependencies
- The appropriate plugins to build the project (limited to one or more of the Maven Publish, Java and War Plugins)

See the Build Init Plugin chapter for a complete list of the automatic conversion features.

One thing to bear in mind is that assemblies are not automatically converted. They aren't necessarily problematic to convert, but you will need to do some manual work. Options include:

- Using the Distribution Plugin
- Using the Java Library Distribution Plugin
- Using the Application Plugin
- Creating custom archive tasks
- Using a suitable community plugin from the Gradle Plugin Portal

If your Maven build does not have many plugins or much in the way of customisation, you can simply run

```
> gradle build
```

once the migration has completed. This will run the tests and produce the required artifacts without any extra intervention on your part.

Migrating dependencies

Gradle's dependency management system is more flexible than Maven's, but it still supports the same concepts of repositories, declared dependencies, scopes (dependency configurations in Gradle), and transitive dependencies. In fact, Gradle works perfectly with Maven-compatible repositories, which makes it easy to migrate your dependencies.

NOTE	One notable difference between the two tools is in how they manage version conflicts. Maven uses a "closest" match algorithm, whereas Gradle picks the newest. Don't worry though, you have a lot of control over which versions are selected, as documented in Managing Transitive Dependencies.

Over the following sections, we will show you how to migrate the most common elements of a Maven build's dependency management information.

Declaring dependencies

Gradle uses the same dependency identifier components as Maven: group ID, artifact ID and version. It also supports classifiers. So all you need to do is substitute the identifier information for a dependency into Gradle's syntax, which is described in the Declaring Dependencies chapter.

For example, consider this Maven-style dependency on Log4J:

```
<dependencies>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.12</version>
    </dependency>
</dependencies>
```

This dependency would look like the following in a Gradle build script:

Example 1. Declaring a simple compile-time dependency

build.gradle

```
dependencies {
    implementation 'log4j:log4j:1.2.12'  ①
}
```

build.gradle.kts

```
dependencies {
    implementation("log4j:log4j:1.2.12")  ①
}
```

① Attaches version 1.2.12 of Log4J to the `implementation` configuration (scope)

The string identifier takes the Maven values of `groupId`, `artifactId` and `version`, although Gradle refers to them as `group`, `module` and `version`.

The above example raises an obvious question: what is that `implementation` configuration? It's one of the standard dependency configurations provided by the Java Plugin and is often used as a substitute for Maven's default `compile` scope.

Several of the differences between Maven's scopes and Gradle's standard configurations come down to Gradle distinguishing between the dependencies required to build a module and the dependencies required to build a module that depends on it. Maven makes no such distinction, so published POMs typically include dependencies that consumers of a library don't actually need.

Here are the main Maven dependency scopes and how you should deal with their migration:

`compile`

Gradle has two configurations that can be used in place of the `compile` scope: `implementation` and `api`. The former is available to any project that applies the Java Plugin, while `api` is only available to projects that specifically apply the Java Library Plugin.

In most cases you should simply use the `implementation` configuration, particularly if you're building an application or webapp. But if you're building a library, you can learn about which dependencies should be declared using `api` in the section on Building Java libraries. Even more information on the differences between `api` and `implementation` is provided in the Java Library Plugin chapter linked above.

`runtime`

Use the `runtimeOnly` configuration.

`test`

Gradle distinguishes between those dependencies that are required to *compile* a project's tests and those that are only needed to *run* them.

Dependencies required for test compilation should be declared against the `testImplementation` configuration. Those that are only required for running the tests should use `testRuntimeOnly`.

`provided`

Use the `compileOnly` configuration.

Note that the War Plugin adds `providedCompile` and `providedRuntime` dependency configurations. These behave slightly differently from `compileOnly` and simply ensure that those dependencies aren't packaged in the WAR file. However, the dependencies are included on runtime and test runtime classpaths, so use these configurations if that's the behavior you need.

`import`

The `import` scope is mostly used within `<dependencyManagement>` blocks and applies solely to POM-only publications. Read the section on Using bills of materials to learn more about how to replicate this behavior.

You can also specify a regular dependency on a POM-only publication. In this case, the dependencies declared in that POM are treated as normal transitive dependencies of the build.

For example, imagine you want to use the `groovy-all` POM for your tests. It's a POM-only publication that has its own dependencies listed inside a `<dependencies>` block. The appropriate configuration in the Gradle build looks like this:

Example 2. Consuming a POM-only dependency

build.gradle

```
dependencies {
    testImplementation 'org.codehaus.groovy:groovy-all:2.5.4'
}
```

build.gradle.kts

```
dependencies {
    testImplementation("org.codehaus.groovy:groovy-all:2.5.4")
}
```

The result of this will be that all `compile` and `runtime` scope dependencies in the `groovy-all` POM get added to the test runtime classpath, while only the `compile` scope dependencies get added to the test compilation classpath. Dependencies with other scopes will be ignored.

Declaring repositories

Gradle allows you to retrieve declared dependencies from any Maven-compatible or Ivy-compatible repository. Unlike Maven, it has no default repository and so you have to declare at least one. In order to have the same behavior as your Maven build, just configure Maven Central in your Gradle build, like this:

Example 3. Configuring the build to use Maven Central

build.gradle

```
repositories {
    mavenCentral()
}
```

build.gradle.kts

```
repositories {
    mavenCentral()
}
```

You can also use the `repositories {}` block to configure custom repositories, as described in the

Repository Types chapter.

Lastly, Gradle allows you to resolve dependencies against the local Maven cache/repository. This helps Gradle builds interoperate with Maven builds, but it shouldn't be a technique that you use if you don't need that interoperability. If you want to share published artifacts via the filesystem, consider configuring a custom Maven repository with a `file://` URL.

You might also be interested in learning about Gradle's own dependency cache, which behaves more reliably than Maven's and can be used safely by multiple concurrent Gradle processes.

Controlling dependency versions

The existence of transitive dependencies means that you can very easily end up with multiple versions of the same dependency in your dependency graph. By default, Gradle will pick the newest version of a dependency in the graph, but that's not always the right solution. That's why it provides several mechanisms for controlling which version of a given dependency is resolved.

On a per-project basis, you can use:

- Dependency constraints
- Bills of materials (Maven BOMs)
- Overriding transitive versions

There are even more, specialized options listed in the controlling transitive dependencies chapter.

If you want to ensure consistency of versions across all projects in a multi-project build, similar to how the `<dependencyManagement>` block in Maven works, you can use the Java Platform Plugin. This allows you declare a set of dependency constraints that can be applied to multiple projects. You can even publish the platform as a Maven BOM or using Gradle's metadata format. See the plugin page for more information on how to do that, and in particular the section on Consuming platforms to see how you can apply a platform to other projects in the same build.

Excluding transitive dependencies

Maven builds use exclusions to keep unwanted dependencies — or unwanted *versions* of dependencies — out of the dependency graph. You can do the same thing with Gradle, but that's not necessarily the *right* thing to do. Gradle provides other options that may be more appropriate for a given situation, so you really need to understand *why* an exclusion is in place to migrate it properly.

If you want to exclude a dependency for reasons unrelated to versions, then check out the section on Excluding transitive dependencies. It shows you how to attach an exclusion either to an entire configuration (often the most appropriate solution) or to a dependency. You can even easily apply an exclusion to all configurations.

If you're more interested in controlling which version of a dependency is actually resolved, see the previous section.

Handling optional dependencies

You are likely to encounter two situations regarding optional dependencies:

- Some of your transitive dependencies are declared as optional
- You want to declare some of your direct dependencies as optional in your project's published POM

For the first scenario, Gradle behaves the same way as Maven and simply ignores any transitive dependencies that are declared as optional. They are not resolved and have no impact on the versions selected if the same dependencies appear elsewhere in the dependency graph as non-optional.

As for publishing dependencies as optional, Gradle provides a richer model called feature variants, which will let you declare the "optional features" your library provides.

Using bills of materials (BOMs)

Maven allows you to share dependency constraints by defining dependencies inside a `<dependencyManagement>` section of a POM file that has a packaging type of `pom`. This special type of POM (a BOM) can then be imported into other POMs so that you have consistent library versions across your projects.

Gradle can use such BOMs for the same purpose, using a special dependency syntax based on platform() and enforcedPlatform() methods. You simply declare the dependency in the normal way, but wrap the dependency identifier in the appropriate method, as shown in this example that "imports" the Spring Boot Dependencies BOM:

Example 4. Importing a BOM in a Gradle build

build.gradle

```
dependencies {
    implementation platform('org.springframework.boot:spring-boot-
dependencies:1.5.8.RELEASE') ①

    implementation 'com.google.code.gson:gson' ②
    implementation 'dom4j:dom4j'
}
```

build.gradle.kts

```
dependencies {
    implementation(platform("org.springframework.boot:spring-boot-
dependencies:1.5.8.RELEASE"))  ①

    implementation("com.google.code.gson:gson")  ②
    implementation("dom4j:dom4j")
}
```

① Applies the Spring Boot Dependencies BOM

② Adds a dependency whose version is defined by that BOM

You can learn more about this feature and the difference between `platform()` and `enforcedPlatform()` in the section on importing version recommendations from a Maven BOM.

NOTE	You can use this feature to apply the `<dependencyManagement>` information from any dependency's POM to the Gradle build, even those that don't have a packaging type of `pom`. Both `platform()` and `enforcedPlatform()` will ignore any dependencies declared in the `<dependencies>` block.

Migrating multi-module builds (project aggregation)

Maven's multi-module builds map nicely to Gradle's multi-project builds. Try the corresponding sample to see how a basic multi-project Gradle build is set up.

To migrate a multi-module Maven build, simply follow these steps:

1. Create a settings script that matches the `<modules>` block of the root POM.

   For example, this `<modules>` block:

```
<modules>
    <module>simple-weather</module>
    <module>simple-webapp</module>
</modules>
```

can be migrated by adding the following line to the settings script:

Example 5. Declaring which projects are part of the build

settings.gradle

```
rootProject.name = 'simple-multi-module'  ①

include 'simple-weather', 'simple-webapp'  ②
```

settings.gradle.kts

```
rootProject.name = "simple-multi-module"  ①

include("simple-weather", "simple-webapp")  ②
```

① Sets the name of the overall project

② Configures two subprojects as part of this build

Output of `gradle projects`

```
> gradle projects

------------------------------------------------------------
Root project 'simple-multi-module'
------------------------------------------------------------

Root project 'simple-multi-module'
+--- Project ':simple-weather'
\--- Project ':simple-webapp'

To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :simple-weather:tasks
```

2. Replace cross-module dependencies with project dependencies.
3. Replicate project inheritance with convention plugins.

This basically involves creating a root project build script that injects shared configuration into

the appropriate subprojects.

Sharing versions across projects

If you want to replicate the Maven pattern of having dependency versions declared in the `dependencyManagement` section of the root POM file, the best approach is to leverage the `java-platform` plugin. You will need to add a dedicated project for this and consume it in the regular projects of your build. See the documentation for more details on this pattern.

Migrating Maven profiles and properties

Maven allows you parameterize builds using properties of various sorts. Some are read-only properties of the project model, others are user-defined in the POM. It even allows you to treat system properties as project properties.

Gradle has a similar system of project properties, although it differentiates between those and system properties. You can, for example, define properties in:

- the build script
- a `gradle.properties` file in the root project directory
- a `gradle.properties` file in the `$HOME/.gradle` directory

Those aren't the only options, so if you are interested in finding out more about how and where you can define properties, check out the Build Environment chapter.

One important piece of behavior you need to be aware of is what happens when the same property is defined in both the build script and one of the external properties files: the build script value takes precedence. Always. Fortunately, you can mimic the concept of profiles to provide overridable default values.

Which brings us on to Maven profiles. These are a way to enable and disable different configurations based on environment, target platform, or any other similar factor. Logically, they are nothing more than limited 'if' statements. And since Gradle has much more powerful ways to declare conditions, it does not need to have formal support for profiles (except in the POMs of dependencies). You can easily get the same behavior by combining conditions with secondary build scripts, as you'll see.

Let's say you have different deployment settings depending on the environment: local development (the default), a test environment, and production. To add profile-like behavior, you first create build scripts for each environment in the project root: `profile-default.gradle`, `profile-test.gradle`, and `profile-prod.gradle`. You can then conditionally apply one of those profile scripts based on a project property of your own choice.

The following example demonstrates the basic technique using a project property called `buildProfile` and profile scripts that simply initialize an extra project property called `message`:

Example 6. Mimicking the behavior of Maven profiles in Gradle

build.gradle

```
if (!hasProperty('buildProfile')) ext.buildProfile = 'default'  ①

apply from: "profile-${buildProfile}.gradle"  ②

tasks.register('greeting') {
    doLast {
        println message  ③
    }
}
```

profile-default.gradle

```
ext.message = 'foobar'  ④
```

profile-test.gradle

```
ext.message = 'testing 1 2 3'  ④
```

profile-prod.gradle

```
ext.message = 'Hello, world!'  ④
```

build.gradle.kts

```
val buildProfile: String? by project  ①

apply(from = "profile-${buildProfile ?: "default"}.gradle.kts")  ②

tasks.register("greeting") {
    val message: String by project.extra
    doLast {
        println(message)  ③
    }
}
```

profile-default.gradle.kts

```
val message by extra("foobar")  ④
```

profile-test.gradle.kts

```
val message by extra("testing 1 2 3")  ④
```

profile-prod.gradle.kts

```
val message by extra("Hello, world!")  ④
```

① Checks for the existence of (Groovy) or binds (Kotlin) the `buildProfile` project property

② Applies the appropriate profile script, using the value of `buildProfile` in the script filename

③ Prints out the value of the `message` extra project property

④ Initializes the `message` extra project property, whose value can then be used in the main build script

With this setup in place, you can activate one of the profiles by passing a value for the project property you're using — `buildProfile` in this case:

Output of `gradle greeting`

```
> gradle greeting
foobar
```

Output of `gradle -PbuildProfile=test greeting`

```
> gradle -PbuildProfile=test greeting
testing 1 2 3
```

You're not limited to checking project properties. You could also check environment variables, the JDK version, the OS the build is running on, or anything else you can imagine.

One thing to bear in mind is that high level condition statements make builds harder to understand and maintain, similar to the way they complicate object-oriented code. The same applies to profiles. Gradle offers you many better ways to avoid the extensive use of profiles that Maven often requires, for example by configuring multiple tasks that are variants of one another. See the `publishPubNamePublicationToRepoNameRepository` tasks created by the Maven Publish Plugin.

For a lengthier discussion on working with Maven profiles in Gradle, look no further than this blog post.

Filtering resources

Maven has a phase called `process-resources` that has the goal `resources:resources` bound to it by default. This gives the build author an opportunity to perform variable substitution on various files, such as web resources, packaged properties files, etc.

The Java plugin for Gradle provides a `processResources` task to do the same thing. This is a ProcessResources task that copies files from the configured resources directory — `src/main/resources` by default — to an output directory. And as with any `ProcessResources` or `Copy` task, you can configure it to perform file filtering, renaming, and content filtering.

As an example, here's a configuration that treats the source files as Groovy `SimpleTemplateEngine` templates, providing `version` and `buildNumber` properties to those templates:

Example 7. Filtering the content of resources via the `processResources` task

build.gradle

```
processResources {
    expand(version: version, buildNumber: currentBuildNumber)
}
```

build.gradle.kts

```
tasks {
    processResources {
        expand("version" to version, "buildNumber" to currentBuildNumber)
    }
}
```

See the API docs for CopySpec to see all the options available to you.

Configuring integration tests

Many Maven builds incorporate integration tests of some sort, which Maven supports through an extra set of phases: `pre-integration-test`, `integration-test`, `post-integration-test`, and `verify`. It also uses the Failsafe plugin in place of Surefire so that failed integration tests don't automatically fail the build (because you may need to clean up resources, such as a running application server).

This behavior is easy to replicate in Gradle with source sets, as explained in our chapter on Testing in Java & JVM projects. You can then configure a clean-up task, such as one that shuts down a test server for example, to always run after the integration tests regardless of whether they succeed or fail using Task.finalizedBy().

If you really don't want your integration tests to fail the build, then you can use the Test.ignoreFailures setting described in the Test execution section of the Java testing chapter.

Source sets also give you a lot of flexibility on where you place the source files for your integration tests. You can easily keep them in the same directory as the unit tests or, more preferably, in a separate source directory like `src/integTest/java`. To support other types of tests, you just add more source sets and Test tasks!

Migrating common plugins

Maven and Gradle share a common approach of extending the build through plugins. Although the plugin systems are very different beneath the surface, they share many feature-based plugins, such as:

- Shade/Shadow
- Jetty
- Checkstyle
- JaCoCo
- AntRun (see further down)

Why does this matter? Because many plugins rely on standard Java conventions, so migration is just a matter of replicating the configuration of the Maven plugin in Gradle. As an example, here's a simple Maven Checkstyle plugin configuration:

```
...
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-checkstyle-plugin</artifactId>
  <version>2.17</version>
  <executions>
    <execution>
      <id>validate</id>
      <phase>validate</phase>
      <configuration>
        <configLocation>checkstyle.xml</configLocation>
        <encoding>UTF-8</encoding>
        <consoleOutput>true</consoleOutput>
        <failsOnError>true</failsOnError>
        <linkXRef>false</linkXRef>
      </configuration>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
  </executions>
</plugin>
...
```

Everything outside of the configuration block can safely be ignored when migrating to Gradle. In this case, the corresponding Gradle configuration looks like the following:

Example 8. Configuring the Gradle Checkstyle Plugin

build.gradle

```
checkstyle {
    config = resources.text.fromFile('checkstyle.xml', 'UTF-8')
    showViolations = true
    ignoreFailures = false
}
```

build.gradle.kts

```
checkstyle {
    config = resources.text.fromFile("checkstyle.xml", "UTF-8")
    isShowViolations = true
    isIgnoreFailures = false
}
```

The Checkstyle tasks are automatically added as dependencies of the `check` task, which also includes `test`. If you want to ensure that Checkstyle runs before the tests, then just specify an ordering with the mustRunAfter() method:

Example 9. Controlling when the `checkstyle` task runs

build.gradle

```
test.mustRunAfter checkstyleMain, checkstyleTest
```

build.gradle.kts

```
tasks {
    test {
        mustRunAfter(checkstyleMain, checkstyleTest)
    }
}
```

As you can see, the Gradle configuration is often much shorter than the Maven equivalent. You also have a much more flexible execution model since you are no longer constrained by Maven's fixed phases.

While migrating a project from Maven, don't forget about source sets. These often provide a more elegant solution for handling integration tests or generated sources than Maven can provide, so you should factor them into your migration plans.

Ant goals

Many Maven builds rely on the AntRun plugin to customize the build without the overhead of implementing a custom Maven plugin. Gradle has no equivalent plugin because Ant is a first-class citizen in Gradle builds, via the `ant` object. For example, you can use Ant's Echo task like this:

Example 10. Invoking Ant tasks

build.gradle

```
tasks.register('sayHello') {
    doLast {
        ant.echo message: 'Hello!'
    }
}
```

build.gradle.kts

```
tasks.register("sayHello") {
    doLast {
        ant.withGroovyBuilder {
            "echo"("message" to "Hello!")
        }
    }
}
```

Even Ant properties and filesets are supported natively. To learn more, see Using Ant from Gradle.

TIP	It may be simpler and cleaner to just create custom task types to replace the work that Ant is doing for you. You can then more readily benefit from incremental build and other useful Gradle features.

Understanding which plugins you don't need

It's worth remembering that Gradle builds are typically easier to extend and customize than Maven ones. In this context, that means you may not need a Gradle plugin to replace a Maven one. For example, the Maven Enforcer plugin allows you to control dependency versions and environmental factors, but these things can easily be configured in a normal Gradle build script.

Dealing with uncommon and custom plugins

You may come across Maven plugins that have no counterpart in Gradle, particularly if you or someone in your organisation has written a custom plugin. Such cases rely on you understanding how Gradle (and potentially Maven) works, because you will usually have to write your own plugin.

For the purposes of migration, there are two key types of Maven plugins:

- Those that use the Maven project object.
- Those that don't.

Why is this important? Because if you use one of the latter, you can trivially reimplement it as a custom Gradle task type. Simply define task inputs and outputs that correspond to the mojo parameters and convert the execution logic into a task action.

If a plugin depends on the Maven project, then you will have to rewrite it. Don't start by considering how the Maven plugin works, but look at what problem it is trying to solve. Then try to work out how to solve that problem in Gradle. You'll probably find that the two build models are different enough that "transcribing" Maven plugin code into a Gradle plugin just won't be effective. On the plus side, the plugin is likely to be much easier to write than the original Maven one because Gradle has a much richer build model and API.

If you do need to implement custom logic, either via build scripts or plugins, check out the Guides related to plugin development. Also be sure to familiarize yourself with Gradle's Groovy DSL Reference, which provides comprehensive documentation on the API that you'll be working with. It details the standard configuration blocks (and the objects that back them), the core types in the system (`Project`, `Task`, etc.), and the standard set of task types. The main entry point is the Project interface as that's the top-level object that backs the build scripts.

Further reading

This chapter has covered the major topics that are specific to migrating Maven builds to Gradle. All that remain are a few other areas that may be useful during or after a migration:

- Learn how to configure Gradle's build environment, including the JVM settings used to run it
- Learn how to structure your builds effectively
- Configure Gradle's logging and use it from your builds

As a final note, this guide has only touched on a few of Gradle's features and we encourage you to learn about the rest from the other chapters of the user manual and from our step-by-step samples.

Migrating Builds From Apache Ant

Apache Ant is a build tool with a long history in the Java world that is still widely used, albeit by a decreasing number of teams. While flexible, it lacks conventions and many of the powerful features that Gradle can provide. Migrating to Gradle is worthwhile so that your builds can become slimmer, simpler and faster, while still retaining the flexibility you enjoy with Ant. You'll also benefit from robust support for multi-project builds and easy-to-use, flexible dependency management.

The biggest challenge in migrating from Ant to Gradle is that there is no such thing as a standard Ant build. That makes it difficult to provide specific instructions. Fortunately, Gradle has some great integration features with Ant that can make the process relatively smooth. And even migrating from Ivy-based dependency management isn't particularly hard because Gradle has a similar model based on dependency configurations that works with Ivy-compatible repositories.

We will start by outlining the things you should consider at the outset of migrating a build from Ant to Gradle and offer some general guidelines on how to proceed.

General guidelines

When you undertake to migrate a build from Ant to Gradle, you should keep in mind the nature of both what you already have and where you would like to end up. Do you want a Gradle build that mirrors the structure of the existing Ant build? Or do you want to move to something that is more idiomatic to Gradle? What are the main benefits you are looking for?

To understand the implications, consider the two extreme endpoints that you could aim for:

- An imported build via `ant.importBuild()`

  This approach is quick, simple and works for many Ant-based builds. You end up with a build that's effectively identical to the original Ant build, except your Ant targets become Gradle tasks. Even the dependencies between targets are retained.

  The downside is that you're still using the Ant build, which you must continue to maintain. You also lose the advantages of Gradle's conventions, many of its plugins, its dependency management, and so on. You can still enhance the build with incremental build information, but it's more effort than would be the case for a normal Gradle build.

- An idiomatic Gradle build

  If you want to future proof your build, this is where you want to end up. Making use of Gradle's conventions and plugins will result in a smaller, easier-to-maintain build, with a structure that is familiar to many Java developers. You will also find it easier to take advantage of Gradle's power features to improve build performance.

  The main downside is the extra work required to perform the migration, particularly if the existing build is complex and has many inter-project dependencies. But such builds often benefit the most from a switch to idomatic Gradle. In addition, Gradle provides many features that can ease the migration, such as the ability to use core and custom Ant tasks directly from a Gradle build.

You ideally want to end up somewhere close to the second option in the long term, but you don't have to get there in one fell swoop.

What follows is a series of steps to help you decide the approach you want to take and how to go about it:

1. Keep the old Ant build and new Gradle build side by side

   You know the Ant build works, so you should keep it until you are confident that the Gradle build produces all the same artifacts and otherwise does what you need. This also means that users can try the Gradle build without getting a new copy of the source tree.

   Don't try to change the directory and file structure of the build until after you're ready to make the switch.

2. Develop a mechanism to verify that the two builds produce the same artifacts

   This is a vitally important step to ensure that your deployments and tests don't break. Even

small changes, such as the contents of a manifest file in a JAR, can cause problems. If your Gradle build produces the same output as the Ant build, this will give you and others confidence in switching over and make it easier to implement the big changes that will provide the greatest benefits.

3. Decide whether you have a multi-project build or not

   Multi-project builds are generally harder to migrate and require more work than single-project ones. We have provided some dedicated advice to help with the process in the Migrating multi-project builds section.

4. Work out what plugins to use for each project

   We expect that the vast majority of Ant builds are for JVM-based projects, for which there are a wealth of plugins that provide a lot of the functionality you need. Not only are there the core plugins that come packaged with Gradle, but you can also find many useful plugins on the Plugin Portal.

   Even if the Java Plugin or one of its derivatives (such as the Java Library Plugin) aren't a good match for your build, you should at least consider the Base Plugin for its lifecycle tasks.

5. Import the Ant build or create a Gradle build from scratch

   This step very much depends on the requirements of your build. If a selection of Gradle plugins can do the vast majority of the work your Ant build does, then it probably makes sense to create a fresh Gradle build script that doesn't depend on the Ant build and either implements the missing pieces itself or utilizes existing Ant tasks.

   The alternative approach is to import the Ant build into the Gradle build script and gradually replace the Ant build functionality. This allows you to have a working Gradle build at each stage, but it requires a bit of work to get the Gradle tasks working properly with the Ant ones. You can learn more about this approach in Working with an imported build.

6. Configure your build for the existing directory and file structure

   Gradle makes use of conventions to eliminate much of the boilerplate associated with older builds and to make it easier for users to work with new builds once they are familiar with those conventions. But that doesn't mean you have to follow them.

   Gradle provides many configuration options that allow for a good degree of customization. Those options are typically made available through the plugins that provide the conventions. For example, the standard source directory structure for production Java code — `src/main/java` — is provided by the Java Plugin, which allows you to configure a different source path. Many paths can be modified via properties on the Project object.

7. Migrate to standard Gradle conventions if you wish

   Once you're confident that the Gradle build is producing the same artifacts and other resources as the Ant build, you can consider migrating to the standard conventions, such as for source directory paths. Doing so will allow you to remove the extra configuration that was required to override those conventions. New team members will also find it easier to work with the build

after the change.

It's up to you to decide whether this step is worth the time, energy and potential disruption that it might incur, which in turn depends on your specific build and team.

The rest of the chapter covers some common scenarios you will likely deal with during the migration, such as dependency management and working with Ant tasks.

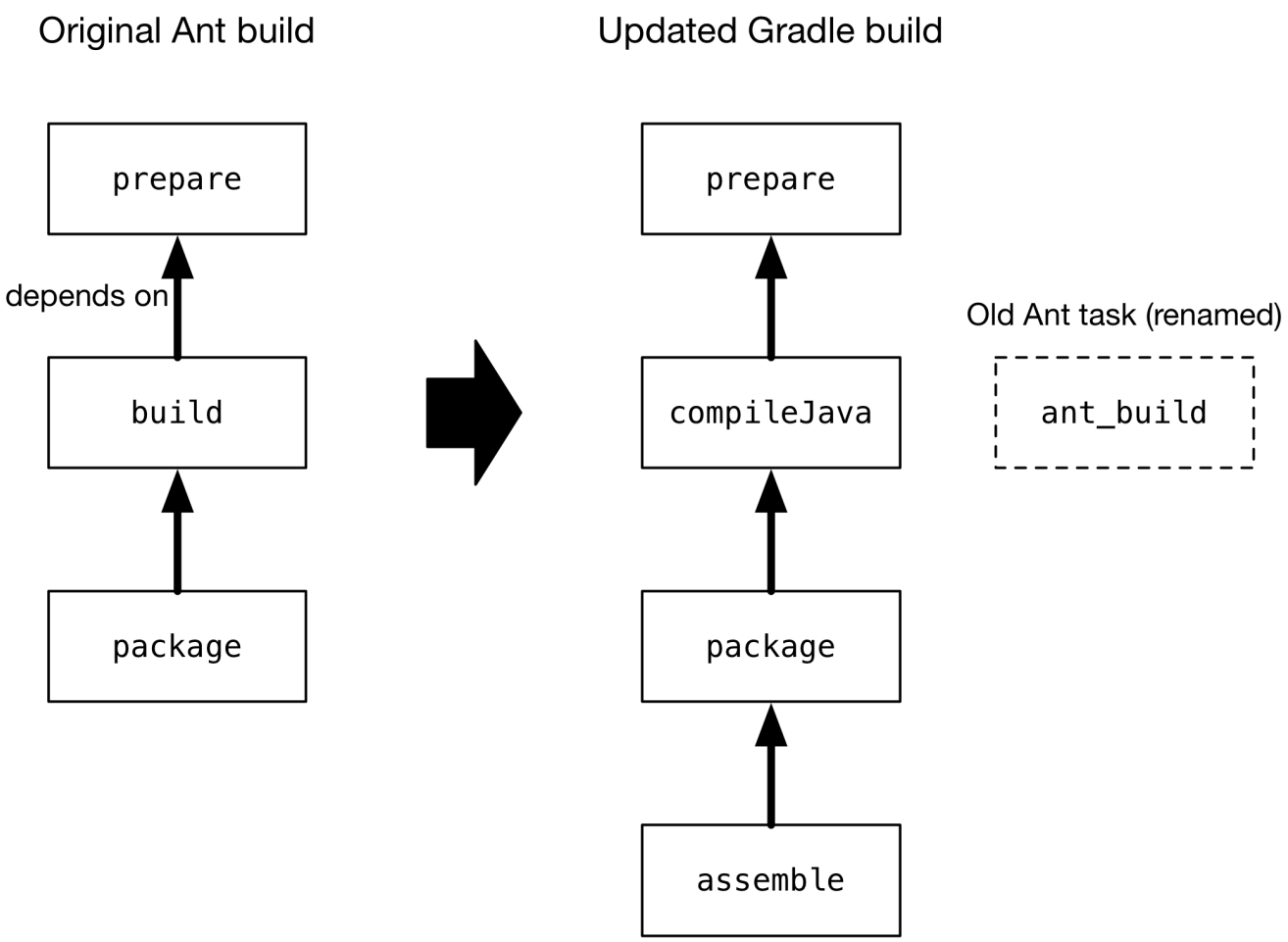
Working with an imported build

The first step of many migrations will involve importing an Ant build using `ant.importBuild()`. If you do that, how do you then move towards a standard Gradle build without replacing everything at once?

The important thing to remember is that the Ant targets become real Gradle tasks, meaning you can do things like modify their task dependencies, attach extra task actions, and so on. This allows you to substitute native Gradle tasks for the equivalent Ant ones, maintaining any links to other existing tasks.

As an example, imagine that you have a Java library project that you want to migrate from Ant to Gradle. The Gradle build script has the line that imports the Ant build and now want to use the standard Gradle mechanism for compiling the Java source files. However, you want to keep using the existing `package` task that creates the library's JAR file.

In diagrammatic form, the scenario looks like the following, where each box represents a target/task:



The idea is to substitute the standard Gradle `compileJava` task for the Ant `build` task. There are several steps involved in this substitution:

1. Applying the Java Library Plugin

   This provides the `compileJava` task shown in the diagram.

2. Renaming the old `build` task

   The name `build` conflicts with the standard `build` task provided by the Base Plugin (via the Java Library Plugin).

3. Configuring the compilation to use the existing directory structure

   There's a good chance the Ant build does not conform to the standard Gradle directory structure, so you need to tell Gradle where to find the source files and where to place the compiled classes so `package` can find them.

4. Updating task dependencies

   `compileJava` must depend on `prepare`, `package` must depend on `compileJava` rather than `ant_build`, and `assemble` must depend on `package` rather than the standard Gradle `jar` task.

Applying the plugin is as simple as inserting a `plugins {}` block at the beginning of the Gradle build script, i.e. before `ant.importBuild()`. Here's how to apply the Java Library Plugin:

Example 11. Applying the Java Library Plugin

build.gradle

```
plugins {
    id 'java-library'
}
```

build.gradle.kts

```
plugins {
    `java-library`
}
```

To rename the `build` task, use the variant of AntBuilder.importBuild() that accepts a transformer, like this:

Example 12. Renaming targets on import

build.gradle

```
ant.importBuild('build.xml') { String oldTargetName ->
    return oldTargetName == 'build' ? 'ant_build' : oldTargetName  ①
}
```

build.gradle.kts

```
ant.importBuild("build.xml") { oldTargetName ->
    if (oldTargetName == "build") "ant_build" else oldTargetName  ①
}
```

① Renames the `build` target to `ant_build` and leaves all other targets unchanged

Configuring a different path for the sources is described in the Building Java & JVM projects chapter, while you can change the output directory for the compiled classes in a similar way.

Let's say the original Ant build stores these paths in Ant properties, `src.dir` for the Java source files and `classes.dir` for the output. Here's how you would configure Gradle to use those paths:

Example 13. Configuring the source sets

build.gradle

```
sourceSets {
    main {
        java {
            srcDirs = [ ant.properties['src.dir'] ]
            destinationDirectory.set(file(ant.properties['classes.dir']))
        }
    }
}
```

build.gradle.kts

```
sourceSets {
    main {
        java.setSrcDirs(listOf(ant.properties["src.dir"]))
        java.destinationDirectory.set(file(ant.properties["classes.dir"] ?:
"$buildDir/classes"))
    }
}
```

You should eventually aim to switch the standard directory structure for your type of project if possible and then you'll be able to remove this customization.

The last step is also straightforward and involves using the Task.dependsOn property and Task.dependsOn() method to detach and link tasks. The property is appropriate for *replacing* dependencies, while the method is the preferred way to add to the existing dependencies.

Here is the required task dependency configuration required by the example scenario, which should come after the Ant build import:

Example 14. Configuring the task dependencies

build.gradle

```
compileJava.dependsOn 'prepare'  ①
tasks.named('package') { dependsOn = [ 'compileJava' ] }  ②
assemble.dependsOn = [ 'package' ]  ③
```

build.gradle.kts

```
tasks {
    compileJava {
        dependsOn("prepare")  ①
    }
    named("package") {
        setDependsOn(listOf(compileJava))  ②
    }
    assemble {
        setDependsOn(listOf("package"))  ③
    }
}
```

① Makes compilation depend on the `prepare` task

② Detaches `package` from the `ant_build` task and makes it depend on `compileJava`

③ Detaches `assemble` from the standard Gradle `jar` task and makes it depend on `package` instead

That's it! These four steps will successfully replace the old Ant compilation with the Gradle implementation. Even this small migration will be a big help because you'll be able to take advantage of Gradle's incremental Java compilation for faster builds.

TIP This is just a demonstration of how to go about performing a migration in stages. It may make more sense to include resource processing — like with properties files — and packaging with the compilation in this stage, since all three aspects are well integrated in Gradle.

One important question you will have to ask yourself is how many tasks to migrate in each stage. The larger the chunks you can migrate in one go the better, but this must be offset against how many custom steps within the Ant build will be affected by the changes.

For example, if the Ant build follows a fairly standard approach for compilation, static resources, packaging and unit tests, then it is probably worth migrating all those together. But if the build performs some extra processing on the compiled classes, or does something unique when processing the static resources, it is probably worth splitting those tasks into separate stages.

Managing dependencies

Ant builds typically take one of two approaches to dealing with binary dependencies (such as libraries):

- Storing them with the project in a local "lib" directory
- Using Apache Ivy to manage them

They each require a different technique for the migration to Gradle, but you will find the process straightforward in either case. We look at the details of each scenario in the following sections.

Serving dependencies from a directory

When you are attempting to migrate a build that stores its dependencies on the filesystem, either locally or on the network, you should consider whether you want to eventually move to managed dependencies using remote repositories. That's because you can incorporate filesystem dependencies into a Gradle build in one of two ways:

- Define a flat-directory repository and use standard dependency declarations
- Attach the files directly to the appropriate dependency configurations (file dependencies)

It's easier to migrate to managed dependencies served from Maven- or Ivy-compatible repositories if you take the first approach, but doing so requires all your files to conform to the naming convention "<moduleName>-<version>.<extension>".

NOTE	If you store your dependencies in the standard Maven repository layout — `<repoDir>/<group>/<module>/<version>` — then you can define a custom Maven repository with a "file://" URL.

To demonstrate the two techniques, consider a project that has the following library JARs in its `libs` directory:

```
libs
 ├─── our-custom.jar
 ├─── log4j-1.2.8.jar
 └─── commons-io-2.1.jar
```

The file `our-custom.jar` lacks a version number, so it has to be added as a file dependency. But the other two JARs match the required naming convention and so can be declared as normal module dependencies that are retrieved from a flat-directory repository.

The following sample build script demonstrates how you can incorporate all of these libraries into a build:

Example 15. Declaring dependencies served from the filesystem

build.gradle

```
repositories {
    flatDir {
        name = 'libs dir'
        dir file('libs')  ①
    }
}

dependencies {
    implementation files('libs/our-custom.jar')  ②
    implementation ':log4j:1.2.8', ':commons-io:2.1'  ③
}
```

build.gradle.kts

```
repositories {
    flatDir {
        name = "libs dir"
        dir(file("libs"))  ①
    }
}

dependencies {
    implementation(files("libs/our-custom.jar"))  ②
    implementation(":log4j:1.2.8")     ③
    implementation(":commons-io:2.1")  ③
}
```

① Specifies the path to the directory containing the JAR files

② Declares a *file dependency* for the unversioned JAR

③ Declares dependencies using standard dependency coordinates — note that no group is specified, but each identifier has a leading `:`, implying an empty group

The above sample will add `our-custom.jar`, `log4j-1.2.8.jar` and `commons-io-2.1.jar` to the `implementation` configuration, which is used to compile the project's code.

NOTE

> You can also specify a group in these module dependencies, even though they don't actually have a group. That's because the flat-directory repository simply ignores the information.
>
> If you then add a normal Maven- or Ivy-compatible repository at a later date, Gradle will preferentially download the module dependencies that are declared with a group from that repository rather than the flat-directory one.

Migrating Ivy dependencies

Apache Ivy is a standalone dependency management tool that is widely used with Ant. It works in a similar fashion to Gradle. In fact, they both allow you to

- Define your own configurations
- Extend configurations from one another
- Attach dependencies to configurations
- Resolve dependencies from Ivy-compatible repositories
- Publish artifacts to Ivy-compatible repositories

The most notable difference is that Gradle has standard configurations for specific types of projects. For example, the Java Plugin defines configurations like `implementation`, `testImplementation` and `runtimeOnly`. You can still define your own dependency configurations, though.

This similarity means that it's usually quite straightforward to migrate from Ivy to Gradle:

- Transcribe the dependency declarations from your module descriptors into the dependencies {} block of your Gradle build script, ideally using the standard configurations provided by any plugins you apply.
- Transcribe any configuration declarations from your module descriptors into the configurations {} block of the build script for any custom configurations that can't be replaced by Gradle's standard ones.
- Transcribe the resolvers from your Ivy settings file into the repositories {} block of the build script.

See the chapters on Managing Dependency Configurations, Declaring Dependencies and Declaring Repositories for more information.

Ivy provides several Ant tasks that handle Ivy's process for fetching dependencies. The basic steps of that process consist of:

1. *Configure* — applies the configuration defined in the Ivy settings file
2. *Resolve* — locates the declared dependencies and downloads them to the cache if necessary
3. *Retrieve* — copies the cached dependencies to another directory

Gradle's process is similar, but you don't have to explicitly invoke the first two steps as it performs them automatically. The third step doesn't happen at all — unless you create a task to do it — because Gradle typically uses the files in the dependency cache directly in classpaths and as the

source for assembling application packages.

Let's look in more detail at how Ivy's steps map to Gradle:

Configuration

Most of Gradle's dependency-related configuration is baked into the build script, as you've seen with elements like the `dependencies {}` block. Another particularly important configuration element is resolutionStrategy, which can be accessed from dependency configurations. This provides many of the features you might get from Ivy's conflict managers and is a powerful way to control transitive dependencies and caching.

Some Ivy configuration options have no equivalent in Gradle. For example, there are no lock strategies because Gradle ensures that its dependency cache is concurrency safe, period. Nor are there "latest strategies" because it's simpler to have a reliable, single strategy for conflict resolution. If the "wrong" version is picked, you can easily override it using forced versions or other resolution strategy options.

See the chapter on controlling transitive dependencies for more information on this aspect of Gradle.

Resolution

At the beginning of the build, Gradle will automatically resolve any dependencies that you have declared and download them to its cache. It searches the repositories for those dependencies, with the search order defined by the order in which the repositories are declared.

It's worth noting that Gradle supports the same dynamic version syntax as Ivy, so you can still use versions like `1.0.+`. You can also use the special `latest.integration` and `latest.release` labels if you wish. If you decide to use such dynamic and changing dependencies, you can configure the caching behavior for them via resolutionStrategy.

You might also want to consider dependency locking if you're using dynamic and/or changing dependencies. It's a way to make the build more reliable and allows for reproducible builds.

Retrieval

As mentioned, Gradle does not automatically copy files from the dependency cache. Its standard tasks typically use the files directly. If you want to copy the dependencies to a local directory, you can use a Copy task like this in your build script:

Example 16. Copying dependencies to a local directory

build.gradle

```
tasks.register('retrieveRuntimeDependencies', Copy) {
    into layout.buildDirectory.dir('libs')
    from configurations.runtimeClasspath
}
```

build.gradle.kts

```
tasks.register<Copy>("retrieveRuntimeDependencies") {
    into(layout.buildDirectory.dir("libs"))
    from(configurations.runtimeClasspath)
}
```

A configuration is also a file collection, hence why it can be used in the `from()` configuration. You can use a similar technique to attach a configuration to a compilation task or one that produces documentation. See the chapter on Working with Files for more examples and information on Gradle's file API.

Publishing artifacts

Projects that use Ivy to manage dependencies often also use it for publishing JARs and other artifacts to repositories. If you're migrating such a build, then you'll be glad to know that Gradle has built-in support for publishing artifacts to Ivy-compatible repositories.

Before you attempt to migrate this particular aspect of your build, read the Publishing chapter to learn about Gradle's publishing model. That chapter's examples are based on Maven repositories, but the same model is used for Ivy repositories as well.

The basic migration process looks like this:

- Apply the Ivy Publish Plugin to your build
- Configure at least one publication, representing what will be published (including additional artifacts if desired)
- Configure one or more repositories to publish artifacts to

Once that's all done, you'll be able to generate an Ivy module descriptor for each publication and publish them to one or more repositories.

Let's say you have defined a publication named "myLibrary" and a repository named "myRepo". Ivy's Ant tasks would then map to the Gradle tasks like this:

- `<deliver>` → `generateDescriptorFileForMyLibraryPublication`
- `<publish>` → `publishMyLibraryPublicationToMyRepoRepository`

There is also a convenient `publish` task that publishes *all* publications to *all* repositories. If you'd prefer to limit which publications go to which repositories, check out the relevant section of the Publishing chapter.

NOTE

On dependency versions

Ivy will, by default, automatically replace dynamic versions of dependencies with the resolved "static" versions when it generates the module descriptor. Gradle does *not* mimic this behavior: declared dependency versions are left unchanged.

You can replicate the default Ivy behavior by using the Nebula Ivy Resolved Plugin. Alternatively, you can customize the descriptor file so that it contains the versions you want.

Dealing with custom Ant tasks

One of the advantages of Ant is that it's fairly easy to create a custom task and incorporate it into a build. If you have such tasks, then there are two main options for migrating them to a Gradle build:

- Using the custom Ant task from the Gradle build
- Rewriting the task as a custom Gradle task type

The first option is usually quick and easy, but not always. And if you want to integrate the task into incremental build, you must use the incremental build runtime API. You also often have to work with Ant paths and filesets, which are clunky.

The second option is preferable in the long term, if you have the time. Gradle task types tend to be simpler than Ant tasks because they don't have to work with an XML-based interface. You also gain access to Gradle's rich APIs. Lastly, this approach can make use of the type-safe incremental build API based on typed properties.

Working with files

Ant has many tasks for working with files, most of which have Gradle equivalents. As with other areas of Ant to Gradle migration, you can use those Ant tasks from within your Gradle build. However, we strongly recommend migrating to native Gradle constructs where possible so that the build benefits from:

- Incremental build
- Easier integration with other parts of the build, such as dependency configurations
- More idiomatic build scripts

That said, it can be convenient to use those Ant tasks that have no direct equivalents, such as `<checksum>` and `<chown>`. Even then, in the long run it may be better to convert these to native Gradle task types that make use of standard Java APIs or third-party libraries to achieve the same thing.

Here are the most common file-related elements used by Ant builds, along with the Gradle equivalents:

- `<copy>` — prefer the Gradle Copy task type
- `<zip>` (plus Java variants) — prefer the Zip task type (plus Jar, War, and Ear)
- `<unzip>` — prefer using the Project.zipTree() method with a `Copy` task

You can see several examples of Gradle's file API and learn more about it in the Working with Files chapter.

NOTE

On paths and filesets

Ant makes use of the concepts of path-like structures and filesets to enable users to work with collections of files and directories. Gradle has a simpler, more powerful model based on FileCollections and FileTrees that can be treated as objects from within the build. Both types allow filtering based on Ant's glob syntax, e.g. `**/books_*`. Learn more about these types and other aspects of Gradle's file API in the Working with Files chapter.

You can still construct Ant paths and filesets from within your build via the `ant` object if you need to interact with an Ant task that requires them. The chapter on Ant integration has examples that use both `<path>` and `<fileset>`. There is even a method on `FileCollection` that will convert a file collection to a fileset or similar Ant type.

Migrating Ant properties

Ant makes use of a properties map to store values that can be reused throughout the build. The big downsides to this approach are that property values are all strings and the properties themselves behave like global variables.

TIP

Interacting with Ant properties in Gradle

Sometimes you will want to make use of an Ant task directly from your Gradle build and that task requires one or more Ant properties to be set. If that's the case, you can easily set those properties via the `ant` object, as described in the Using Ant from Gradle chapter.

Gradle does use something similar in the form of project properties, which are a reasonable way to parameterize a build. These can be set from the command line, in a `gradle.properties` file, or even via specially named system properties and environment variables.

If you have existing Ant properties files, you can copy their contents into the project's `gradle.properties` file. Just be aware of two important points:

- Properties set in `gradle.properties` **do not** override extra project properties defined in the build script with the same name
- Imported Ant tasks will not automatically "see" the Gradle project properties — you must copy them into the Ant properties map for that to happen

Another important factor to understand is that a Gradle build script works with an object-oriented API and it's often best to use the properties of tasks, source sets and other objects where possible. For example, this build script fragment creates tasks for packaging Javadoc documentation as a JAR and unpacking it, linking tasks via their properties:

Example 17. Using task properties in place of project properties

build.gradle

```
def tmpDistDir = layout.buildDirectory.dir('dist')

tasks.register('javadocJarArchive', Jar) {
    from javadoc  ①
    archiveClassifier = 'javadoc'
}

tasks.register('unpackJavadocs', Copy) {
    from zipTree(javadocJarArchive.archiveFile)  ②
    into tmpDistDir  ③
}
```

build.gradle.kts

```
val tmpDistDir = layout.buildDirectory.dir("dist")

tasks.register<Jar>("javadocJarArchive") {
    from(tasks.javadoc)  ①
    archiveClassifier.set("javadoc")
}

tasks.register<Copy>("unpackJavadocs") {
    from(zipTree(tasks.named<Jar>("javadocJarArchive").get().archiveFile))
②
    into(tmpDistDir)  ③
}
```

① Packages all `javadoc`'s output files — equivalent to `from javadoc.destinationDir`

② Uses the location of the Javadoc JAR held by the `javadocJar` task

③ Uses an project property called `tmpDistDir` to define the location of the 'dist' directory

As you can see from the example with `tmpDistDir`, there is often still a need to define paths and the like through properties, which is why Gradle also provides extra properties that can be attached to the project, tasks and some other types of objects.

Migrating multi-project builds

Multi-project builds are a particular challenge to migrate because there is no standard approach in Ant for either structuring them or handling inter-project dependencies. Most of them likely use the `<ant>` task in some way, but that's about all that one can say.

Fortunately, Gradle's multi-project support can handle fairly diverse project structures and it provides much more robust and helpful support than Ant for constructing and maintaining multi-project builds. The `ant.importBuild()` method also handles `<ant>` and `<antcall>` tasks transparently, which allows for a phased migration.

We will suggest one process for migration here and hope that it either works for your case or at least gives you some ideas. It breaks down like this:

1. Start by learning how Gradle configures multi-project builds.
2. Create a Gradle build script in each project of the build, setting their contents to this line:

   ```
   ant.importBuild 'build.xml'
   ```

   ```
   ant.importBuild("build.xml")
   ```

   Replace `build.xml` with the path to the actual Ant build file that corresponds to the project. If there is no corresponding Ant build file, leave the Gradle build script empty. Your build may not be suitable in that case for this migration approach, but continue with these steps to see whether there is still a way to do a phased migration.

3. Create a settings file that includes all the projects that now have a Gradle build script.
4. Implement inter-project dependencies.

   Some projects in your multi-project build will depend on artifacts produced by one or more other projects in that build. Such projects need to ensure that those projects they depend on have produced their artifacts and that they know the paths to those artifacts.

   Ensuring the production of the required artifacts typically means calling into other projects' builds via the `<ant>` task. This unfortunately bypasses the Gradle build, negating any changes you make to the Gradle build scripts. You will need to replace targets that use `<ant>` tasks with Gradle task dependencies.

   For example, imagine you have a web project that depends on a "util" library that's part of the same build. The Ant build file for "web" might have a target like this:

web/build.xml

```
<target name="buildRequiredProjects">
    <ant dir="${root.dir}/util" target="build"/>  ①
</target>
```

① `root.dir` would have to be defined by the build

This can be replaced by an inter-project task dependency in the corresponding Gradle build script, as demonstrated in the following example that assumes the "web" project's "compile" task is the thing that requires "util" to be built beforehand:

web/build.gradle

```
ant.importBuild 'build.xml'

compile.dependsOn = [ ':util:build' ]
```

web/build.gradle.kts

```
ant.importBuild("build.xml")

tasks {
    named<Task>("compile") {
        setDependsOn(listOf(":util:build"))
    }
}
```

This is not as robust or powerful as Gradle's project dependencies, but it solves the immediate problem without big changes to the build. Just be careful to remove or override any dependencies on tasks that delegate to other subprojects, like the `buildRequiredProjects` task.

5. Identify the projects that have no dependencies on other projects and migrate them to idiomatic Gradle builds scripts.

   Just follow the advice in the rest of this guide to migrate individual project builds. As mentioned elsewhere, you should ideally use Gradle standard plugins where possible. This may mean that you need to add an extra copy task to each build that copies the generated artifacts to the location expected by the rest of the Ant builds.

6. Migrate projects as and when they depend solely on projects with fully migrated Gradle builds.

At this point, you should be able to switch to using proper project dependencies attached to the appropriate dependency configurations.

7. Clean up projects once no part of the Ant build depends on them.

   We mentioned in step 5 that you might need to add copy tasks to satisfy the requirements of dependent Ant builds. Once those builds have been migrated, such build logic will no longer be needed and should be removed.

At the end of the process you should have a Gradle build that you are confident works as it should, with much less build logic than before.

Further reading

This chapter has covered the major topics that are specific to migrating Ant builds to Gradle. All that remain are a few other areas that may be useful during or after a migration:

- Learn how to configure Gradle's build environment, including the JVM settings used to run it
- Learn how to structure your builds effectively
- Configure Gradle's logging and use it from your builds

As a final note, this guide has only touched on a few of Gradle's features and we encourage you to learn about the rest from the other chapters of the user manual and from our step-by-step samples.

Running Gradle Builds

Build Environment

TIP	Interested in configuring your Build Cache to speed up builds? Register here for our Build Cache training session to learn some of the tips and tricks top engineering teams are using to increase build speed.

Gradle provides multiple mechanisms for configuring behavior of Gradle itself and specific projects. The following is a reference for using these mechanisms.

When configuring Gradle behavior you can use these methods, listed in order of highest to lowest precedence (first one wins):

- Command-line flags such as `--build-cache`. These have precedence over properties and environment variables.
- System properties such as `systemProp.http.proxyHost=somehost.org` stored in a `gradle.properties` file in a root project directory.
- Gradle properties such as `org.gradle.caching=true` that are typically stored in a `gradle.properties` file in a project directory or in the `GRADLE_USER_HOME`.
- Environment variables such as `GRADLE_OPTS` sourced by the environment that executes Gradle.

Aside from configuring Gradle behavior you can configure the build using the same mechanisms and reading the environment from the build logic.

Gradle properties

Gradle provides several options that make it easy to configure the Java process that will be used to execute your build. While it's possible to configure these in your local environment via `GRADLE_OPTS` or `JAVA_OPTS`, it is useful to be able to store certain settings like JVM memory configuration and Java home location in version control so that an entire team can work with a consistent environment. To do so, place these settings into a `gradle.properties` file committed to your version control system.

The final configuration taken into account by Gradle is a combination of all Gradle properties set on the command line and your `gradle.properties` files. If an option is configured in multiple locations, the *first one* found in any of these locations wins:

- command line, as set using `-D`.
- `gradle.properties` in `GRADLE_USER_HOME` directory.
- `gradle.properties` in the project's directory, then its parent project's directory up to the build's root directory.
- `gradle.properties` in Gradle installation directory.

Note that the location of the Gradle user home may have been changed beforehand via the `-Dgradle.user.home` system property passed on the command line.

The following properties can be used to configure the Gradle build environment:

`org.gradle.caching=(true,false)`

When set to true, Gradle will reuse task outputs from any previous build, when possible, resulting in much faster builds. Learn more about using the build cache. *By default, the build cache is **not** enabled.*

`org.gradle.caching.debug=(true,false)`

When set to true, individual input property hashes and the build cache key for each task are logged on the console. Learn more about task output caching. *Default is* `false`.

`org.gradle.configureondemand=(true,false)`

Enables incubating configuration on demand, where Gradle will attempt to configure only necessary projects. *Default is* `false`.

`org.gradle.console=(auto,plain,rich,verbose)`

Customize console output coloring or verbosity. *Default depends on how Gradle is invoked. See command-line logging for additional details.*

`org.gradle.continuous.quietperiod=(# of quiet period millis)`

When using continuous build, Gradle will wait for the quiet period to pass before triggering another build. Any additional changes within this quiet period restart waiting for the quiet period. *Default is* `250` *milliseconds.*

`org.gradle.daemon=(true,false)`

When set to `true` the Gradle Daemon is used to run the build. *Default is* `true`, *builds will be run using the daemon.*

`org.gradle.daemon.idletimeout=(# of idle millis)`

Gradle Daemon will terminate itself after specified number of idle milliseconds. *Default is* `10800000` *(3 hours).*

`org.gradle.debug=(true,false)`

When set to `true`, Gradle will run the build with remote debugging enabled, listening on port 5005. Note that this is the equivalent of adding `-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005` to the JVM command line and will suspend the virtual machine until a debugger is attached. *Default is* `false`.

`org.gradle.debug.host=(host address)`

Specifies the host address to listen on or connect to when debug is enabled. In the server mode on Java 9 and above, passing * for the host will make the server listen on all network interfaces. By default, no host address is passed to JDWP, so on Java 9 and above, the loopback address is used, while earlier versions listen on all interfaces.

`org.gradle.debug.port=(port number)`

Specifies the port number to listen on when debug is enabled. *Default is* `5005`.

`org.gradle.debug.server=(true,false)`

If set to `true` and debugging is enabled, Gradle will run the build with the socket-attach mode of the debugger. Otherwise, the socket-listen mode is used. *Default is* `true`.

`org.gradle.debug.suspend=(true,false)`

When set to `true` and debugging is enabled, the JVM running Gradle will suspend until a debugger is attached. *Default is* `true`.

`org.gradle.java.home=(path to JDK home)`

Specifies the Java home for the Gradle build process. The value can be set to either a `jdk` or `jre` location, however, depending on what your build does, using a JDK is safer. This does not affect the version of Java used to launch the Gradle client VM (see Environment variables). *A reasonable default is derived from your environment (*`JAVA_HOME` *or the path to* `java`*) if the setting is unspecified.*

`org.gradle.jvmargs=(JVM arguments)`

Specifies the JVM arguments used for the Gradle Daemon. The setting is particularly useful for configuring JVM memory settings for build performance. This does not affect the JVM settings for the Gradle client VM. *The default is* `-Xmx512m "-XX:MaxMetaspaceSize=256m"`.

`org.gradle.logging.level=(quiet,warn,lifecycle,info,debug)`

When set to quiet, warn, lifecycle, info, or debug, Gradle will use this log level. The values are not case sensitive. See Choosing a log level. *The* `lifecycle` *level is the default.*

`org.gradle.parallel=(true,false)`

When configured, Gradle will fork up to `org.gradle.workers.max` JVMs to execute projects in parallel. To learn more about parallel task execution, see the section on Gradle build performance. *Default is* `false`.

`org.gradle.priority=(low,normal)`

Specifies the scheduling priority for the Gradle daemon and all processes launched by it. See also performance command-line options. *Default is* `normal`.

`org.gradle.vfs.verbose=(true,false)`

Configures verbose logging when watching the file system. *Default is* `false`.

`org.gradle.vfs.watch=(true,false)`

Toggles watching the file system. When enabled Gradle re-uses information it collects about the file system between builds. *Enabled by default on operating systems where Gradle supports this feature.*

`org.gradle.warning.mode=(all,fail,summary,none)`

When set to `all`, `summary` or `none`, Gradle will use different warning type display. See Command-line logging options for details. *Default is* `summary`.

`org.gradle.workers.max=(max # of worker processes)`

When configured, Gradle will use a maximum of the given number of workers. See also performance command-line options. *Default is number of CPU processors.*

`org.gradle.logging.stacktrace=(internal,all,full)`

Specifies whether stacktraces should be displayed as part of the build result upon an exception. See also the --stacktrace command-line option. When set to `internal`, a stacktrace is present in the output only in case of internal exceptions. When set to `all` or `full`, a stacktrace is present in the output for all exceptions and build failures. Using `full` doesn't truncate the stacktrace, which leads to a much more verbose output. *Default is* `internal`.

`org.gradle.welcome=(never,once)`

Controls whether Gradle should print a welcome message. If set to *never* then the welcome message will be suppressed. If set to *once* then the message is printed once for each new version of Gradle. *Default is* `once`.

The following examples demonstrate how to use Gradle properties.

Example 18. Setting Gradle properties with a `gradle.properties` *file*

gradle.properties

```
gradlePropertiesProp=gradlePropertiesValue
gradleProperties.with.dots=gradlePropertiesDottedValue
```

Example 19. Reading Gradle properties at configuration time

settings.gradle

```
// Using the API, provides a lazy Provider<String>
println providers.gradleProperty('gradlePropertiesProp').get()

// Using Groovy dynamic names
println gradlePropertiesProp
println settings.gradlePropertiesProp

// Using Groovy dynamic array notation on `settings`
println settings['gradlePropertiesProp']
```

build.gradle

```
// Using the API, provides a lazy Provider<String>
println providers.gradleProperty('gradlePropertiesProp').get()

// Using Groovy dynamic names
println gradlePropertiesProp
println project.gradlePropertiesProp

// Using Groovy dynamic array notation on `project`
println project['gradlePropertiesProp']
```

settings.gradle.kts

```
// Using the API, provides a lazy Provider<String>
println(providers.gradleProperty("gradlePropertiesProp").get())

// Using Kotlin delegated properties on `settings`
val gradlePropertiesProp: String by settings
println(gradlePropertiesProp)
```

build.gradle.kts

```
// Using the API, provides a lazy Provider<String>
println(providers.gradleProperty("gradlePropertiesProp").get())

// Using Kotlin delegated properties on `project`
val gradlePropertiesProp: String by project
println(gradlePropertiesProp)
```

Note that if a Gradle property has a dot in its name, using the dynamic Groovy names is not possible. You have to use the API or the dynamic array notation instead.

The Kotlin delegated properties are part of the Gradle Kotlin DSL. You need to explicitly specify the type as `String`. If you need to branch depending on the presence of the property, you can also use `String?` and check for `null`.

Example 20. Reading Gradle properties for consumption at execution time

build.gradle

```
tasks.register('printProperty', PrintValue) {
    // Using the API, provides a lazy Provider<String> wired to a task input
    inputValue = providers.gradleProperty('gradlePropertiesProp')
}
```

build.gradle.kts

```
tasks.register<PrintValue>("printProperty") {
    // Using the API, provides a lazy Provider<String> wired to a task input
    inputValue.set(providers.gradleProperty("gradlePropertiesProp"))
}
```

Example 21. Setting Gradle properties from the command line

```
$ gradle -DgradlePropertiesProp=commandLineValue
```

Note that initialization scripts can't read Gradle properties directly. The earliest Gradle properties can be read in initialization scripts is on `settingsEvaluated {}`:

Example 22. Reading Gradle properties from initialization scripts

init.gradle

```
settingsEvaluated { settings ->
    // Using the API, provides a lazy Provider<String>
    println settings.providers.gradleProperty('gradlePropertiesProp').get()

    // Using Groovy dynamic names
    println settings.gradlePropertiesProp

    // Using Groovy dynamic array notation on `settings`
    println settings['gradlePropertiesProp']
}
```

init.gradle.kts

```
settingsEvaluated {
    // Using the API, provides a lazy Provider<String>
    println(providers.gradleProperty("gradlePropertiesProp").get())

    // Using Kotlin delegated properties on `settings`
    val gradlePropertiesProp: String by this
    println(gradlePropertiesProp)
}
```

Properties declared in a `gradle.properties` file present in a subproject directory are only available to that project and its children.

System properties

Using the `-D` command-line option, you can pass a system property to the JVM which runs Gradle. The `-D` option of the `gradle` command has the same effect as the `-D` option of the `java` command.

You can also set system properties in `gradle.properties` files with the prefix `systemProp.`

Example 23. Specifying system properties in `gradle.properties`

```
systemProp.gradle.wrapperUser=myuser
systemProp.gradle.wrapperPassword=mypassword
```

The following system properties are available. Note that command-line options take precedence over system properties.

`gradle.wrapperUser=(myuser)`

Specify user name to download Gradle distributions from servers using HTTP Basic Authentication. Learn more in Authenticated wrapper downloads.

`gradle.wrapperPassword=(mypassword)`

Specify password for downloading a Gradle distribution using the Gradle wrapper.

`gradle.user.home=(path to directory)`

Specify the Gradle user home directory.

`https.protocols`

Specify the supported TLS versions in a comma separated format. For example: `TLSv1.2,TLSv1.3`.

In a multi project build, "`systemProp.`" properties set in any project except the root will be ignored. That is, only the root project's `gradle.properties` file will be checked for properties that begin with the "`systemProp.`" prefix.

The following examples demonstrate how to use System properties.

Example 24. Setting System properties with a `gradle.properties` file

gradle.properties

```
systemProp.system=gradlePropertiesValue
```

Example 25. Reading System properties at configuration time

init.gradle

```
// Using the Java API
println System.getProperty('system')
```

settings.gradle

```
// Using the Java API
println System.getProperty('system')

// Using the Gradle API, provides a lazy Provider<String>
println providers.systemProperty('system').get()
```

build.gradle

```
// Using the Java API
println System.getProperty('system')

// Using the Gradle API, provides a lazy Provider<String>
println providers.systemProperty('system').get()
```

init.gradle.kts

```
// Using the Java API
println(System.getProperty("system"))
```

settings.gradle.kts

```
// Using the Java API
println(System.getProperty("system"))

// Using the Gradle API, provides a lazy Provider<String>
println(providers.systemProperty("system").get())
```

build.gradle.kts

```
// Using the Java API
println(System.getProperty("system"))

// Using the Gradle API, provides a lazy Provider<String>
println(providers.systemProperty("system").get())
```

Example 26. Reading System properties for consumption at execution time

build.gradle

```
tasks.register('printProperty', PrintValue) {
    // Using the Gradle API, provides a lazy Provider<String> wired to a task
input
    inputValue = providers.systemProperty('system')
}
```

build.gradle.kts

```
tasks.register<PrintValue>("printProperty") {
    // Using the Gradle API, provides a lazy Provider<String> wired to a task
input
    inputValue.set(providers.systemProperty("system"))
}
```

Example 27. Setting System properties from the command line

```
$ gradle -Dsystem=commandLineValue
```

Environment variables

The following environment variables are available for the `gradle` command. Note that command-line options and system properties take precedence over environment variables.

`GRADLE_OPTS`

Specifies JVM arguments to use when starting the Gradle client VM. The client VM only handles command line input/output, so it is rare that one would need to change its VM options. The actual build is run by the Gradle daemon, which is not affected by this environment variable.

`GRADLE_USER_HOME`

Specifies the Gradle user home directory (which defaults to `$USER_HOME/.gradle` if not set).

`JAVA_HOME`

Specifies the JDK installation directory to use for the client VM. This VM is also used for the daemon, unless a different one is specified in a Gradle properties file with `org.gradle.java.home`.

The following examples demonstrate how to use environment variables.

Example 28. Reading environment variables at configuration time

init.gradle

```
// Using the Java API
println System.getenv('ENVIRONMENTAL')
```

settings.gradle

```
// Using the Java API
println System.getenv('ENVIRONMENTAL')

// Using the Gradle API, provides a lazy Provider<String>
println providers.environmentVariable('ENVIRONMENTAL').get()
```

build.gradle

```
// Using the Java API
println System.getenv('ENVIRONMENTAL')

// Using the Gradle API, provides a lazy Provider<String>
println providers.environmentVariable('ENVIRONMENTAL').get()
```

init.gradle.kts

```
// Using the Java API
println(System.getenv("ENVIRONMENTAL"))
```

settings.gradle.kts

```
// Using the Java API
println(System.getenv("ENVIRONMENTAL"))

// Using the Gradle API, provides a lazy Provider<String>
println(providers.environmentVariable("ENVIRONMENTAL").get())
```

build.gradle.kts

```
// Using the Java API
println(System.getenv("ENVIRONMENTAL"))

// Using the Gradle API, provides a lazy Provider<String>
println(providers.environmentVariable("ENVIRONMENTAL").get())
```

Example 29. Reading environment variables for consumption at execution time

build.gradle

```
tasks.register('printValue', PrintValue) {
    // Using the Gradle API, provides a lazy Provider<String> wired to a task
input
    inputValue = providers.environmentVariable('ENVIRONMENTAL')
}
```

build.gradle.kts

```
tasks.register<PrintValue>("printValue") {
    // Using the Gradle API, provides a lazy Provider<String> wired to a task
input
    inputValue.set(providers.environmentVariable("ENVIRONMENTAL"))
}
```

Project properties

Project properties are available on the Project object. They can be set from the command line using the `-P` / `--project-prop` environment option.

Example 30. Setting a project property via the command line

```
$ gradle -PgradlePropertiesProp=commandLineValue
```

Gradle can also set project properties when it sees specially-named system properties or environment variables. If the environment variable name looks like `ORG_GRADLE_PROJECT_prop=somevalue`, then Gradle will set a `prop` property on your project object, with the value of `somevalue`. Gradle also supports this for system properties, but with a different naming pattern, which looks like `org.gradle.project.prop`. Both of the following will set the `foo` property on your Project object to `"bar"`.

Example 31. Setting a project property via a system property

```
org.gradle.project.foo=bar
```

Example 32. Setting a project property via an environment variable

```
ORG_GRADLE_PROJECT_foo=bar
```

This feature is useful when you don't have admin rights to a continuous integration server, and you need to set property values that should not be easily visible. Since you cannot use the `-P` option in that scenario, nor change the system-level configuration files, the correct strategy is to change the configuration of your continuous integration build job, adding an environment variable setting that matches an expected pattern. This won't be visible to normal users on the system.

The following examples demonstrate how to use project properties.

Example 33. Reading project properties at configuration time

build.gradle

```
// Querying the presence of a project property
if (hasProperty('myProjectProp')) {
    // Accessing the value, throws if not present
    println property('myProjectProp')
}

// Accessing the value of a project property, null if absent
println findProperty('myProjectProp')

// Accessing the Map<String, ?> of project properties
println properties['myProjectProp']

// Using Groovy dynamic names, throws if not present
println myProjectProp
```

build.gradle.kts

```
// Querying the presence of a project property
if (hasProperty("myProjectProp")) {
    // Accessing the value, throws if not present
    println(property("myProjectProp"))
}

// Accessing the value of a project property, null if absent
println(findProperty("myProjectProp"))

// Accessing the Map<String, Any?> of project properties
println(properties["myProjectProp"])

// Using Kotlin delegated properties on `project`
val myProjectProp: String by project
println(myProjectProp)
```

Note that if a Project property has a dot in its name, using the dynamic Groovy names is not possible. You have to use the API or the dynamic array notation instead.

The Kotlin delegated properties are part of the Gradle Kotlin DSL. You need to explicitly specify the type as `String`. If you need to branch depending on the presence of the property, you can also use `String?` and check for `null`.

Example 34. Reading project properties for consumption at execution time

build.gradle

```
tasks.register('printValue', PrintValue) {
    // Eagerly accessing the value of a project property, set as a task input
    inputValue = project.property('myProjectProp')
}
```

build.gradle.kts

```
tasks.register<PrintValue>("printValue") {
    // Eagerly accessing the value of a project property, set as a task input
    inputValue.set(project.property("myProjectProp").toString())
}
```

NOTE

If a project property is referenced but does not exist, an exception will be thrown and the build will fail.

You should check for existence of optional project properties before you access them using the Project.hasProperty(java.lang.String) method.

Configuring JVM memory

You can adjust JVM options for Gradle in the following ways:

The `org.gradle.jvmargs` Gradle property controls the VM running the build. It defaults to `-Xmx512m "-XX:MaxMetaspaceSize=256m"`

Example 35. Changing JVM settings for the build VM

```
org.gradle.jvmargs=-Xmx2g -XX:MaxMetaspaceSize=512m
-XX:+HeapDumpOnOutOfMemoryError -Dfile.encoding=UTF-8
```

The `JAVA_OPTS` environment variable controls the command line client, which is only used to display console output. It defaults to `-Xmx64m`

Example 36. Changing JVM settings for the client VM

```
JAVA_OPTS="-Xmx64m -XX:+HeapDumpOnOutOfMemoryError -Dfile.encoding=UTF-8"
```

NOTE	There is one case where the client VM can also serve as the build VM: If you deactivate the Gradle Daemon and the client VM has the same settings as required for the build VM, the client VM will run the build directly. Otherwise the client VM will fork a new VM to run the actual build in order to honor the different settings.

Certain tasks, like the `test` task, also fork additional JVM processes. You can configure these through the tasks themselves. They all use `-Xmx512m` by default.

Example 37. Set Java compile options for JavaCompile tasks

build.gradle

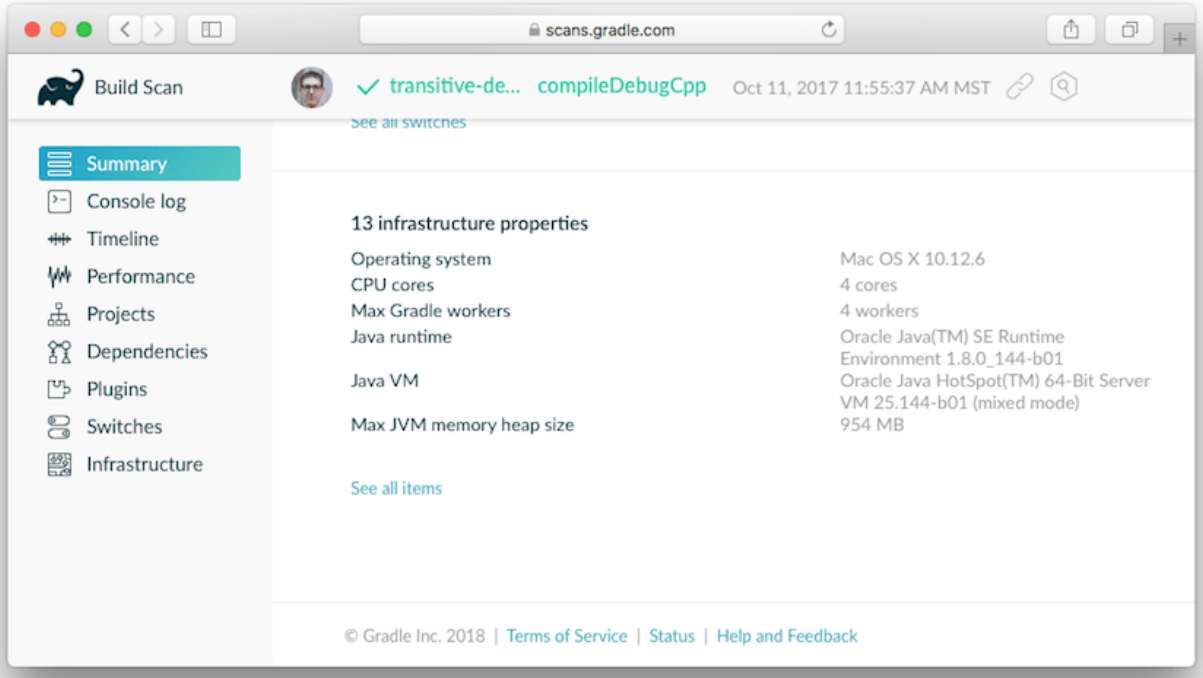
```
plugins {
    id 'java'
}

tasks.withType(JavaCompile) {
    options.compilerArgs += ['-Xdoclint:none', '-Xlint:none', '-nowarn']
}
```

build.gradle.kts

```
plugins {
    java
}

tasks.withType<JavaCompile>().configureEach {
    options.compilerArgs = listOf("-Xdoclint:none", "-Xlint:none", "-nowarn")
}
```

See other examples in the Test API documentation and test execution in the Java plugin reference.

Build scans will tell you information about the JVM that executed the build when you use the `--scan` option.



Configuring a task using project properties

It's possible to change the behavior of a task based on project properties specified at invocation time.

Suppose you'd like to ensure release builds are only triggered by CI. A simple way to handle this is through an `isCI` project property.

Example 38. Prevent releasing outside of CI

build.gradle

```
tasks.register('performRelease') {
    doLast {
        if (project.hasProperty("isCI")) {
            println("Performing release actions")
        } else {
            throw new InvalidUserDataException("Cannot perform release
outside of CI")
        }
    }
}
```

build.gradle.kts

```
tasks.register("performRelease") {
    doLast {
        if (project.hasProperty("isCI")) {
            println("Performing release actions")
        } else {
            throw InvalidUserDataException("Cannot perform release outside of
CI")
        }
    }
}
```

```
$ gradle performRelease -PisCI=true --quiet
Performing release actions
```

Accessing the web through a proxy

Configuring a proxy (for downloading dependencies, for example) is done via standard JVM system properties. These properties can be set directly in the build script; for example, setting the HTTP proxy host would be done with `System.setProperty('http.proxyHost', 'www.somehost.org')`. Alternatively, the properties can be specified in gradle.properties.

Example 39. Configuring an HTTP proxy using `gradle.properties`

```
systemProp.http.proxyHost=www.somehost.org
systemProp.http.proxyPort=8080
systemProp.http.proxyUser=userid
systemProp.http.proxyPassword=password
systemProp.http.nonProxyHosts=*.nonproxyrepos.com|localhost
```

There are separate settings for HTTPS.

Example 40. Configuring an HTTPS proxy using `gradle.properties`

```
systemProp.https.proxyHost=www.somehost.org
systemProp.https.proxyPort=8080
systemProp.https.proxyUser=userid
systemProp.https.proxyPassword=password
# NOTE: this is not a typo.
systemProp.http.nonProxyHosts=*.nonproxyrepos.com|localhost
```

There are separate settings for SOCKS.

Example 41. Configuring a SOCKS proxy using `gradle.properties`

```
systemProp.socksProxyHost=www.somehost.org
systemProp.socksProxyPort=1080
systemProp.java.net.socks.username=userid
systemProp.java.net.socks.password=password
```

You may need to set other properties to access other networks. Here are 2 references that may be helpful:

- ProxySetup.java in the Ant codebase
- JDK 8 Proxies

NTLM Authentication

If your proxy requires NTLM authentication, you may need to provide the authentication domain as well as the username and password. There are 2 ways that you can provide the domain for authenticating to a NTLM proxy:

- Set the `http.proxyUser` system property to a value like `domain/username`.
- Provide the authentication domain via the `http.auth.ntlm.domain` system property.

The Gradle Daemon

> A daemon is a computer program that runs as a background process, rather than being under the direct control of an interactive user.
>
> — Wikipedia

Gradle runs on the Java Virtual Machine (JVM) and uses several supporting libraries with non-trivial initialization time. Startup can be slow. The **Gradle Daemon** solves this problem.

The Daemon is a long-lived background process that reduces the time it takes to run a build. The Daemon reduces build times by:

- caching project information across builds
- running in the background so every Gradle build doesn't have to wait for JVM startup
- benefiting from continuous runtime optimization in the JVM
- watching the file system to calculate exactly what needs to be rebuilt before you run a build

Check Status

To get a list of running Daemons and their statuses, use the `--status` command:

```
$ gradle --status
```

Sample output:

```
  PID STATUS   INFO
28486 IDLE     7.5
34247 BUSY     7.5
```

Currently, a given Gradle version can only connect to Daemons of the same version. This means the status output only shows Daemons spawned running the same version of Gradle as the current project.

Find all Daemons

If you have the Java Development Kit (JDK) installed, you can view live Daemons with the `jps` command. Live Daemons appear under the name `GradleDaemon`. Because this command uses the JDK, you can view Daemons running any version of Gradle.

Enable

Gradle enables the Daemon by default since Gradle 3.0. If your project doesn't use the Daemon, you can enable it for a single build with the `--daemon` flag when you run a build:

```
$ gradle <task> --daemon
```

This flag overrides any settings that disable the Daemon in your project or user `gradle.properties` files.

To enable the Daemon by default in older Gradle versions, add the following setting to the `gradle.properties` file in the project root or your Gradle user home:

gradle.properties

```
org.gradle.daemon=true
```

Disable

You can disable the Daemon in multiple ways.

Disable for a Build

To disable the Daemon for a single build, pass the `--no-daemon` flag when you run a build:

```
$ gradle <task> --no-daemon
```

This flag overrides any settings that enable the Daemon in your project or user `gradle.properties` files.

Disable for a Project

To disable the Daemon for all builds of a project, add `org.gradle.daemon=false` to the `gradle.properties` file in the project root.

Disable for a User

On Windows, this command disables the Daemon for the current user:

```
(if not exist "%USERPROFILE%/.gradle" mkdir "%USERPROFILE%/.gradle") && (echo. >>
"%USERPROFILE%/.gradle/gradle.properties" && echo org.gradle.daemon=false >>
"%USERPROFILE%/.gradle/gradle.properties")
```

On UNIX-like operating systems, the following Bash shell command disables the Daemon for the current user:

```
mkdir -p ~/.gradle && echo "org.gradle.daemon=false" >> ~/.gradle/gradle.properties
```

Disable Globally

There are two recommended ways to disable the Daemon globally across an environment:

- add `org.gradle.daemon=false` to the `<GRADLE_USER_HOME>/gradle.properties` file
- add the flag `-Dorg.gradle.daemon=false` to the `GRADLE_OPTS` environment variable

Stop

It can be helpful to stop the Daemon when troubleshooting or debugging a failure. Daemons automatically stop given any of the following conditions:

- available system memory is low
- the Daemon has been idle for 3 hours

To stop running Daemon processes, use the following command:

```
$ gradle --stop
```

This terminates all Daemon processes started with the same version of Gradle used to execute the command.

You can also kill Daemons manually with your operating system. To find the PIDs for all Daemons regardless of Gradle version, see Find all Daemons.

Tools & IDEs

The Gradle Tooling API used by IDEs and other tools to integrate with Gradle *always* uses the Gradle Daemon to execute builds. If you execute Gradle builds from within your IDE, you already use the Gradle Daemon. There's no need to enable it for your environment.

Continuous Integration

We recommend using the Daemon for both developer machines and Continuous Integration servers.

Compatibility

Gradle starts a new Daemon if no idle or compatible Daemons exist. The following values determine compatibility:

- **Requested build environment**, including the following:
  - Java version

    - JVM attributes
    - JVM properties
- Gradle version

Compatibility is based on exact matches of these values. For example:

- If a Daemon is available with a Java 8 runtime, but the requested build environment calls for Java 10, then the Daemon is not compatible.
- If a Daemon is available running Gradle 7.0, but the current build uses Gradle 7.4, then the Daemon is not compatible.

Certain properties of a Java runtime are *immutable*: they cannot be changed once the JVM has started. The following JVM system properties are immutable:

- `file.encoding`
- `user.language`
- `user.country`
- `user.variant`
- `java.io.tmpdir`
- `javax.net.ssl.keyStore`
- `javax.net.ssl.keyStorePassword`
- `javax.net.ssl.keyStoreType`
- `javax.net.ssl.trustStore`
- `javax.net.ssl.trustStorePassword`
- `javax.net.ssl.trustStoreType`
- `com.sun.management.jmxremote`

The following JVM attributes controlled by startup arguments are also immutable:

- The maximum heap size (the `-Xmx` JVM argument)
- The minimum heap size (the `-Xms` JVM argument)
- The boot classpath (the `-Xbootclasspath` argument)
- The “assertion” status (the `-ea` argument)

If the requested build environment requirements for any of these properties and attributes differ from the Daemon’s JVM requirements, the Daemon is not compatible.

NOTE	For more information about build environments, see the build environment documentation.

Performance Impact

The Daemon can reduce build times by 15-75% when you build the same project repeatedly.

TIP	To get a sense of the Daemon's impact on your builds, you can profile your build with `--profile`.

In between builds, the Daemon waits idly for the next build. As a result, your machine only loads Gradle into memory once for multiple builds, instead of once per build. This is a significant performance optimization. But that's not where it stops.

Runtime Code Optimizations

The JVM gains significant performance from **runtime code optimization**: optimizations applied to code while it runs. JVM implementations like OpenJDK's Hotspot progressively optimize code during execution. Subsequent builds can be faster purely due to this optimization process. Experiments with HotSpot show that it takes somewhere between 5 and 10 builds for optimization to stabilize. Thanks to the Daemon, perceived build times can drop dramatically between the first build and tenth builds of a project.

Memory Caching

The Daemon enables in-memory caching across builds. This includes classes for plugins and build scripts. Similarly, the Daemon maintains in-memory caches of build data such as the hashes of task inputs and outputs for incremental builds.

Performance Monitoring

Gradle actively monitors heap usage to detect memory leaks in the Daemon. When a memory leak exhausts available heap space, the Daemon:

1. Finishes the currently running build.
2. Restarts before running the next build.

Gradle enables this monitoring by default. To disable this monitoring, set the `org.gradle.daemon.performance.enable-monitoring` Daemon option to `false`. You can do this on the command line with the following command:

```
$ gradle <task> -Dorg.gradle.daemon.performance.enable-monitoring=false
```

Or configure the property in the `gradle.properties` file in the project root or your Gradle project home:

gradle.properties

```
org.gradle.daemon.performance.enable-monitoring=false
```

FAQ

Why is there more than one Daemon process on my machine?

Gradle starts a new Daemon process for your build if no idle Daemon exists with a compatible configuration. For more information about compability, see Compatibility.

How much memory does the Daemon use and can I give it more?

If the requested build environment does not specify a maximum heap size, the Daemon uses up to 512MB of heap. Daemons use the JVM's default minimum heap size. 512MB is more than enough for most builds. Larger builds with hundreds of subprojects, lots of configuration, and source code may benefit from a larger heap size.

To increase the amount of memory the Daemon can use, specify the appropriate flags as part of the requested build environment. Please see the build environment documentation for details.

What can go wrong with the Daemon?

User build scripts and third party plugins can destabilize the Daemon through memory leaks, poor resource management, and global state corruption. Failing to close files after reading and writing is a common cause. The Microsoft Windows operating system frequently stops Daemon processes destabilized in this way.

Specify the `--no-daemon` switch for a build to prevent use of the Daemon. This can help determine if the Daemon is causing a problem with your build.

Initialization Scripts

Gradle provides a powerful mechanism to allow customizing the build based on the current environment. This mechanism also supports tools that wish to integrate with Gradle.

Note that this is completely different from the "`init`" task provided by the "`build-init`" plugin (see Build Init Plugin).

Basic usage

Initialization scripts (a.k.a. *init scripts*) are similar to other scripts in Gradle. These scripts, however, are run before the build starts. Here are several possible uses:

- Set up enterprise-wide configuration, such as where to find custom plugins.
- Set up properties based on the current environment, such as a developer's machine vs. a continuous integration server.

- Supply personal information about the user that is required by the build, such as repository or database authentication credentials.
- Define machine specific details, such as where JDKs are installed.
- Register build listeners. External tools that wish to listen to Gradle events might find this useful.
- Register build loggers. You might wish to customize how Gradle logs the events that it generates.

One main limitation of init scripts is that they cannot access classes in the `buildSrc` project (see Using buildSrc to extract imperative logic for details of this feature).

Using an init script

There are several ways to use an init script:

- Specify a file on the command line. The command line option is `-I` or `--init-script` followed by the path to the script. The command line option can appear more than once, each time adding another init script. The build will fail if any of the files specified on the command line does not exist.
- Put a file called `init.gradle` (or `init.gradle.kts` for Kotlin) in the `USER_HOME/.gradle/` directory.
- Put a file that ends with `.gradle` (or `.init.gradle.kts` for Kotlin) in the `USER_HOME/.gradle/init.d/` directory.
- Put a file that ends with `.gradle` (or `.init.gradle.kts` for Kotlin) in the `GRADLE_HOME/init.d/` directory, in the Gradle distribution. This allows you to package up a custom Gradle distribution containing some custom build logic and plugins. You can combine this with the Gradle wrapper as a way to make custom logic available to all builds in your enterprise.

If more than one init script is found they will all be executed, in the order specified above. Scripts in a given directory are executed in alphabetical order. This allows, for example, a tool to specify an init script on the command line and the user to put one in their home directory for defining the environment and both scripts will run when Gradle is executed.

Writing an init script

Similar to a Gradle build script, an init script is a Groovy or Kotlin script. Each init script has a Gradle instance associated with it. Any property reference and method call in the init script will delegate to this `Gradle` instance.

Each init script also implements the Script interface.

NOTE	When writing init scripts, pay attention to the scope of the reference you are trying to access. For example, properties loaded from `gradle.properties` are available on `Settings` or `Project` instances but not on the `Gradle` one.

Configuring projects from an init script

You can use an init script to configure the projects in the build. This works in a similar way to configuring projects in a multi-project build. The following sample shows how to perform extra configuration from an init script *before* the projects are evaluated. This sample uses this feature to

configure an extra repository to be used only for certain environments.

Example 42. Using init script to perform extra configuration before projects are evaluated

build.gradle

```
repositories {
    mavenCentral()
}

tasks.register('showRepos') {
    doLast {
        println "All repos:"
        println repositories.collect { it.name }
    }
}
```

init.gradle

```
allprojects {
    repositories {
        mavenLocal()
    }
}
```

build.gradle.kts

```
repositories {
    mavenCentral()
}

tasks.register("showRepos") {
    doLast {
        println("All repos:")
        println(repositories.map { it.name })
    }
}
```

init.gradle.kts

```
allprojects {
    repositories {
        mavenLocal()
    }
}
```

Output when applying the init script

```
> gradle --init-script init.gradle -q showRepos
All repos:
[MavenLocal, MavenRepo]
```

```
> gradle --init-script init.gradle.kts -q showRepos
All repos:
[MavenLocal, MavenRepo]
```

External dependencies for the init script

In External dependencies for the build script it was explained how to add external dependencies to a build script. Init scripts can also declare dependencies. You do this with the `initscript()` method, passing in a closure which declares the init script classpath.

Example 43. Declaring external dependencies for an init script

init.gradle

```
initscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.apache.commons:commons-math:2.0'
    }
}
```

init.gradle.kts

```
initscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.apache.commons:commons-math:2.0")
    }
}
```

The closure passed to the `initscript()` method configures a ScriptHandler instance. You declare the init script classpath by adding dependencies to the `classpath` configuration. This is the same way you declare, for example, the Java compilation classpath. You can use any of the dependency types

described in Declaring Dependencies, except project dependencies.

Having declared the init script classpath, you can use the classes in your init script as you would any other classes on the classpath. The following example adds to the previous example, and uses classes from the init script classpath.

Example 44. An init script with external dependencies

init.gradle

```
import org.apache.commons.math.fraction.Fraction

initscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.apache.commons:commons-math:2.0'
    }
}

println Fraction.ONE_FIFTH.multiply(2)
```

build.gradle

```
tasks.register('doNothing')
```

init.gradle.kts

```
import org.apache.commons.math.fraction.Fraction

initscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.apache.commons:commons-math:2.0")
    }
}

println(Fraction.ONE_FIFTH.multiply(2))
```

build.gradle.kts

```
tasks.register("doNothing")
```

Output when applying the init script

```
> gradle --init-script init.gradle -q doNothing
2 / 5
```

```
> gradle --init-script init.gradle.kts -q doNothing
2 / 5
```

Init script plugins

Similar to a Gradle build script or a Gradle settings file, plugins can be applied on init scripts.

Example 45. Using plugins in init scripts

init.gradle

```
apply plugin: EnterpriseRepositoryPlugin

class EnterpriseRepositoryPlugin implements Plugin<Gradle> {

    private static String ENTERPRISE_REPOSITORY_URL =
"https://repo.gradle.org/gradle/repo"

    void apply(Gradle gradle) {
        // ONLY USE ENTERPRISE REPO FOR DEPENDENCIES
        gradle.allprojects { project ->
            project.repositories {

                // Remove all repositories not pointing to the enterprise
repository url
                all { ArtifactRepository repo ->
                    if (!(repo instanceof MavenArtifactRepository) ||
                          repo.url.toString() != ENTERPRISE_REPOSITORY_URL) {
                        project.logger.lifecycle "Repository ${repo.url}
removed. Only $ENTERPRISE_REPOSITORY_URL is allowed"
                        remove repo
                    }
                }

                // add the enterprise repository
                maven {
                    name "STANDARD_ENTERPRISE_REPO"
                    url ENTERPRISE_REPOSITORY_URL
                }
            }
        }
    }
}
```

build.gradle

```
repositories{
    mavenCentral()
}

tasks.register('showRepositories') {
     doLast {
         repositories.each {
             println "repository: ${it.name} ('${it.url}')"
         }
     }
}
```

init.gradle.kts

```
apply<EnterpriseRepositoryPlugin>()

class EnterpriseRepositoryPlugin : Plugin<Gradle> {
    companion object {
        const val ENTERPRISE_REPOSITORY_URL =
"https://repo.gradle.org/gradle/repo"
    }

    override fun apply(gradle: Gradle) {
        // ONLY USE ENTERPRISE REPO FOR DEPENDENCIES
        gradle.allprojects {
            repositories {

                // Remove all repositories not pointing to the enterprise
repository url
                all {
                    if (this !is MavenArtifactRepository || url.toString() !=
ENTERPRISE_REPOSITORY_URL) {
                        project.logger.lifecycle("Repository ${(this as?
MavenArtifactRepository)?.url ?: name} removed. Only
$ENTERPRISE_REPOSITORY_URL is allowed")
                        remove(this)
                    }
                }

                // add the enterprise repository
                add(maven {
                    name = "STANDARD_ENTERPRISE_REPO"
                    url = uri(ENTERPRISE_REPOSITORY_URL)
                })
            }
        }
    }
}
```

build.gradle.kts

```
repositories{
    mavenCentral()
}

tasks.register("showRepositories") {
    doLast {
        repositories.map { it as MavenArtifactRepository }.forEach {
            println("repository: ${it.name} ('${it.url}')")
        }
    }
}
```

Output when applying the init script

```
> gradle --init-script init.gradle -q showRepositories
repository: STANDARD_ENTERPRISE_REPO ('https://repo.gradle.org/gradle/repo')
```

```
> gradle --init-script init.gradle.kts -q showRepositories
repository: STANDARD_ENTERPRISE_REPO ('https://repo.gradle.org/gradle/repo')
```

The plugin in the init script ensures that only a specified repository is used when running the build.

When applying plugins within the init script, Gradle instantiates the plugin and calls the plugin instance's Plugin.apply(T) method. The `gradle` object is passed as a parameter, which can be used to configure all aspects of a build. Of course, the applied plugin can be resolved as an external dependency as described in External dependencies for the init script

Executing Multi-Project Builds

Only the smallest of projects has a single build file and source tree, unless it happens to be a massive, monolithic application. It's often much easier to digest and understand a project that has been split into smaller, inter-dependent modules. The word "inter-dependent" is important, though, and is why you typically want to link the modules together through a single build.

Gradle supports this scenario through *multi-project* builds.

For details about authoring *multi-project* builds, consult the Authoring Multi-Project Builds section of the user manual.

Identifying project structure

To identify the project structure, you can use `gradle projects` command. As an example, let's use a multi-project build with the following structure:

```
> gradle -q projects

------------------------------------------------------------
Root project 'multiproject'
------------------------------------------------------------

Root project 'multiproject'
+--- Project ':api'
+--- Project ':services'
|    +--- Project ':services:shared'
|    \--- Project ':services:webservice'
\--- Project ':shared'

To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :api:tasks
```

From a user's perspective, multi-project builds are still collections of tasks you can run. The difference is that you may want to control *which* project's tasks get executed. The following sections will cover the two options you have for executing tasks in a multi-project build.

Executing tasks by name

The command `gradle test` will execute the `test` task in any subprojects, relative to the current working directory, that have that task. If you run the command from the root project directory, you'll run `test` in *api*, *shared*, *services:shared* and *services:webservice*. If you run the command from the *services* project directory, you'll only execute the task in *services:shared* and *services:webservice*.

The basic rule behind Gradle's behavior is: execute all tasks down the hierarchy which have this name. Only complain if there is *no* such task found in any of the subprojects traversed.

NOTE	Some tasks selectors, like `help` or `dependencies`, will only run the task on the project they are invoked on and not on all the subprojects. The main motivation for this is that these tasks print out information that would be hard to process if it combined the information from all projects.

Gradle looks down the hierarchy, starting with the *current dir*, for tasks with the given name and executes them. One thing is very important to note. Gradle *always* evaluates *every* project of the multi-project build and creates all existing task objects. Then, according to the task name arguments and the current directory, Gradle filters the tasks which should be executed. Because of Gradle's cross project configuration, *every* project has to be evaluated before *any* task gets executed.

When you're using the Gradle wrapper, executing a task for a specific subproject by running Gradle from the subproject's directory doesn't work well because you have to specify the path to the wrapper script if you're not in the project root. For example, if want to run `build` task for the *webservice* subproject and you're in the *webservice* subproject directory, you would have to run `../../gradlew build`. The next section shows how this can be achieved directly from the project's root directory.

Executing tasks by fully qualified name

You can use a task's fully qualified name to execute a specific task in a specific subproject. For example: `gradle :services:webservice:build` will run the `build` task of the *webservice* subproject. The fully qualified name of a task is simply its project path plus the task name.

A project path has the following pattern: it starts with an optional colon, which denotes the root project. The root project is the only project in a path that is not specified by its name. The rest of a project path is a colon-separated sequence of project names, where the next project is a subproject of the previous project. You can see the project paths when running `gradle projects` as shown in identifying project structure section.

This approach works for any task, so if you want to know what tasks are in a particular subproject, just use the `tasks` task, e.g. `gradle :services:webservice:tasks`.

Regardless of which technique you use to execute tasks, Gradle will take care of building any subprojects that the target depends on. You don't have to worry about the inter-project dependencies yourself. If you're interested in how this is configured, you can read about writing multi-project builds later in the user manual.

That's all you really need to know about multi-project builds as a build user. You can now identify whether a build is a multi-project one and you can discover its structure. And finally, you can execute tasks within specific subprojects.

Multi-Project Building and Testing

The `build` task of the Java plugin is typically used to compile, test, and perform code style checks (if the CodeQuality plugin is used) of a single project. In multi-project builds you may often want to do all of these tasks across a range of projects. The `buildNeeded` and `buildDependents` tasks can help with this.

In this example, the `:services:person-service` project depends on both the `:api` and `:shared` projects. The `:api` project also depends on the `:shared` project.

Assume you are working on a single project, the `:api` project. You have been making changes, but have not built the entire project since performing a clean. You want to build any necessary supporting jars, but only perform code quality and unit tests on the project you have changed. The `build` task does this.

Example 46. Build and Test Single Project

Output of `gradle :api:build`

```
> gradle :api:build
> Task :shared:compileJava
> Task :shared:processResources
> Task :shared:classes
> Task :shared:jar
> Task :api:compileJava
> Task :api:processResources
> Task :api:classes
> Task :api:jar
> Task :api:assemble
> Task :api:compileTestJava
> Task :api:processTestResources
> Task :api:testClasses
> Task :api:test
> Task :api:check
> Task :api:build

BUILD SUCCESSFUL in 0s
```

If you have just gotten the latest version of source from your version control system which included changes in other projects that `:api` depends on, you might want to not only build all the projects you depend on, but test them as well. The `buildNeeded` task also tests all the projects from the project dependencies of the testRuntime configuration.

Example 47. Build and Test Depended On Projects

Output of `gradle :api:buildNeeded`

```
> gradle :api:buildNeeded
> Task :shared:compileJava
> Task :shared:processResources
> Task :shared:classes
> Task :shared:jar
> Task :api:compileJava
> Task :api:processResources
> Task :api:classes
> Task :api:jar
> Task :api:assemble
> Task :api:compileTestJava
> Task :api:processTestResources
> Task :api:testClasses
> Task :api:test
> Task :api:check
> Task :api:build
> Task :shared:assemble
> Task :shared:compileTestJava
> Task :shared:processTestResources
> Task :shared:testClasses
> Task :shared:test
> Task :shared:check
> Task :shared:build
> Task :shared:buildNeeded
> Task :api:buildNeeded

BUILD SUCCESSFUL in 0s
```

You also might want to refactor some part of the `:api` project that is used in other projects. If you make these types of changes, it is not sufficient to test just the `:api` project, you also need to test all projects that depend on the `:api` project. The `buildDependents` task also tests all the projects that have a project dependency (in the testRuntime configuration) on the specified project.

Example 48. Build and Test Dependent Projects

Output of `gradle :api:buildDependents`

```
> gradle :api:buildDependents
> Task :shared:compileJava
> Task :shared:processResources
> Task :shared:classes
> Task :shared:jar
> Task :api:compileJava
> Task :api:processResources
> Task :api:classes
> Task :api:jar
> Task :api:assemble
> Task :api:compileTestJava
> Task :api:processTestResources
> Task :api:testClasses
> Task :api:test
> Task :api:check
> Task :api:build
> Task :services:person-service:compileJava
> Task :services:person-service:processResources
> Task :services:person-service:classes
> Task :services:person-service:jar
> Task :services:person-service:assemble
> Task :services:person-service:compileTestJava
> Task :services:person-service:processTestResources
> Task :services:person-service:testClasses
> Task :services:person-service:test
> Task :services:person-service:check
> Task :services:person-service:build
> Task :services:person-service:buildDependents
> Task :api:buildDependents

BUILD SUCCESSFUL in 0s
```

Finally, you may want to build and test everything in all projects. Any task you run in the root project folder will cause that same named task to be run on all the children. So you can just run `gradle build` to build and test all projects.

Build Cache

TIP Want to learn the tips and tricks top engineering teams use to keep builds fast and performant? Register here for our Build Cache Training.

Overview

The Gradle *build cache* is a cache mechanism that aims to save time by reusing outputs produced by other builds. The build cache works by storing (locally or remotely) build outputs and allowing builds to fetch these outputs from the cache when it is determined that inputs have not changed, avoiding the expensive work of regenerating them.

A first feature using the build cache is *task output caching*. Essentially, task output caching leverages the same intelligence as up-to-date checks that Gradle uses to avoid work when a previous local build has already produced a set of task outputs. But instead of being limited to the previous build in the same workspace, task output caching allows Gradle to reuse task outputs from any earlier build in any location on the local machine. When using a shared build cache for task output caching this even works across developer machines and build agents.

Apart from tasks, artifact transforms can also leverage the build cache and re-use their outputs similarly to task output caching.

TIP	For a hands-on approach to learning how to use the build cache, start with reading through the use cases for the build cache and the follow up sections. It covers the different scenarios that caching can improve and has detailed discussions of the different caveats you need to be aware of when enabling caching for a build.

Enable the Build Cache

By default, the build cache is not enabled. You can enable the build cache in a couple of ways:

Run with `--build-cache` on the command-line

Gradle will use the build cache for this build only.

Put `org.gradle.caching=true` in your `gradle.properties`

Gradle will try to reuse outputs from previous builds for all builds, unless explicitly disabled with `--no-build-cache`.

When the build cache is enabled, it will store build outputs in the Gradle user home. For configuring this directory or different kinds of build caches see Configure the Build Cache.

Task Output Caching

Beyond incremental builds described in up-to-date checks, Gradle can save time by reusing outputs from previous executions of a task by matching inputs to the task. Task outputs can be reused between builds on one computer or even between builds running on different computers via a build cache.

We have focused on the use case where users have an organization-wide remote build cache that is populated regularly by continuous integration builds. Developers and other continuous integration agents should load cache entries from the remote build cache. We expect that developers will not be allowed to populate the remote build cache, and all continuous integration builds populate the build cache after running the `clean` task.

For your build to play well with task output caching it must work well with the incremental build feature. For example, when running your build twice in a row all tasks with outputs should be `UP-TO-DATE`. You cannot expect faster builds or correct builds when enabling task output caching when this prerequisite is not met.

Task output caching is automatically enabled when you enable the build cache, see Enable the Build Cache.

What does it look like

Let us start with a project using the Java plugin which has a few Java source files. We run the build the first time.

```
> gradle --build-cache compileJava
:compileJava
:processResources
:classes
:jar
:assemble

BUILD SUCCESSFUL
```

We see the directory used by the local build cache in the output. Apart from that the build was the same as without the build cache. Let's clean and run the build again.

```
> gradle clean
:clean

BUILD SUCCESSFUL
```

```
> gradle --build-cache assemble
:compileJava FROM-CACHE
:processResources
:classes
:jar
:assemble

BUILD SUCCESSFUL
```

Now we see that, instead of executing the `:compileJava` task, the outputs of the task have been loaded from the build cache. The other tasks have not been loaded from the build cache since they are not cacheable. This is due to `:classes` and `:assemble` being lifecycle tasks and `:processResources` and `:jar` being Copy-like tasks which are not cacheable since it is generally faster to execute them.

Cacheable tasks

Since a task describes all of its inputs and outputs, Gradle can compute a *build cache key* that uniquely defines the task's outputs based on its inputs. That build cache key is used to request previous outputs from a build cache or store new outputs in the build cache. If the previous build outputs have been already stored in the cache by someone else, e.g. your continuous integration server or other developers, you can avoid executing most tasks locally.

The following inputs contribute to the build cache key for a task in the same way that they do for up-to-date checks:

- The task type and its classpath
- The names of the output properties
- The names and values of properties annotated as described in the section called "Custom task types"
- The names and values of properties added by the DSL via TaskInputs
- The classpath of the Gradle distribution, buildSrc and plugins
- The content of the build script when it affects execution of the task

Task types need to opt-in to task output caching using the @CacheableTask annotation. Note that @CacheableTask is not inherited by subclasses. Custom task types are *not* cacheable by default.

Built-in cacheable tasks

Currently, the following built-in Gradle tasks are cacheable:

- Java toolchain: JavaCompile, Javadoc
- Groovy toolchain: GroovyCompile, Groovydoc
- Scala toolchain: ScalaCompile, PlatformScalaCompile, ScalaDoc
- Native toolchain: CppCompile, CCompile, SwiftCompile
- Testing: Test
- Code quality tasks: Checkstyle, CodeNarc, Pmd
- JaCoCo: JacocoMerge, JacocoReport
- Other tasks: AntlrTask, ValidatePlugins, WriteProperties

All other built-in tasks are currently not cacheable.

Some tasks, like Copy or Jar, usually do not make sense to make cacheable because Gradle is only copying files from one location to another. It also doesn't make sense to make tasks cacheable that do not produce outputs or have no task actions.

Third party plugins

There are third party plugins that work well with the build cache. The most prominent examples are the Android plugin 3.1+ and the Kotlin plugin 1.2.21+. For other third party plugins, check their

documentation to find out whether they support the build cache.

Declaring task inputs and outputs

It is very important that a cacheable task has a complete picture of its inputs and outputs, so that the results from one build can be safely re-used somewhere else.

Missing task inputs can cause incorrect cache hits, where different results are treated as identical because the same cache key is used by both executions. Missing task outputs can cause build failures if Gradle does not completely capture all outputs for a given task. Wrongly declared task inputs can lead to cache misses especially when containing volatile data or absolute paths. (See the section called "Task inputs and outputs" on what should be declared as inputs and outputs.)

NOTE	The task path is *not* an input to the build cache key. This means that tasks with different task paths can re-use each other's outputs as long as Gradle determines that executing them yields the same result.

In order to ensure that the inputs and outputs are properly declared use integration tests (for example using TestKit) to check that a task produces the same outputs for identical inputs and captures all output files for the task. We suggest adding tests to ensure that the task inputs are relocatable, i.e. that the task can be loaded from the cache into a different build directory (see @PathSensitive).

In order to handle volatile inputs for your tasks consider configuring input normalization.

Marking tasks as non-cacheable by default

There are certain tasks that don't benefit from using the build cache. One example is a task that only moves data around the file system, like a `Copy` task. You can signify that a task is not to be cached by adding the `@DisableCachingByDefault` annotation to it. You can also give a human-readable reason for not caching the task by default. The annotation can be used on its own, or together with `@CacheableTask`.

NOTE	This annotation is only for documenting the reason behind not caching the task by default. Build logic can override this decision via the runtime API (see below).

Enable caching of non-cacheable tasks

As we have seen, built-in tasks, or tasks provided by plugins, are cacheable if their class is annotated with the `Cacheable` annotation. But what if you want to make cacheable a task whose class is not cacheable? Let's take a concrete example: your build script uses a generic `NpmTask` task to create a JavaScript bundle by delegating to NPM (and running `npm run bundle`). This process is similar to a complex compilation task, but `NpmTask` is too generic to be cacheable by default: it just takes arguments and runs npm with those arguments.

The inputs and outputs of this task are simple to figure out. The inputs are the directory containing the JavaScript files, and the NPM configuration files. The output is the bundle file generated by this task.

Using annotations

We create a subclass of the `NpmTask` and use annotations to declare the inputs and outputs.

When possible, it is better to use delegation instead of creating a subclass. That is the case for the built in `JavaExec`, `Exec`, `Copy` and `Sync` tasks, which have a method on `Project` to do the actual work.

If you're a modern JavaScript developer, you know that bundling can be quite long, and is worth caching. To achieve that, we need to tell Gradle that it's allowed to cache the output of that task, using the @CacheableTask annotation.

This is sufficient to make the task cacheable on your own machine. However, input files are identified by default by their absolute path. So if the cache needs to be shared between several developers or machines using different paths, that won't work as expected. So we also need to set the path sensitivity. In this case, the relative path of the input files can be used to identify them.

Note that it is possible to override property annotations from the base class by overriding the getter of the base class and annotating that method.

Example 49. Custom cacheable BundleTask

build.gradle

```
@CacheableTask                                          ①
abstract class BundleTask extends NpmTask {

    @Override @Internal                                 ②
    ListProperty<String> getArgs() {
        super.getArgs()
    }

    @InputDirectory
    @SkipWhenEmpty
    @PathSensitive(PathSensitivity.RELATIVE)            ③
    abstract DirectoryProperty getScripts()

    @InputFiles
    @PathSensitive(PathSensitivity.RELATIVE)            ④
    abstract ConfigurableFileCollection getConfigFiles()

    @OutputFile
    abstract RegularFileProperty getBundle()

    BundleTask() {
        args.addAll("run", "bundle")
        bundle.set(project.layout.buildDirectory.file("bundle.js"))
        scripts.set(project.layout.projectDirectory.dir("scripts"))
        configFiles.from(project.layout.projectDirectory.file("package.json"
))
        configFiles.from(project.layout.projectDirectory.file("package-
lock.json"))
    }
}

tasks.register('bundle', BundleTask)
```

build.gradle.kts

```
@CacheableTask                                          ①
abstract class BundleTask : NpmTask() {

    @get:Internal                                       ②
    override val args
        get() = super.args


    @get:InputDirectory
    @get:SkipWhenEmpty
    @get:PathSensitive(PathSensitivity.RELATIVE)        ③
    abstract val scripts: DirectoryProperty

    @get:InputFiles
    @get:PathSensitive(PathSensitivity.RELATIVE)        ④
    abstract val configFiles: ConfigurableFileCollection

    @get:OutputFile
    abstract val bundle: RegularFileProperty

    init {
        args.addAll("run", "bundle")
        bundle.set(project.layout.buildDirectory.file("bundle.js"))
        scripts.set(project.layout.projectDirectory.dir("scripts"))

configFiles.from(project.layout.projectDirectory.file("package.json"))
        configFiles.from(project.layout.projectDirectory.file("package-
lock.json"))
    }
}

tasks.register<BundleTask>("bundle")
```

- (1) Add `@CacheableTask` to enable caching for the task.
- (2) Override the getter of a property of the base class to change the input annotation to `@Internal`.
- (3) (4) Declare the path sensitivity.

Using the runtime API

If for some reason you cannot create a new custom task class, it is also possible to make a task cacheable using the runtime API to declare the inputs and outputs.

For enabling caching for the task you need to use the TaskOutputs.cacheIf() method.

The declarations via the runtime API have the same effect as the annotations described above. Note that you cannot override file inputs and outputs via the runtime API. Input properties can be overridden by specifying the same property name.

Example 50. Make the bundle task cacheable

build.gradle

```
tasks.register('bundle', NpmTask) {
    args = ['run', 'bundle']

    outputs.cacheIf { true }

    inputs.dir(file("scripts"))
        .withPropertyName("scripts")
        .withPathSensitivity(PathSensitivity.RELATIVE)

    inputs.files("package.json", "package-lock.json")
        .withPropertyName("configFiles")
        .withPathSensitivity(PathSensitivity.RELATIVE)

    outputs.file("$buildDir/bundle.js")
        .withPropertyName("bundle")
}
```

build.gradle.kts

```
tasks.register<NpmTask>("bundle") {
    args.set(listOf("run", "bundle"))

    outputs.cacheIf { true }

    inputs.dir(file("scripts"))
        .withPropertyName("scripts")
        .withPathSensitivity(PathSensitivity.RELATIVE)

    inputs.files("package.json", "package-lock.json")
        .withPropertyName("configFiles")
        .withPathSensitivity(PathSensitivity.RELATIVE)

    outputs.file("$buildDir/bundle.js")
        .withPropertyName("bundle")
}
```

Configure the Build Cache

You can configure the build cache by using the Settings.buildCache(org.gradle.api.Action) block in `settings.gradle`.

Gradle supports a `local` and a `remote` build cache that can be configured separately. When both build caches are enabled, Gradle tries to load build outputs from the local build cache first, and then tries the remote build cache if no build outputs are found. If outputs are found in the remote cache, they are also stored in the local cache, so next time they will be found locally. Gradle stores ("pushes") build outputs in any build cache that is enabled and has BuildCache.isPush() set to `true`.

By default, the local build cache has push enabled, and the remote build cache has push disabled.

The local build cache is pre-configured to be a DirectoryBuildCache and enabled by default. The remote build cache can be configured by specifying the type of build cache to connect to (BuildCacheConfiguration.remote(java.lang.Class)).

Built-in local build cache

The built-in local build cache, DirectoryBuildCache, uses a directory to store build cache artifacts. By default, this directory resides in the Gradle user home directory, but its location is configurable.

Gradle will periodically clean-up the local cache directory by removing entries that have not been used recently to conserve disk space. Note that cache entries are cleaned-up regardless of the project they were produced by.

For more details on the configuration options refer to the DSL documentation of DirectoryBuildCache. Here is an example of the configuration.

Example 51. Configure the local cache

settings.gradle

```
buildCache {
    local {
        directory = new File(rootDir, 'build-cache')
        removeUnusedEntriesAfterDays = 30
    }
}
```

settings.gradle.kts

```
buildCache {
    local {
        directory = File(rootDir, "build-cache")
        removeUnusedEntriesAfterDays = 30
    }
}
```

Remote HTTP build cache

HttpBuildCache provides the ability read to and write from a remote cache via HTTP.

With the following configuration, the local build cache will be used for storing build outputs while the local and the remote build cache will be used for retrieving build outputs.

Example 52. Load from HttpBuildCache

settings.gradle

```
buildCache {
    remote(HttpBuildCache) {
        url = 'https://example.com:8123/cache/'
    }
}
```

settings.gradle.kts

```
buildCache {
    remote<HttpBuildCache> {
        url = uri("https://example.com:8123/cache/")
    }
}
```

When attempting to load an entry, a `GET` request is made to `https://example.com:8123/cache/«cache-key»`. The response must have a `2xx` status and the cache entry as the body, or a `404 Not Found` status if the entry does not exist.

When attempting to store an entry, a `PUT` request is made to `https://example.com:8123/cache/«cache-key»`. Any `2xx` response status is interpreted as success. A `413 Payload Too Large` response may be returned to indicate that the payload is larger than the server will accept, which will not be treated as an error.

Specifying access credentials

HTTP Basic Authentication is supported, with credentials being sent preemptively.

Example 53. Specifying access credentials

settings.gradle

```
buildCache {
    remote(HttpBuildCache) {
        url = 'https://example.com:8123/cache/'
        credentials {
            username = 'build-cache-user'
            password = 'some-complicated-password'
        }
    }
}
```

settings.gradle.kts

```
buildCache {
    remote<HttpBuildCache> {
        url = uri("https://example.com:8123/cache/")
        credentials {
            username = "build-cache-user"
            password = "some-complicated-password"
        }
    }
}
```

Redirects

`3xx` redirecting responses will be followed automatically.

Servers must take care when redirecting `PUT` requests as only `307` and `308` redirect responses will be followed with a `PUT` request. All other redirect responses will be followed with a `GET` request, as per RFC 7231, without the entry payload as the body.

Network error handling

Requests that fail during request transmission, after having established a TCP connection, will be retried automatically.

This prevents temporary problems, such as connection drops, read or write timeouts, and low level network failures such as a connection resets, causing cache operations to fail and disabling the remote cache for the remainder of the build.

Requests will be retried up to 3 times. If the problem persists, the cache operation will fail and the remote cache will be disabled for the remainder of the build.

Using SSL

By default, use of HTTPS requires the server to present a certificate that is trusted by the build's Java runtime. If your server's certificate is not trusted, you can:

1. Update the trust store of your Java runtime to allow it to be trusted
2. Change the build environment to use an alternative trust store for the build runtime
3. Disable the requirement for a trusted certificate

The trust requirement can be disabled by setting HttpBuildCache.isAllowUntrustedServer() to `true`. Enabling this option is a security risk, as it allows any cache server to impersonate the intended server. It should only be used as a temporary measure or in very tightly controlled network environments.

Example 54. Allow untrusted cache server

settings.gradle

```
buildCache {
    remote(HttpBuildCache) {
        url = 'https://example.com:8123/cache/'
        allowUntrustedServer = true
    }
}
```

settings.gradle.kts

```
buildCache {
    remote<HttpBuildCache> {
        url = uri("https://example.com:8123/cache/")
        isAllowUntrustedServer = true
    }
}
```

HTTP expect-continue

Use of HTTP Expect-Continue can be enabled. This causes upload requests to happen in two parts: first a check whether a body would be accepted, then transmission of the body if the server indicates it will accept it.

This is useful when uploading to cache servers that routinely redirect or reject upload requests, as it avoids uploading the cache entry just to have it rejected (e.g. the cache entry is larger than the cache will allow) or redirected. This additional check incurs extra latency when the server accepts the request, but reduces latency when the request is rejected or redirected.

Not all HTTP servers and proxies reliably implement Expect-Continue. Be sure to check that your cache server does support it before enabling.

To enable, set HttpBuildCache.isUseExpectContinue() to `true`.

Example 55. Use Expect-Continue

settings.gradle

```
buildCache {
    remote(HttpBuildCache) {
        url = 'https://example.com:8123/cache/'
        useExpectContinue = true
    }
}
```

settings.gradle.kts

```
buildCache {
    remote<HttpBuildCache> {
        url = uri("https://example.com:8123/cache/")
        isUseExpectContinue = true
    }
}
```

Configuration use cases

The recommended use case for the remote build cache is that your continuous integration server populates it from clean builds while developers only load from it. The configuration would then look as follows.

Example 56. Recommended setup for CI push use case

settings.gradle

```
boolean isCiServer = System.getenv().containsKey("CI")

buildCache {
    remote(HttpBuildCache) {
        url = 'https://example.com:8123/cache/'
        push = isCiServer
    }
}
```

settings.gradle.kts

```
val isCiServer = System.getenv().containsKey("CI")

buildCache {
    remote<HttpBuildCache> {
        url = uri("https://example.com:8123/cache/")
        isPush = isCiServer
    }
}
```

It is also possible to configure the build cache from an init script, which can be used from the command line, added to your Gradle user home or be a part of your custom Gradle distribution.

Example 57. Init script to configure the build cache

init.gradle

```
gradle.settingsEvaluated { settings ->
    settings.buildCache {
        // vvv Your custom configuration goes here
        remote(HttpBuildCache) {
            url = 'https://example.com:8123/cache/'
        }
        // ^^^ Your custom configuration goes here
    }
}
```

init.gradle.kts

```
gradle.settingsEvaluated {
    buildCache {
        // vvv Your custom configuration goes here
        remote<HttpBuildCache> {
            url = uri("https://example.com:8123/cache/")
        }
        // ^^^ Your custom configuration goes here
    }
}
```

Build cache, composite builds and `buildSrc`

Gradle's composite build feature allows including other complete Gradle builds into another. Such included builds will inherit the build cache configuration from the top level build, regardless of whether the included builds define build cache configuration themselves or not.

The build cache configuration present for any included build is effectively ignored, in favour of the top level build's configuration. This also applies to any `buildSrc` projects of any included builds.

The `buildSrc` directory is treated as an included build, and as such it inherits the build cache configuration from the top-level build.

NOTE	This configuration precedence does not apply to plugin builds included through `pluginManagement` as these are loaded *before* the cache configuration itself.

How to set up an HTTP build cache backend

Gradle provides a Docker image for a build cache node, which can connect with Gradle Enterprise for centralized management. The cache node can also be used without a Gradle Enterprise

installation with restricted functionality.

Implement your own Build Cache

Using a different build cache backend to store build outputs (which is not covered by the built-in support for connecting to an HTTP backend) requires implementing your own logic for connecting to your custom build cache backend. To this end, custom build cache types can be registered via BuildCacheConfiguration.registerBuildCacheService(java.lang.Class, java.lang.Class).

Gradle Enterprise includes a high-performance, easy to install and operate, shared build cache backend.

Authoring Gradle Builds

Build Script Basics

This chapter introduces you to the basics of writing Gradle build scripts. It uses toy examples to explain basic functionality of Gradle, which is helpful to get an understanding of the basic concepts. Especially if you move to Gradle from other build tools like Ant and want to understand differences and advantages.

However, to get started with a standard project setup, you don't even need to go into these concepts in detail. Instead, you can have a quick hands-on introduction, through our step-by-step samples.

Projects, plugins and tasks

Every Gradle build is made up of one or more *projects*. What a project represents depends on what it is that you are doing with Gradle. For example, a project might represent a library JAR or a web application. It might represent a distribution ZIP assembled from the JARs produced by other projects. A project does not necessarily represent a thing to be built. It might represent a thing to be done, such as deploying your application to staging or production environments. Don't worry if this seems a little vague for now. Gradle's build-by-convention support adds a more concrete definition for what a project is.

The work that Gradle can do on a project is defined by one or more *tasks*. A task represents some atomic piece of work which a build performs. This might be compiling some classes, creating a JAR, generating Javadoc, or publishing some archives to a repository.

Typically, tasks are provided by applying a plugin so that you do not have to define them yourself. Still, to give you an idea of what a task is, we will look at defining some simple tasks in a build with one project in this chapter.

Hello world

You run a Gradle build using the `gradle` command. The `gradle` command looks for a file called `build.gradle` in the current directory.[1] We call this `build.gradle` file a *build script*, although strictly speaking it is a build configuration script, as we will see later. The build script defines a project and its tasks.

To try this out, create the following build script named `build.gradle`.

You run a Gradle build using the `gradle` command. The `gradle` command looks for a file called `build.gradle.kts` in the current directory.[2] We call this `build.gradle.kts` file a *build script*, although strictly speaking it is a build configuration script, as we will see later. The build script defines a project and its tasks.

To try this out, create the following build script named `build.gradle.kts`.

Example 58. Your first build script

build.gradle

```
tasks.register('hello') {
    doLast {
        println 'Hello world!'
    }
}
```

build.gradle.kts

```
tasks.register("hello") {
    doLast {
        println("Hello world!")
    }
}
```

In a command-line shell, move to the containing directory and execute the build script with `gradle -q hello`:

TIP

What does `-q` do?

Most of the examples in this user guide are run with the `-q` command-line option. This suppresses Gradle's log messages, so that only the output of the tasks is shown. This keeps the example output in this user guide a little clearer. You don't need to use this option if you don't want to. See Logging for more details about the command-line options which affect Gradle's output.

Example 59. Execution of a build script

Output of `gradle -q hello`

```
> gradle -q hello
Hello world!
```

What's going on here? This build script defines a single task, called `hello`, and adds an action to it. When you run `gradle hello`, Gradle executes the `hello` task, which in turn executes the action you've provided. The action is simply a block containing some code to execute.

If you think this looks similar to Ant's targets, you would be right. Gradle tasks are the equivalent to Ant targets, but as you will see, they are much more powerful. We have used a different terminology than Ant as we think the word *task* is more expressive than the word *target*. Unfortunately this introduces a terminology clash with Ant, as Ant calls its commands, such as

`javac` or `copy`, tasks. So when we talk about tasks, we *always* mean Gradle tasks, which are the equivalent to Ant's targets. If we talk about Ant tasks (Ant commands), we explicitly say *Ant task*.

Build scripts are code

Gradle's build scripts give you the full power of Groovy and Kotlin. As an appetizer, have a look at this:

Example 60. Using Groovy or Kotlin in Gradle's tasks

build.gradle

```
tasks.register('upper') {
    doLast {
        String someString = 'mY_nAmE'
        println "Original: $someString"
        println "Upper case: ${someString.toUpperCase()}"
    }
}
```

build.gradle.kts

```
tasks.register("upper") {
    doLast {
        val someString = "mY_nAmE"
        println("Original: $someString")
        println("Upper case: ${someString.toUpperCase()}")
    }
}
```

Output of `gradle -q upper`

```
> gradle -q upper
Original: mY_nAmE
Upper case: MY_NAME
```

or

Example 61. Using Groovy or Kotlin in Gradle's tasks

build.gradle

```
tasks.register('count') {
    doLast {
        4.times { print "$it " }
    }
}
```

build.gradle.kts

```
tasks.register("count") {
    doLast {
        repeat(4) { print("$it ") }
    }
}
```

Output of `gradle -q count`

```
> gradle -q count
0 1 2 3
```

Task dependencies

As you probably have guessed, you can declare tasks that depend on other tasks.

Example 62. Declaration of task that depends on other task

build.gradle

```
tasks.register('hello') {
    doLast {
        println 'Hello world!'
    }
}
tasks.register('intro') {
    dependsOn tasks.hello
    doLast {
        println "I'm Gradle"
    }
}
```

build.gradle.kts

```
tasks.register("hello") {
    doLast {
        println("Hello world!")
    }
}
tasks.register("intro") {
    dependsOn("hello")
    doLast {
        println("I'm Gradle")
    }
}
```

Output of `gradle -q intro`

```
> gradle -q intro
Hello world!
I'm Gradle
```

To add a dependency, the corresponding task does not need to exist.

Example 63. Lazy dependsOn - the other task does not exist (yet)

build.gradle

```
tasks.register('taskX') {
    dependsOn 'taskY'
    doLast {
        println 'taskX'
    }
}
tasks.register('taskY') {
    doLast {
        println 'taskY'
    }
}
```

build.gradle.kts

```
tasks.register("taskX") {
    dependsOn("taskY")
    doLast {
        println("taskX")
    }
}
tasks.register("taskY") {
    doLast {
        println("taskY")
    }
}
```

Output of `gradle -q taskX`

```
> gradle -q taskX
taskY
taskX
```

The dependency of `taskX` to `taskY` may be declared before `taskY` is defined. Task dependencies are discussed in more detail in Adding dependencies to a task.

Flexible task registration

The power of Groovy or Kotlin can be used for more than defining what a task does. For example, you can use it to register multiple tasks of the same type in a loop.

Example 64. Flexible registration of a task

build.gradle

```
4.times { counter ->
    tasks.register("task$counter") {
        doLast {
            println "I'm task number $counter"
        }
    }
}
```

build.gradle.kts

```
repeat(4) { counter ->
    tasks.register("task$counter") {
        doLast {
            println("I'm task number $counter")
        }
    }
}
```

Output of `gradle -q task1`

```
> gradle -q task1
I'm task number 1
```

Manipulating existing tasks

Once tasks are registered, they can be accessed via an *API*. For instance, you could use this to dynamically add dependencies to a task, at runtime. Ant doesn't allow anything like this.

Example 65. Accessing a task via API - adding a dependency

build.gradle

```
4.times { counter ->
    tasks.register("task$counter") {
        doLast {
            println "I'm task number $counter"
        }
    }
}
tasks.named('task0') { dependsOn('task2', 'task3') }
```

build.gradle.kts

```
repeat(4) { counter ->
    tasks.register("task$counter") {
        doLast {
            println("I'm task number $counter")
        }
    }
}
tasks.named("task0") { dependsOn("task2", "task3") }
```

Output of `gradle -q task0`

```
> gradle -q task0
I'm task number 2
I'm task number 3
I'm task number 0
```

Or you can add behavior to an existing task.

Example 66. Accessing a task via API - adding behaviour

build.gradle

```
tasks.register('hello') {
    doLast {
        println 'Hello Earth'
    }
}
tasks.named('hello') {
    doFirst {
        println 'Hello Venus'
    }
}
tasks.named('hello') {
    doLast {
        println 'Hello Mars'
    }
}
tasks.named('hello') {
    doLast {
        println 'Hello Jupiter'
    }
}
```

build.gradle.kts

```
tasks.register("hello") {
    doLast {
        println("Hello Earth")
    }
}
tasks.named("hello") {
    doFirst {
        println("Hello Venus")
    }
}
tasks.named("hello") {
    doLast {
        println("Hello Mars")
    }
}
tasks.named("hello") {
    doLast {
        println("Hello Jupiter")
    }
}
```

Output of `gradle -q hello`

```
> gradle -q hello
Hello Venus
Hello Earth
Hello Mars
Hello Jupiter
```

The calls `doFirst` and `doLast` can be executed multiple times. They add an action to the beginning or the end of the task's actions list. When the task executes, the actions in the action list are executed in order.

Using Ant Tasks

Ant tasks are first-class citizens in Gradle. Gradle provides excellent integration for Ant tasks by simply relying on Groovy. Groovy is shipped with the fantastic `AntBuilder`. Using Ant tasks from Gradle is as convenient and more powerful than using Ant tasks from a `build.xml` file. And it is usable from Kotlin too. From the example below, you can learn how to execute Ant tasks and how to access Ant properties:

Example 67. Using AntBuilder to execute ant.loadfile target

build.gradle

```
tasks.register('loadfile') {
    doLast {
        def files = file('./antLoadfileResources').listFiles().sort()
        files.each { File file ->
            if (file.isFile()) {
                ant.loadfile(srcFile: file, property: file.name)
                println " *** $file.name ***"
                println "${ant.properties[file.name]}"
            }
        }
    }
}
```

build.gradle.kts

```
tasks.register("loadfile") {
    doLast {
        val files = file("./antLoadfileResources").listFiles().sorted()
        files.forEach { file ->
            if (file.isFile) {
                ant.withGroovyBuilder {
                    "loadfile"("srcFile" to file, "property" to file.name)
                }
                println(" *** ${file.name} ***")
                println("${ant.properties[file.name]}")
            }
        }
    }
}
```

Output of `gradle -q loadfile`

```
> gradle -q loadfile
 *** agile.manifesto.txt ***
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration  over contract negotiation
Responding to change over following a plan
 *** gradle.manifesto.txt ***
Make the impossible possible, make the possible easy and make the easy elegant.
(inspired by Moshe Feldenkrais)
```

There is lots more you can do with Ant in your build scripts. You can find out more in Ant.

Using methods

Gradle scales in how you can organize your build logic. The first level of organizing your build logic for the example above, is extracting a method.

Example 68. Using methods to organize your build logic

build.gradle

```
tasks.register('checksum') {
    doLast {
        fileList('./antLoadfileResources').each { File file ->
            ant.checksum(file: file, property: "cs_$file.name")
            println "$file.name Checksum: ${ant.properties["cs_$file.name"]}"
        }
    }
}

tasks.register('loadfile') {
    doLast {
        fileList('./antLoadfileResources').each { File file ->
            ant.loadfile(srcFile: file, property: file.name)
            println "I'm fond of $file.name"
        }
    }
}

File[] fileList(String dir) {
    file(dir).listFiles({file -> file.isFile() } as FileFilter).sort()
}
```

build.gradle.kts

```
tasks.register("checksum") {
    doLast {
        fileList("./antLoadfileResources").forEach { file ->
            ant.withGroovyBuilder {
                "checksum"("file" to file, "property" to "cs_${file.name}")
            }
            println("$file.name Checksum:
${ant.properties["cs_${file.name}"]}")
        }
    }
}

tasks.register("loadfile") {
    doLast {
        fileList("./antLoadfileResources").forEach { file ->
            ant.withGroovyBuilder {
                "loadfile"("srcFile" to file, "property" to file.name)
            }
            println("I'm fond of ${file.name}")
        }
    }
}

fun fileList(dir: String): List<File> =
    file(dir).listFiles { file: File -> file.isFile }.sorted()
```

Output of `gradle -q loadfile`

```
> gradle -q loadfile
I'm fond of agile.manifesto.txt
I'm fond of gradle.manifesto.txt
```

Later you will see that such methods can be shared among subprojects in multi-project builds. If your build logic becomes more complex, Gradle offers you other very convenient ways to organize it. We have devoted a whole chapter to this. See Organizing Gradle Projects.

Default tasks

Gradle allows you to define one or more default tasks that are executed if no other tasks are specified.

Example 69. Defining a default task

build.gradle

```
defaultTasks 'clean', 'run'

tasks.register('clean') {
    doLast {
        println 'Default Cleaning!'
    }
}

tasks.register('run') {
    doLast {
        println 'Default Running!'
    }
}

tasks.register('other') {
    doLast {
        println "I'm not a default task!"
    }
}
```

build.gradle.kts

```
defaultTasks("clean", "run")

tasks.register("clean") {
    doLast {
        println("Default Cleaning!")
    }
}

tasks.register("run") {
    doLast {
        println("Default Running!")
    }
}

tasks.register("other") {
    doLast {
        println("I'm not a default task!")
    }
}
```

Output of `gradle -q`

```
> gradle -q
Default Cleaning!
Default Running!
```

This is equivalent to running `gradle clean run`. In a multi-project build every subproject can have its own specific default tasks. If a subproject does not specify default tasks, the default tasks of the parent project are used (if defined).

External dependencies for the build script

NOTE	Instead of manipulating the script classpath directly, it is recommended to apply plugins that come with their own classpath. For custom build logic, the recommendation is to use a custom plugin.

If your build script needs to use external libraries, you can add them to the script's classpath in the build script itself. You do this using the `buildscript()` method, passing in a block which declares the build script classpath.

Example 70. Declaring external dependencies for the build script

build.gradle

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'commons-codec', name: 'commons-codec', version:
'1.2'
    }
}
```

build.gradle.kts

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        "classpath"(group = "commons-codec", name = "commons-codec", version
= "1.2")
    }
}
```

The block passed to the `buildscript()` method configures a ScriptHandler instance. You declare the build script classpath by adding dependencies to the `classpath` configuration. This is the same way you declare, for example, the Java compilation classpath. You can use any of the dependency types except project dependencies.

Having declared the build script classpath, you can use the classes in your build script as you would any other classes on the classpath. The following example adds to the previous example, and uses classes from the build script classpath.

Example 71. A build script with external dependencies

build.gradle

```
import org.apache.commons.codec.binary.Base64

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'commons-codec', name: 'commons-codec', version:
'1.2'
    }
}

tasks.register('encode') {
    doLast {
        def byte[] encodedString = new Base64().encode('hello world\n'
.getBytes())
        println new String(encodedString)
    }
}
```

build.gradle.kts

```
import org.apache.commons.codec.binary.Base64

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        "classpath"(group = "commons-codec", name = "commons-codec", version
= "1.2")
    }
}

tasks.register("encode") {
    doLast {
        val encodedString = Base64().encode("hello world\n".toByteArray())
        println(String(encodedString))
    }
}
```

Output of `gradle -q encode`

```
> gradle -q encode
aGVsbG8gd29ybGQK
```

For multi-project builds, the dependencies declared with a project's `buildscript()` method are available to the build scripts of all its sub-projects.

Build script dependencies may be Gradle plugins. Please consult Using Gradle Plugins for more information on Gradle plugins.

Every project automatically has a `buildEnvironment` task of type BuildEnvironmentReportTask that can be invoked to report on the resolution of the build script dependencies.

Further Reading

This chapter only scratched the surface with what's possible. Here are some other topics that may be interesting:

- Authoring maintainable build scripts
- Organizing your Gradle projects
- Writing Custom tasks

Authoring Tasks

In the introductory tutorial you learned how to create simple tasks. You also learned how to add additional behavior to these tasks later on, and you learned how to create dependencies between tasks. This was all about simple tasks, but Gradle takes the concept of tasks further. Gradle supports tasks that have their own properties and methods. Such tasks are either provided by you or built into Gradle.

Task outcomes

When Gradle executes a task, it can label the task with different outcomes in the console UI and via the Tooling API. These labels are based on if a task has actions to execute, if it should execute those actions, if it did execute those actions and if those actions made any changes.

`(no label)` or `EXECUTED`

Task executed its actions.

- Task has actions and Gradle has determined they should be executed as part of a build.
- Task has no actions and some dependencies, and any of the dependencies are executed. See also Lifecycle Tasks.

`UP-TO-DATE`

Task's outputs did not change.

- Task has outputs and inputs and they have not changed. See Incremental Builds.
- Task has actions, but the task tells Gradle it did not change its outputs.
- Task has no actions and some dependencies, but all of the dependencies are up-to-date, skipped or from cache. See also Lifecycle Tasks.
- Task has no actions and no dependencies.

`FROM-CACHE`

Task's outputs could be found from a previous execution.

- Task has outputs restored from the build cache. See Build Cache.

`SKIPPED`

Task did not execute its actions.

- Task has been explicitly excluded from the command-line. See Excluding tasks from execution.
- Task has an `onlyIf` predicate return false. See Using a predicate.

`NO-SOURCE`

Task did not need to execute its actions.

- Task has inputs and outputs, but no sources. For example, source files are `.java` files for JavaCompile.

Defining tasks

We have already seen how to define tasks using strings for task names in this chapter. There are a few variations on this style, which you may need to use in certain situations.

NOTE	The task configuration APIs are described in more detail in the task configuration avoidance chapter.

Example 72. Defining tasks using strings for task names

build.gradle

```
tasks.register('hello') {
    doLast {
        println 'hello'
    }
}

tasks.register('copy', Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

build.gradle.kts

```
tasks.register("hello") {
    doLast {
        println("hello")
    }
}

tasks.register<Copy>("copy") {
    from(file("srcDir"))
    into(buildDir)
}
```

We add the tasks to the `tasks` collection. Have a look at TaskContainer for more variations of the `register()` method.

In the Kotlin DSL there is also a specific delegated properties syntax that is useful if you need the registered task for further reference.

Example 73. Assigning tasks to variables with DSL specific syntax

build.gradle

```
// Assigning registered tasks to a variable in Groovy

def hello = tasks.register('hello') {
    doLast {
        println 'hello'
    }
}

def copy = tasks.register('copy', Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

build.gradle.kts

```
// Using Kotlin delegated properties

val hello by tasks.registering {
    doLast {
        println("hello")
    }
}

val copy by tasks.registering(Copy::class) {
    from(file("srcDir"))
    into(buildDir)
}
```

WARNING If you look at the API of the *tasks* container you may notice that there are additional methods to *create* tasks. The use of these methods is discouraged and will be deprecated in future versions. These methods only exist for backward compatibility as they were introduced before task configuration avoidance was added to Gradle.

Locating tasks

You often need to locate the tasks that you have defined in the build file, for example, to configure them or use them for dependencies. There are a number of ways of doing this. Firstly, just like with defining tasks there are language specific syntaxes for the Groovy and Kotlin DSL:

In general, tasks are available through the `tasks` collection. You should use of the methods that

return a *task provider* – `register()` or `named()` – to make sure you do not break task configuration avoidance.

Example 74. Accessing tasks via tasks collection

build.gradle

```
tasks.register('hello')
tasks.register('copy', Copy)

println tasks.named('hello').get().name

println tasks.named('copy').get().destinationDir
```

build.gradle.kts

```
tasks.register("hello")
tasks.register<Copy>("copy")

println(tasks.named("hello").get().name) // or just 'tasks.hello' if the task
was added by a plugin

println(tasks.named<Copy>("copy").get().destinationDir)
```

Tasks of a specific type can also be accessed by using the `tasks.withType()` method. This enables to easily avoid duplication of code and reduce redundancy.

Example 75. Accessing tasks by their type

build.gradle

```
tasks.withType(Tar).configureEach {
    enabled = false
}

tasks.register('test') {
    dependsOn tasks.withType(Copy)
}
```

build.gradle.kts

```
tasks.withType<Tar>().configureEach {
    enabled = false
}

tasks.register("test") {
    dependsOn(tasks.withType<Copy>())
}
```

WARNING	The following shows how to access a task by path. This is not a recommended practice anymore as it breaks task configuration avoidance and project isolation. Dependencies between projects should be declared as dependencies.

You can access tasks from any project using the task's path using the `tasks.getByPath()` method. You can call the `getByPath()` method with a task name, or a relative path, or an absolute path.

Example 76. Accessing tasks by path

project-a/build.gradle

```
tasks.register('hello')
```

build.gradle

```
tasks.register('hello')

println tasks.getByPath('hello').path
println tasks.getByPath(':hello').path
println tasks.getByPath('project-a:hello').path
println tasks.getByPath(':project-a:hello').path
```

project-a/build.gradle.kts

```
tasks.register("hello")
```

build.gradle.kts

```
tasks.register("hello")

println(tasks.getByPath("hello").path)
println(tasks.getByPath(":hello").path)
println(tasks.getByPath("project-a:hello").path)
println(tasks.getByPath(":project-a:hello").path)
```

Output of `gradle -q hello`

```
> gradle -q hello
:hello
:hello
:project-a:hello
:project-a:hello
```

Have a look at TaskContainer for more options for locating tasks.

Configuring tasks

As an example, let's look at the `Copy` task provided by Gradle. To register a `Copy` task for your build, you can declare in your build script:

Example 77. Registering a copy task

build.gradle

```
tasks.register('myCopy', Copy)
```

build.gradle.kts

```
tasks.register<Copy>("myCopy")
```

This registers a copy task with no default behavior. The task can be configured using its API (see Copy). The following examples show several different ways to achieve the same configuration.

Just to be clear, realize that the name of this task is `myCopy`, but it is of *type* `Copy`. You can have multiple tasks of the same *type*, but with different names. You'll find this gives you a lot of power to implement cross-cutting concerns across all tasks of a particular type.

Example 78. Configuring a task

build.gradle

```
tasks.named('myCopy') {
    from 'resources'
    into 'target'
    include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

build.gradle.kts

```
tasks.named<Copy>("myCopy") {
    from("resources")
    into("target")
    include("**/*.txt", "**/*.xml", "**/*.properties")
}
```

You can also store the task reference in a variable and use to configure the task further at a later point in the script.

Example 79. Retrieve a task reference and use it to configuring the task

build.gradle

```
// Configure task through a task provider
def myCopy = tasks.named('myCopy')  {
    from 'resources'
    into 'target'
}
myCopy.configure {
    include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

build.gradle.kts

```
// Configure task using Kotlin delegated properties and a lambda
val myCopy by tasks.existing(Copy::class) {
    from("resources")
    into("target")
}
myCopy {
    include("**/*.txt", "**/*.xml", "**/*.properties")
}
```

Have a look at TaskContainer for more options for configuring tasks.

TIP If you use the *Kotlin DSL* and the task you want to configure was added by a plugin, you can use a convenient accessor for the task. That is, instead of `tasks.named("test")` you can just write `tasks.test`.

You can also use a configuration block when you define a task.

Example 80. Defining a task with a configuration block

build.gradle

```
tasks.register('copy', Copy) {
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

build.gradle.kts

```
tasks.register<Copy>("copy") {
   from("resources")
   into("target")
   include("**/*.txt", "**/*.xml", "**/*.properties")
}
```

TIP

Don't forget about the build phases

A task has both configuration and actions. When using the `doLast`, you are simply using a shortcut to define an action. Code defined in the configuration section of your task will get executed during the configuration phase of the build regardless of what task was targeted. See Build Lifecycle for more details about the build lifecycle.

Passing arguments to a task constructor

As opposed to configuring the mutable properties of a `Task` after creation, you can pass argument values to the `Task` class's constructor. In order to pass values to the `Task` constructor, you must annotate the relevant constructor with `@javax.inject.Inject`.

Example 81. Task class with `@Inject` constructor

build.gradle

```
abstract class CustomTask extends DefaultTask {
    final String message
    final int number

    @Inject
    CustomTask(String message, int number) {
        this.message = message
        this.number = number
    }
}
```

build.gradle.kts

```
abstract class CustomTask @Inject constructor(
    private val message: String,
    private val number: Int
) : DefaultTask()
```

You can then create a task, passing the constructor arguments at the end of the parameter list.

Example 82. Registering a task with constructor arguments using TaskContainer

build.gradle

```
tasks.register('myTask', CustomTask, 'hello', 42)
```

build.gradle.kts

```
tasks.register<CustomTask>("myTask", "hello", 42)
```

NOTE It's recommended to use the Task Configuration Avoidance APIs to improve configuration time.

In all circumstances, the values passed as constructor arguments must be non-null. If you attempt to pass a `null` value, Gradle will throw a `NullPointerException` indicating which runtime value is `null`.

Adding dependencies to a task

There are several ways you can define the dependencies of a task. In Task dependencies you were introduced to defining dependencies using task names. Task names can refer to tasks in the same project as the task, or to tasks in other projects. To refer to a task in another project, you prefix the name of the task with the path of the project it belongs to. The following is an example which adds a dependency from `project-a:taskX` to `project-b:taskY`:

Example 83. Adding dependency on task from another project

build.gradle

```
project('project-a') {
    tasks.register('taskX')  {
        dependsOn ':project-b:taskY'
        doLast {
            println 'taskX'
        }
    }
}

project('project-b') {
    tasks.register('taskY') {
        doLast {
            println 'taskY'
        }
    }
}
```

build.gradle.kts

```
project("project-a") {
    tasks.register("taskX") {
        dependsOn(":project-b:taskY")
        doLast {
            println("taskX")
        }
    }
}

project("project-b") {
    tasks.register("taskY") {
        doLast {
            println("taskY")
        }
    }
}
```

Output of `gradle -q taskX`

```
> gradle -q taskX
taskY
taskX
```

Instead of using a task name, you can define a dependency using a `TaskProvider` object, as shown in this example:

Example 84. Adding dependency using task provider object

build.gradle

```
def taskX = tasks.register('taskX') {
    doLast {
        println 'taskX'
    }
}

def taskY = tasks.register('taskY') {
    doLast {
        println 'taskY'
    }
}

taskX.configure {
    dependsOn taskY
}
```

build.gradle.kts

```
val taskX by tasks.registering {
    doLast {
        println("taskX")
    }
}

val taskY by tasks.registering {
    doLast {
        println("taskY")
    }
}

taskX {
    dependsOn(taskY)
}
```

Output of `gradle -q taskX`

```
> gradle -q taskX
taskY
taskX
```

For more advanced uses, you can define a task dependency using a lazy block. When evaluated, the

block is passed the task whose dependencies are being calculated. The lazy block should return a single `Task` or collection of `Task` objects, which are then treated as dependencies of the task. The following example adds a dependency from `taskX` to all the tasks in the project whose name starts with `lib`:

Example 85. Adding dependency using a lazy block

build.gradle

```
def taskX = tasks.register('taskX') {
    doLast {
        println 'taskX'
    }
}

// Using a Gradle Provider
taskX.configure {
    dependsOn(provider {
        tasks.findAll { task -> task.name.startsWith('lib') }
    })
}

tasks.register('lib1') {
    doLast {
        println('lib1')
    }
}

tasks.register('lib2') {
    doLast {
        println('lib2')
    }
}

tasks.register('notALib') {
    doLast {
        println('notALib')
    }
}
```

build.gradle.kts

```
val taskX by tasks.registering {
    doLast {
        println("taskX")
    }
}

// Using a Gradle Provider
taskX {
    dependsOn(provider {
        tasks.filter { task -> task.name.startsWith("lib") }
    })
}

tasks.register("lib1") {
    doLast {
        println("lib1")
    }
}

tasks.register("lib2") {
    doLast {
        println("lib2")
    }
}

tasks.register("notALib") {
    doLast {
        println("notALib")
    }
}
```

Output of `gradle -q taskX`

```
> gradle -q taskX
lib1
lib2
taskX
```

For more information about task dependencies, see the Task API.

Ordering tasks

In some cases it is useful to control the *order* in which 2 tasks will execute, without introducing an explicit dependency between those tasks. The primary difference between a task *ordering* and a task *dependency* is that an ordering rule does not influence which tasks will be executed, only the

order in which they will be executed.

Task ordering can be useful in a number of scenarios:

- Enforce sequential ordering of tasks: e.g. 'build' never runs before 'clean'.
- Run build validations early in the build: e.g. validate I have the correct credentials before starting the work for a release build.
- Get feedback faster by running quick verification tasks before long verification tasks: e.g. unit tests should run before integration tests.
- A task that aggregates the results of all tasks of a particular type: e.g. test report task combines the outputs of all executed test tasks.

There are two ordering rules available: "*must run after*" and "*should run after*".

When you use the "must run after" ordering rule you specify that `taskB` must always run after `taskA`, whenever both `taskA` and `taskB` will be run. This is expressed as `taskB.mustRunAfter(taskA)`. The "should run after" ordering rule is similar but less strict as it will be ignored in two situations. Firstly if using that rule introduces an ordering cycle. Secondly when using parallel execution and all dependencies of a task have been satisfied apart from the "should run after" task, then this task will be run regardless of whether its "should run after" dependencies have been run or not. You should use "should run after" where the ordering is helpful but not strictly required.

With these rules present it is still possible to execute `taskA` without `taskB` and vice-versa.

Example 86. Adding a 'must run after' task ordering

build.gradle

```
def taskX = tasks.register('taskX') {
    doLast {
        println 'taskX'
    }
}
def taskY = tasks.register('taskY') {
    doLast {
        println 'taskY'
    }
}
taskY.configure {
    mustRunAfter taskX
}
```

build.gradle.kts

```
val taskX by tasks.registering {
    doLast {
        println("taskX")
    }
}
val taskY by tasks.registering {
    doLast {
        println("taskY")
    }
}
taskY {
    mustRunAfter(taskX)
}
```

Output of `gradle -q taskY taskX`

```
> gradle -q taskY taskX
taskX
taskY
```

Example 87. Adding a 'should run after' task ordering

build.gradle

```
def taskX = tasks.register('taskX') {
    doLast {
        println 'taskX'
    }
}
def taskY = tasks.register('taskY') {
    doLast {
        println 'taskY'
    }
}
taskY.configure {
    shouldRunAfter taskX
}
```

build.gradle.kts

```
val taskX by tasks.registering {
    doLast {
        println("taskX")
    }
}
val taskY by tasks.registering {
    doLast {
        println("taskY")
    }
}
taskY {
    shouldRunAfter(taskX)
}
```

Output of `gradle -q taskY taskX`

```
> gradle -q taskY taskX
taskX
taskY
```

In the examples above, it is still possible to execute `taskY` without causing `taskX` to run:

Example 88. Task ordering does not imply task execution

Output of `gradle -q taskY`

```
> gradle -q taskY
taskY
```

To specify a “must run after” or “should run after” ordering between 2 tasks, you use the Task.mustRunAfter(java.lang.Object...) and Task.shouldRunAfter(java.lang.Object...) methods. These methods accept a task instance, a task name or any other input accepted by Task.dependsOn(java.lang.Object...).

Note that “`B.mustRunAfter(A)`” or “`B.shouldRunAfter(A)`” does not imply any execution dependency between the tasks:

- It is possible to execute tasks `A` and `B` independently. The ordering rule only has an effect when both tasks are scheduled for execution.
- When run with `--continue`, it is possible for `B` to execute in the event that `A` fails.

As mentioned before, the “should run after” ordering rule will be ignored if it introduces an ordering cycle:

Example 89. A 'should run after' task ordering is ignored if it introduces an ordering cycle

build.gradle

```
def taskX = tasks.register('taskX') {
    doLast {
        println 'taskX'
    }
}
def taskY = tasks.register('taskY') {
    doLast {
        println 'taskY'
    }
}
def taskZ = tasks.register('taskZ') {
    doLast {
        println 'taskZ'
    }
}
taskX.configure { dependsOn(taskY) }
taskY.configure { dependsOn(taskZ) }
taskZ.configure { shouldRunAfter(taskX) }
```

build.gradle.kts

```
val taskX by tasks.registering {
    doLast {
        println("taskX")
    }
}
val taskY by tasks.registering {
    doLast {
        println("taskY")
    }
}
val taskZ by tasks.registering {
    doLast {
        println("taskZ")
    }
}
taskX { dependsOn(taskY) }
taskY { dependsOn(taskZ) }
taskZ { shouldRunAfter(taskX) }
```

Output of `gradle -q taskX`

```
> gradle -q taskX
taskZ
taskY
taskX
```

Adding a description to a task

You can add a description to your task. This description is displayed when executing `gradle tasks`.

Example 90. Adding a description to a task

build.gradle

```
tasks.register('copy', Copy) {
   description 'Copies the resource directory to the target directory.'
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

build.gradle.kts

```
tasks.register<Copy>("copy") {
   description = "Copies the resource directory to the target directory."
   from("resources")
   into("target")
   include("**/*.txt", "**/*.xml", "**/*.properties")
}
```

Skipping tasks

Gradle offers multiple ways to skip the execution of a task.

Using a predicate

You can use `Task.onlyIf` to attach a predicate to a task. The task's actions are only executed if the predicate evaluates to true. The predicate is passed the task as a parameter, and should return true if the task should execute and false if the task should be skipped. The predicate is evaluated just before the task is executed. Passing an optional reason string to `onlyIf()` is useful for explaining why the task is skipped.

Example 91. Skipping a task using a predicate

build.gradle

```
def hello = tasks.register('hello') {
    doLast {
        println 'hello world'
    }
}

hello.configure {
    onlyIf("there is no property skipHello") { !project.hasProperty(
'skipHello') }
}
```

build.gradle.kts

```
val hello by tasks.registering {
    doLast {
        println("hello world")
    }
}

hello {
    onlyIf("there is no property skipHello") {
!project.hasProperty("skipHello") }
}
```

Output of `gradle hello -PskipHello`

```
> gradle hello -PskipHello
> Task :hello SKIPPED

BUILD SUCCESSFUL in 0s
```

It is possible to find the reason for a task being skipped by running the build with the `--info` logging level.

Output of `gradle hello -PskipHello --hello`

```
> gradle hello -PskipHello --info
...

> Task :hello SKIPPED
Skipping task ':hello' as task onlyIf 'there is no property skipHello' is false.
:hello (Thread[included builds,5,main]) completed. Took 0.018 secs.

BUILD SUCCESSFUL in 13s
```

Using StopExecutionException

If the logic for skipping a task can't be expressed with a predicate, you can use the StopExecutionException. If this exception is thrown by an action, the further execution of this action as well as the execution of any following action of this task is skipped. The build continues with executing the next task.

Example 92. Skipping tasks with StopExecutionException

build.gradle

```
def compile = tasks.register('compile') {
    doLast {
        println 'We are doing the compile.'
    }
}

compile.configure {
    doFirst {
        // Here you would put arbitrary conditions in real life.
        if (true) {
            throw new StopExecutionException()
        }
    }
}
tasks.register('myTask') {
    dependsOn('compile')
    doLast {
        println 'I am not affected'
    }
}
```

build.gradle.kts

```
val compile by tasks.registering {
    doLast {
        println("We are doing the compile.")
    }
}

compile {
    doFirst {
        // Here you would put arbitrary conditions in real life.
        if (true) {
            throw StopExecutionException()
        }
    }
}
tasks.register("myTask") {
    dependsOn(compile)
    doLast {
        println("I am not affected")
    }
}
```

Output of `gradle -q myTask`

```
> gradle -q myTask
I am not affected
```

This feature is helpful if you work with tasks provided by Gradle. It allows you to add *conditional* execution of the built-in actions of such a task.[3]

Enabling and disabling tasks

Every task has an `enabled` flag which defaults to `true`. Setting it to `false` prevents the execution of any of the task's actions. A disabled task will be labelled SKIPPED.

Example 93. Enabling and disabling tasks

build.gradle

```
def disableMe = tasks.register('disableMe') {
    doLast {
        println 'This should not be printed if the task is disabled.'
    }
}

disableMe.configure {
    enabled = false
}
```

build.gradle.kts

```
val disableMe by tasks.registering {
    doLast {
        println("This should not be printed if the task is disabled.")
    }
}

disableMe {
    enabled = false
}
```

Output of `gradle disableMe`

```
> gradle disableMe
> Task :disableMe SKIPPED

BUILD SUCCESSFUL in 0s
```

Task timeouts

Every task has a `timeout` property which can be used to limit its execution time. When a task reaches its timeout, its task execution thread is interrupted. The task will be marked as failed. Finalizer tasks will still be run. If `--continue` is used, other tasks can continue running after it. Tasks that don't respond to interrupts can't be timed out. All of Gradle's built-in tasks respond to timeouts in a timely manner.

Example 94. Specifying task timeouts

build.gradle

```
tasks.register("hangingTask") {
    doLast {
        Thread.sleep(100000)
    }
    timeout = Duration.ofMillis(500)
}
```

build.gradle.kts

```
tasks.register("hangingTask") {
    doLast {
        Thread.sleep(100000)
    }
    timeout.set(Duration.ofMillis(500))
}
```

Up-to-date checks (AKA Incremental Build)

An important part of any build tool is the ability to avoid doing work that has already been done. Consider the process of compilation. Once your source files have been compiled, there should be no need to recompile them unless something has changed that affects the output, such as the modification of a source file or the removal of an output file. And compilation can take a significant amount of time, so skipping the step when it's not needed saves a lot of time.

Gradle supports this behavior out of the box through a feature it calls incremental build. You have almost certainly already seen it in action: it's active nearly every time the `UP-TO-DATE` text appears next to the name of a task when you run a build. Task outcomes are described in Task outcomes.

How does incremental build work? And what does it take to make use of it in your own tasks? Let's take a look.

Task inputs and outputs

In the most common case, a task takes some inputs and generates some outputs. If we use the compilation example from earlier, we can see that the source files are the inputs and, in the case of Java, the generated class files are the outputs. Other inputs might include things like whether debug information should be included.

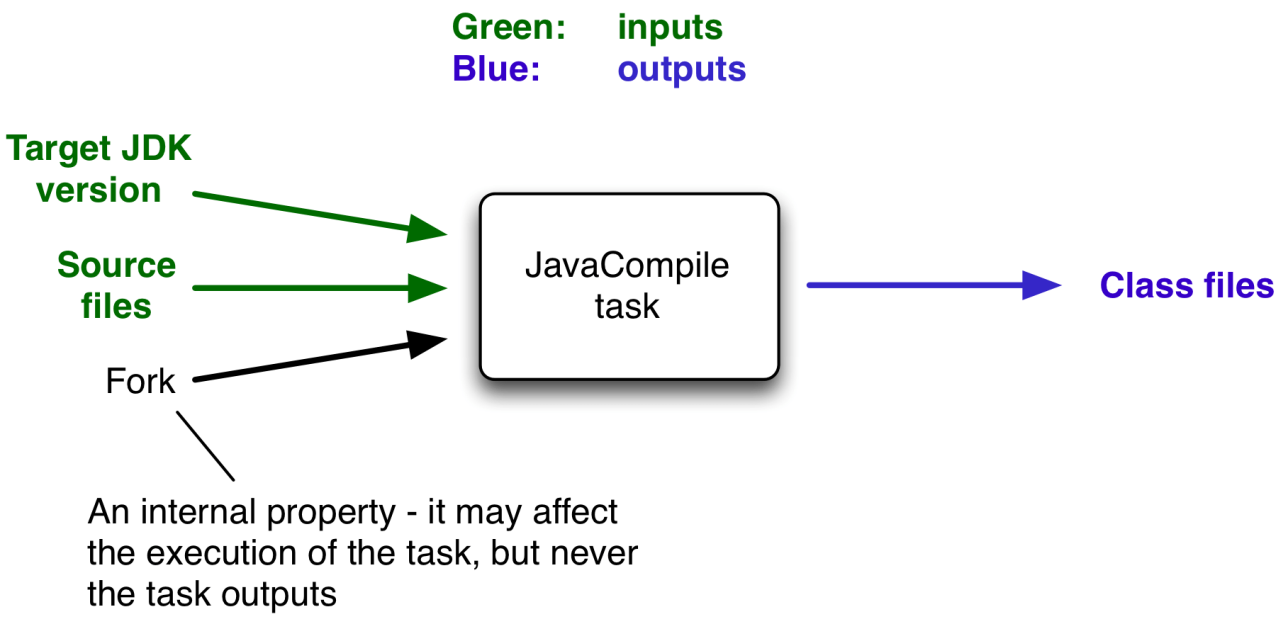


Figure 6. Example task inputs and outputs

An important characteristic of an input is that it affects one or more outputs, as you can see from the previous figure. Different bytecode is generated depending on the content of the source files and the minimum version of the Java runtime you want to run the code on. That makes them task inputs. But whether compilation has 500MB or 600MB of maximum memory available, determined by the `memoryMaximumSize` property, has no impact on what bytecode gets generated. In Gradle terminology, `memoryMaximumSize` is just an internal task property.

As part of incremental build, Gradle tests whether any of the task inputs or outputs has changed since the last build. If they haven't, Gradle can consider the task up to date and therefore skip executing its actions. Also note that incremental build won't work unless a task has at least one task output, although tasks usually have at least one input as well.

What this means for build authors is simple: you need to tell Gradle which task properties are inputs and which are outputs. If a task property affects the output, be sure to register it as an input, otherwise the task will be considered up to date when it's not. Conversely, don't register properties as inputs if they don't affect the output, otherwise the task will potentially execute when it doesn't need to. Also be careful of non-deterministic tasks that may generate different output for exactly the same inputs: these should not be configured for incremental build as the up-to-date checks won't work.

Let's now look at how you can register task properties as inputs and outputs.

Custom task types

If you're implementing a custom task as a class, then it takes just two steps to make it work with incremental build:

1. Create typed properties (via getter methods) for each of your task inputs and outputs
2. Add the appropriate annotation to each of those properties

NOTE	Annotations must be placed on getters or on Groovy properties. Annotations placed on setters, or on a Java field without a corresponding annotated getter, are ignored.

Gradle supports four main categories of inputs and outputs:

- Simple values

  Things like strings and numbers. More generally, a simple value can have any type that implements `Serializable`.

- Filesystem types

  These consist of `RegularFile`, `Directory` and the standard `File` class but also derivatives of Gradle's FileCollection type and anything else that can be passed to either the Project.file(java.lang.Object) method — for single file/directory properties — or the Project.files(java.lang.Object...) method.

- Dependency resolution results

  This includes the ResolvedArtifactResult type for artifact metadata and the ResolvedComponentResult type for dependency graphs. Note that they are only supported wrapped in a `Provider`.

- Nested values

  Custom types that don't conform to the other two categories but have their own properties that are inputs or outputs. In effect, the task inputs or outputs are nested inside these custom types.

As an example, imagine you have a task that processes templates of varying types, such as FreeMarker, Velocity, Moustache, etc. It takes template source files and combines them with some model data to generate populated versions of the template files.

This task will have three inputs and one output:

- Template source files
- Model data
- Template engine
- Where the output files are written

When you're writing a custom task class, it's easy to register properties as inputs or outputs via annotations. To demonstrate, here is a skeleton task implementation with some suitable inputs and outputs, along with their annotations:

Example 95. Custom task class

buildSrc/src/main/java/org/example/ProcessTemplates.java

```
package org.example;

import java.util.HashMap;
import org.gradle.api.DefaultTask;
import org.gradle.api.file.ConfigurableFileCollection;
import org.gradle.api.file.DirectoryProperty;
import org.gradle.api.provider.Property;
import org.gradle.api.tasks.*;

public abstract class ProcessTemplates extends DefaultTask {

    @Input
    public abstract Property<TemplateEngineType> getTemplateEngine();

    @InputFiles
    public abstract ConfigurableFileCollection getSourceFiles();

    @Nested
    public abstract TemplateData getTemplateData();

    @OutputDirectory
    public abstract DirectoryProperty getOutputDir();

    @TaskAction
    public void processTemplates() {
        // ...
    }
}
```

buildSrc/src/main/java/org/example/TemplateData.java

```
package org.example;

import org.gradle.api.provider.MapProperty;
import org.gradle.api.provider.Property;
import org.gradle.api.tasks.Input;

public abstract class TemplateData {

    @Input
    public abstract Property<String> getName();

    @Input
    public abstract MapProperty<String, String> getVariables();
}
```

Output of `gradle processTemplates`

```
> gradle processTemplates
> Task :processTemplates

BUILD SUCCESSFUL in 0s
3 actionable tasks: 3 up-to-date
```

Output of `gradle processTemplates` *(run again)*

```
> gradle processTemplates
> Task :processTemplates UP-TO-DATE

BUILD SUCCESSFUL in 0s
3 actionable tasks: 3 up-to-date
```

There's plenty to talk about in this example, so let's work through each of the input and output properties in turn:

- `templateEngine`

  Represents which engine to use when processing the source templates, e.g. FreeMarker, Velocity, etc. You could implement this as a string, but in this case we have gone for a custom enum as it provides greater type information and safety. Since enums implement `Serializable` automatically, we can treat this as a simple value and use the `@Input` annotation, just as we would with a `String` property.

- `sourceFiles`

  The source templates that the task will be processing. Single files and collections of files need their own special annotations. In this case, we're dealing with a collection of input files and so we use the `@InputFiles` annotation. You'll see more file-oriented annotations in a table later.

- `templateData`

  For this example, we're using a custom class to represent the model data. However, it does not implement `Serializable`, so we can't use the `@Input` annotation. That's not a problem as the properties within `TemplateData` — a string and a hash map with serializable type parameters — are serializable and can be annotated with `@Input`. We use `@Nested` on `templateData` to let Gradle know that this is a value with nested input properties.

- `outputDir`

  The directory where the generated files go. As with input files, there are several annotations for output files and directories. A property representing a single directory requires `@OutputDirectory`. You'll learn about the others soon.

These annotated properties mean that Gradle will skip the task if none of the source files, template engine, model data or generated files has changed since the previous time Gradle executed the task.

This will often save a significant amount of time. You can learn how Gradle detects changes later.

This example is particularly interesting because it works with collections of source files. What happens if only one source file changes? Does the task process all the source files again or just the modified one? That depends on the task implementation. If the latter, then the task itself is incremental, but that's a different feature to the one we're discussing here. Gradle does help task implementers with this via its incremental task inputs feature.

Now that you have seen some of the input and output annotations in practice, let's take a look at all the annotations available to you and when you should use them. The table below lists the available annotations and the corresponding property type you can use with each one.

Table 4. Incremental build property type annotations

Annotation	Expected property type	Description
`@Input`	Any `Serializable` type or dependency resolution result types	A simple input value or dependency resolution results
`@InputFile`	`File`*	A single input file (not directory)
`@InputDirectory`	`File`*	A single input directory (not file)
`@InputFiles`	`Iterable<File>`*	An iterable of input files and directories

Annotation	Expected property type	Description
`@Classpath`	`Iterable<File>`*	An iterable of input files and directories that represent a Java classpath. This allows the task to ignore irrelevant changes to the property, such as different names for the same files. It is similar to annotating the property `@PathSensitive(RELATIVE)` but it will ignore the names of JAR files directly added to the classpath, and it will consider changes in the order of the files as a change in the classpath. Gradle will inspect the contents of jar files on the classpath and ignore changes that do not affect the semantics of the classpath (such as file dates and entry order). See also Using the classpath annotations. **Note:** The `@Classpath` annotation was introduced in Gradle 3.2. To stay compatible with earlier Gradle versions, classpath properties should also be annotated with `@InputFiles`.

Annotation	Expected property type	Description
`@CompileClasspath`	`Iterable<File>*`	An iterable of input files and directories that represent a Java compile classpath. This allows the task to ignore irrelevant changes that do not affect the API of the classes in classpath. See also Using the classpath annotations. The following kinds of changes to the classpath will be ignored: • Changes to the path of jar or top level directories. • Changes to timestamps and the order of entries in Jars. • Changes to resources and Jar manifests, including adding or removing resources. • Changes to private class elements, such as private fields, methods and inner classes. • Changes to code, such as method bodies, static initializers and field initializers (except for constants). • Changes to debug information, for example when a change to a comment affects the line numbers in class debug information. • Changes to directories, including directory entries in Jars.

Annotation	Expected property type	Description
`@OutputFile`	`File`*	A single output file (not directory)
`@OutputDirectory`	`File`*	A single output directory (not file)
`@OutputFiles`	`Map<String, File>`** or `Iterable<File>`*	An iterable or map of output files. Using a file tree turns caching off for the task.
`@OutputDirectories`	`Map<String, File>`** or `Iterable<File>`*	An iterable of output directories. Using a file tree turns caching off for the task.
`@Destroys`	`File` or `Iterable<File>`*	Specifies one or more files that are removed by this task. Note that a task can define either inputs/outputs or destroyables, but not both.
`@LocalState`	`File` or `Iterable<File>`*	Specifies one or more files that represent the local state of the task. These files are removed when the task is loaded from cache.
`@Nested`	Any custom type	A custom type that may not implement `Serializable` but does have at least one field or property marked with one of the annotations in this table. It could even be another `@Nested`.
`@Console`	Any type	Indicates that the property is neither an input nor an output. It simply affects the console output of the task in some way, such as increasing or decreasing the verbosity of the task.
`@Internal`	Any type	Indicates that the property is used internally but is neither an input nor an output.

Annotation	Expected property type	Description
`@ReplacedBy`	Any type	Indicates that the property has been replaced by another and should be ignored as an input or output.
`@SkipWhenEmpty`	`File` or `Iterable<File>`*	Used with `@InputFiles` or `@InputDirectory` to tell Gradle to skip the task if the corresponding files or directory are empty, along with all other input files declared with this annotation. Tasks that have been skipped due to all of their input files that were declared with this annotation being empty will result in a distinct “no source” outcome. For example, `NO-SOURCE` will be emitted in the console output. Implies `@Incremental`.
`@Incremental`	`Provider<FileSystemLocation>` or `FileCollection`	Used with `@InputFiles` or `@InputDirectory` to instruct Gradle to track changes to the annotated file property, so the changes can be queried via `@InputChanges.getFileChanges()`. Required for incremental tasks.
`@Optional`	Any type	Used with any of the property type annotations listed in the Optional API documentation. This annotation disables validation checks on the corresponding property. See the section on validation for more details.

Annotation	Expected property type	Description
`@PathSensitive`	`File` or `Iterable<File>`*	Used with any input file property to tell Gradle to only consider the given part of the file paths as important. For example, if a property is annotated with `@PathSensitive(PathSensitivity.NAME_ONLY)`, then moving the files around without changing their contents will not make the task out-of-date.
`@IgnoreEmptyDirectories`	`File` or `Iterable<File>`*	Used with `@InputFiles` or `@InputDirectory` to instruct Gradle to track only changes to the contents of directories and not differences in the directories themselves. For example, removing, renaming or adding an empty directory somewhere in the directory structure will not make the task out-of-date.
`@NormalizeLineEndings`	`File` or `Iterable<File>`*	Used with `@InputFiles`, `@InputDirectory` or `@Classpath` to instruct Gradle to normalize line endings when calculating up-to-date checks or build cache keys. For example, switching a file between Unix line endings and Windows line endings (or vice-versa) will not make the task out-of-date.

NOTE

* In fact, `File` can be any type accepted by Project.file(java.lang.Object) and `Iterable<File>` can be any type accepted by Project.files(java.lang.Object...). This includes instances of `Callable`, such as closures, allowing for lazy evaluation of the property values. Be aware that the types `FileCollection` and `FileTree` are `Iterable<File>`s.

** Similar to the above, `File` can be any type accepted by Project.file(java.lang.Object). The `Map` itself can be wrapped in `Callable`s, such as closures.

Annotations are inherited from all parent types including implemented interfaces. Property type annotations override any other property type annotation declared in a parent type. This way an `@InputFile` property can be turned into an `@InputDirectory` property in a child task type.

Annotations on a property declared in a type override similar annotations declared by the superclass and in any implemented interfaces. Superclass annotations take precedence over annotations declared in implemented interfaces.

The Console and Internal annotations in the table are special cases as they don't declare either task inputs or task outputs. So why use them? It's so that you can take advantage of the Java Gradle Plugin Development plugin to help you develop and publish your own plugins. This plugin checks whether any properties of your custom task classes lack an incremental build annotation. This protects you from forgetting to add an appropriate annotation during development.

Using dependency resolution results

Dependency resolution results can be consumed as task inputs in two ways. First by consuming the graph of the resolved metadata using ResolvedComponentResult. Second by consuming the flat set of the resolved artifacts using ResolvedArtifactResult.

A resolved graph can be obtained lazily from the incoming resolution result of a `Configuration` and wired to an `@Input` property:

Example 96. Resolved graph as task input

Task declaration

```
link:https://docs.gradle.org/7.6.6-20260912032715+0000/samples/writing-tasks/tasks-with-dependency-resolution-result-inputs/common/dependency-reports/src/main/java/com/example/GraphResolvedComponents.java[tag=inputs]
```

Task configuration

```
link:https://docs.gradle.org/7.6.6-20260912032715+0000/samples/writing-tasks/tasks-with-dependency-resolution-result-inputs/common/dependency-reports/src/main/java/com/example/DependencyReportsPlugin.java[tag=graphResolvedComponents]
```

The resolved set of artifacts can be obtained lazily from the incoming artifacts of a `Configuration`. Given the `ResolvedArtifactResult` type contains both metadata and file information, instances need to be transformed to metadata only before being wired to an `@Input` property:

Example 97. Resolved artifacts as task input

Task declaration

```
link:https://docs.gradle.org/7.6.6-20260912032715+0000/samples/writing-tasks/tasks-with-dependency-resolution-result-inputs/common/dependency-reports/src/main/java/com/example/ListResolvedArtifacts.java[tag=inputs]
```

Task configuration

```
link:https://docs.gradle.org/7.6.6-20260912032715+0000/samples/writing-tasks/tasks-with-dependency-resolution-result-inputs/common/dependency-reports/src/main/java/com/example/DependencyReportsPlugin.java[tag=listResolvedArtifacts]
```

Both graph and flat results can be combined and augmented with resolved file information. This is all demonstrated in the Tasks with dependency resolution result inputs sample.

Using the classpath annotations

Besides `@InputFiles`, for JVM-related tasks Gradle understands the concept of classpath inputs. Both runtime and compile classpaths are treated differently when Gradle is looking for changes.

As opposed to input properties annotated with `@InputFiles`, for classpath properties the order of the entries in the file collection matter. On the other hand, the names and paths of the directories and jar files on the classpath itself are ignored. Timestamps and the order of class files and resources inside jar files on a classpath are ignored, too, thus recreating a jar file with different file dates will not make the task out of date.

Runtime classpaths are marked with `@Classpath`, and they offer further customization via classpath normalization.

Input properties annotated with `@CompileClasspath` are considered Java compile classpaths. Additionally to the aforementioned general classpath rules, compile classpaths ignore changes to everything but class files. Gradle uses the same class analysis described in Java compile avoidance to further filter changes that don't affect the class' ABIs. This means that changes which only touch the implementation of classes do not make the task out of date.

Nested inputs

When analyzing `@Nested` task properties for declared input and output sub-properties Gradle uses the type of the actual value. Hence it can discover all sub-properties declared by a runtime sub-type.

When adding `@Nested` to a `Provider`, the value of the `Provider` is treated as a nested input.

When adding `@Nested` to an iterable, each element is treated as a separate nested input. Each nested input in the iterable is assigned a name, which by default is the dollar sign followed by the index in

the iterable, e.g. `$2`. If an element of the iterable implements `Named`, then the name is used as property name. The ordering of the elements in the iterable is crucial for reliable up-to-date checks and caching if not all of the elements implement `Named`. Multiple elements which have the same name are not allowed.

When adding `@Nested` to a map, then for each value a nested input is added, using the key as name.

The type and classpath of nested inputs is tracked, too. This ensures that changes to the implementation of a nested input causes the build to be out of date. By this it is also possible to add user provided code as an input, e.g. by annotating an `@Action` property with `@Nested`. Note that any inputs to such actions should be tracked, either by annotated properties on the action or by manually registering them with the task.

Using nested inputs allows richer modeling and extensibility for tasks, as e.g. shown by Test.getJvmArgumentProviders().

This allows us to model the JaCoCo Java agent, thus declaring the necessary JVM arguments and providing the inputs and outputs to Gradle:

JacocoAgent.java

```
class JacocoAgent implements CommandLineArgumentProvider {
    private final JacocoTaskExtension jacoco;

    public JacocoAgent(JacocoTaskExtension jacoco) {
        this.jacoco = jacoco;
    }

    @Nested
    @Optional
    public JacocoTaskExtension getJacoco() {
        return jacoco.isEnabled() ? jacoco : null;
    }

    @Override
    public Iterable<String> asArguments() {
        return jacoco.isEnabled() ? ImmutableList.of(jacoco.getAsJvmArg()) :
Collections.<String>emptyList();
    }
}

test.getJvmArgumentProviders().add(new JacocoAgent(extension));
```

For this to work, `JacocoTaskExtension` needs to have the correct input and output annotations.

The approach works for Test JVM arguments, since `Test.getJvmArgumentProviders()` is an `Iterable` annotated with `@Nested`.

There are other task types where this kind of nested inputs are available:

- JavaExec.getArgumentProviders() - model e.g. custom tools
- JavaExec.getJvmArgumentProviders() - used for Jacoco Java agent
- CompileOptions.getCompilerArgumentProviders() - model e.g. annotation processors
- Exec.getArgumentProviders() - model e.g. custom tools
- JavaCompile.getOptions().getForkOptions().getJvmArgumentProviders() - model Java compiler daemon command line arguments
- GroovyCompile.getGroovyOptions().getForkOptions().getJvmArgumentProviders() - model Groovy compiler daemon command line arguments
- ScalaCompile.getScalaOptions().getForkOptions().getJvmArgumentProviders() - model Scala compiler daemon command line arguments

In the same way, this kind of modelling is available to custom tasks.

Runtime validation

When executing the build Gradle checks if task types are declared with the proper annotations. It tries to identify problems where e.g. annotations are used on incompatible types, or on setters etc. Any getter not annotated with an input/output annotation is also flagged. These problems then fail the build or are turned into deprecation warnings when the task is executed.

NOTE

Tasks that have a validation warning are executed without any optimizations. Specifically, they never can be:

- up-to-date,
- loaded from or stored in the build cache,
- executed in parallel with other tasks, even if parallel execution is enabled,
- executed incrementally.

The in-memory representation of the file system state (aka *Virtual File System)* is also invalidated before an invalid task is executed.

Runtime API

Custom task classes are an easy way to bring your own build logic into the arena of incremental build, but you don't always have that option. That's why Gradle also provides an alternative API that can be used with any tasks, which we look at next.

When you don't have access to the source for a custom task class, there is no way to add any of the annotations we covered in the previous section. Fortunately, Gradle provides a runtime API for scenarios just like that. It can also be used for ad-hoc tasks, as you'll see next.

Using it for ad-hoc tasks

This runtime API is provided through a couple of aptly named properties that are available on every Gradle task:

- Task.getInputs() of type TaskInputs
- Task.getOutputs() of type TaskOutputs
- Task.getDestroyables() of type TaskDestroyables

These objects have methods that allow you to specify files, directories and values which constitute the task's inputs and outputs. In fact, the runtime API has almost feature parity with the annotations. All it lacks is an equivalent for `@Nested`.

Let's take the template processing example from before and see how it would look as an ad-hoc task that uses the runtime API:

Example 98. Ad-hoc task

build.gradle

```
tasks.register('processTemplatesAdHoc') {
    inputs.property('engine', TemplateEngineType.FREEMARKER)
    inputs.files(fileTree('src/templates'))
        .withPropertyName('sourceFiles')
        .withPathSensitivity(PathSensitivity.RELATIVE)
    inputs.property('templateData.name', 'docs')
    inputs.property('templateData.variables', [year: '2013'])
    outputs.dir(layout.buildDirectory.dir('genOutput2'))
        .withPropertyName('outputDir')

    doLast {
        // Process the templates here
    }
}
```

build.gradle.kts

```
tasks.register("processTemplatesAdHoc") {
    inputs.property("engine", TemplateEngineType.FREEMARKER)
    inputs.files(fileTree("src/templates"))
        .withPropertyName("sourceFiles")
        .withPathSensitivity(PathSensitivity.RELATIVE)
    inputs.property("templateData.name", "docs")
    inputs.property("templateData.variables", mapOf("year" to "2013"))
    outputs.dir(layout.buildDirectory.dir("genOutput2"))
        .withPropertyName("outputDir")

    doLast {
        // Process the templates here
    }
}
```

Output of `gradle processTemplatesAdHoc`

```
> gradle processTemplatesAdHoc
> Task :processTemplatesAdHoc

BUILD SUCCESSFUL in 0s
3 actionable tasks: 3 executed
```

As before, there's much to talk about. To begin with, you should really write a custom task class for

this as it's a non-trivial implementation that has several configuration options. In this case, there are no task properties to store the root source folder, the location of the output directory or any of the other settings. That's deliberate to highlight the fact that the runtime API doesn't require the task to have any state. In terms of incremental build, the above ad-hoc task will behave the same as the custom task class.

All the input and output definitions are done through the methods on `inputs` and `outputs`, such as `property()`, `files()`, and `dir()`. Gradle performs up-to-date checks on the argument values to determine whether the task needs to run again or not. Each method corresponds to one of the incremental build annotations, for example `inputs.property()` maps to `@Input` and `outputs.dir()` maps to `@OutputDirectory`.

The files that a task removes can be specified through `destroyables.register()`.

Example 99. Ad-hoc task declaring a destroyable

build.gradle

```
tasks.register('removeTempDir') {
    destroyables.register(layout.projectDirectory.dir('tmpDir'))
    doLast {
        delete(layout.projectDirectory.dir('tmpDir'))
    }
}
```

build.gradle.kts

```
tasks.register("removeTempDir") {
    destroyables.register(layout.projectDirectory.dir("tmpDir"))
    doLast {
        delete(layout.projectDirectory.dir("tmpDir"))
    }
}
```

One notable difference between the runtime API and the annotations is the lack of a method that corresponds directly to `@Nested`. That's why the example uses two `property()` declarations for the template data, one for each `TemplateData` property. You should utilize the same technique when using the runtime API with nested values. Any given task can either declare destroyables or inputs/outputs, but cannot declare both.

Fine-grained configuration

The runtime API methods only allow you to declare your inputs and outputs in themselves. However, the file-oriented ones return a builder — of type TaskInputFilePropertyBuilder — that lets you provide additional information about those inputs and outputs.

You can learn about all the options provided by the builder in its API documentation, but we'll show you a simple example here to give you an idea of what you can do.

Let's say we don't want to run the `processTemplates` task if there are no source files, regardless of whether it's a clean build or not. After all, if there are no source files, there's nothing for the task to do. The builder allows us to configure this like so:

Example 100. Using skipWhenEmpty() via the runtime API

build.gradle

```
tasks.register('processTemplatesAdHocSkipWhenEmpty') {
    // ...

    inputs.files(fileTree('src/templates') {
            include '**/*.fm'
        })
        .skipWhenEmpty()
        .withPropertyName('sourceFiles')
        .withPathSensitivity(PathSensitivity.RELATIVE)
        .ignoreEmptyDirectories()

    // ...
}
```

build.gradle.kts

```
tasks.register("processTemplatesAdHocSkipWhenEmpty") {
    // ...

    inputs.files(fileTree("src/templates") {
            include("**/*.fm")
        })
        .skipWhenEmpty()
        .withPropertyName("sourceFiles")
        .withPathSensitivity(PathSensitivity.RELATIVE)
        .ignoreEmptyDirectories()

    // ...
}
```

Output of `gradle clean processTemplatesAdHocSkipWhenEmpty`

```
> gradle clean processTemplatesAdHocSkipWhenEmpty
> Task :processTemplatesAdHocSkipWhenEmpty NO-SOURCE

BUILD SUCCESSFUL in 0s
3 actionable tasks: 2 executed, 1 up-to-date
```

The `TaskInputs.files()` method returns a builder that has a `skipWhenEmpty()` method. Invoking this method is equivalent to annotating to the property with `@SkipWhenEmpty`.

Now that you have seen both the annotations and the runtime API, you may be wondering which API you should be using. Our recommendation is to use the annotations wherever possible, and it's sometimes worth creating a custom task class just so that you can make use of them. The runtime API is more for situations in which you can't use the annotations.

Using it for custom task types

Another type of example involves registering additional inputs and outputs for instances of a custom task class. For example, imagine that the `ProcessTemplates` task also needs to read `src/headers/headers.txt` (e.g. because it is included from one of the sources). You'd want Gradle to know about this input file, so that it can re-execute the task whenever the contents of this file change. With the runtime API you can do just that:

Example 101. Using runtime API with custom task type

build.gradle

```
tasks.register('processTemplatesWithExtraInputs', ProcessTemplates) {
    // ...

    inputs.file('src/headers/headers.txt')
        .withPropertyName('headers')
        .withPathSensitivity(PathSensitivity.NONE)
}
```

build.gradle.kts

```
tasks.register<ProcessTemplates>("processTemplatesWithExtraInputs") {
    // ...

    inputs.file("src/headers/headers.txt")
        .withPropertyName("headers")
        .withPathSensitivity(PathSensitivity.NONE)
}
```

Using the runtime API like this is a little like using `doLast()` and `doFirst()` to attach extra actions to a task, except in this case we're attaching information about inputs and outputs.

WARNING	If the task type is already using the incremental build annotations, registering inputs or outputs with the same property names will result in an error.

Important beneficial side effects

Once you declare a task's formal inputs and outputs, Gradle can then infer things about those properties. For example, if an input of one task is set to the output of another, that means the first

task depends on the second, right? Gradle knows this and can act upon it.

We'll look at this feature next and also some other features that come from Gradle knowing things about inputs and outputs.

Inferred task dependencies

Consider an archive task that packages the output of the `processTemplates` task. A build author will see that the archive task obviously requires `processTemplates` to run first and so may add an explicit `dependsOn`. However, if you define the archive task like so:

Example 102. Inferred task dependency via task outputs

build.gradle

```
tasks.register('packageFiles', Zip) {
    from processTemplates.map {it.outputs }
}
```

build.gradle.kts

```
tasks.register<Zip>("packageFiles") {
    from(processTemplates.map {it.outputs })
}
```

Output of `gradle clean packageFiles`

```
> gradle clean packageFiles
> Task :processTemplates
> Task :packageFiles

BUILD SUCCESSFUL in 0s
5 actionable tasks: 4 executed, 1 up-to-date
```

Gradle will automatically make `packageFiles` depend on `processTemplates`. It can do this because it's aware that one of the inputs of packageFiles requires the output of the processTemplates task. We call this an inferred task dependency.

The above example can also be written as

Example 103. Inferred task dependency via a task argument

build.gradle

```
tasks.register('packageFiles2', Zip) {
    from processTemplates
}
```

build.gradle.kts

```
tasks.register<Zip>("packageFiles2") {
    from(processTemplates)
}
```

Output of `gradle clean packageFiles2`

```
> gradle clean packageFiles2
> Task :processTemplates
> Task :packageFiles2

BUILD SUCCESSFUL in 0s
5 actionable tasks: 4 executed, 1 up-to-date
```

This is because the `from()` method can accept a task object as an argument. Behind the scenes, `from()` uses the `project.files()` method to wrap the argument, which in turn exposes the task's formal outputs as a file collection. In other words, it's a special case!

Input and output validation

The incremental build annotations provide enough information for Gradle to perform some basic validation on the annotated properties. In particular, it does the following for each property before the task executes:

- `@InputFile` - verifies that the property has a value and that the path corresponds to a file (not a directory) that exists.
- `@InputDirectory` - same as for `@InputFile`, except the path must correspond to a directory.
- `@OutputDirectory` - verifies that the path doesn't match a file and also creates the directory if it doesn't already exist.

If one task produces an output in a location and another task consumes that location by referring to it as an input, then Gradle checks that the consumer task depends on the producer task. When the producer and the consumer tasks are executing at the same time, the build fails to avoid capturing an incorrect state.

Such validation improves the robustness of the build, allowing you to identify issues related to inputs and outputs quickly.

You will occasionally want to disable some of this validation, specifically when an input file may validly not exist. That's why Gradle provides the `@Optional` annotation: you use it to tell Gradle that a particular input is optional and therefore the build should not fail if the corresponding file or directory doesn't exist.

Continuous build

Another benefit of defining task inputs and outputs is continuous build. Since Gradle knows what files a task depends on, it can automatically run a task again if any of its inputs change. By activating continuous build when you run Gradle — through the `--continuous` or `-t` options — you will put Gradle into a state in which it continually checks for changes and executes the requested tasks when it encounters such changes.

You can find out more about this feature in Continuous build.

Task parallelism

One last benefit of defining task inputs and outputs is that Gradle can use this information to make decisions about how to run tasks when the "--parallel" option is used. For instance, Gradle will inspect the outputs of tasks when selecting the next task to run and will avoid concurrent execution of tasks that write to the same output directory. Similarly, Gradle will use the information about what files a task destroys (e.g. specified by the `Destroys` annotation) and avoid running a task that removes a set of files while another task is running that consumes or creates those same files (and vice versa). It can also determine that a task that creates a set of files has already run and that a task that consumes those files has yet to run and will avoid running a task that removes those files in between. By providing task input and output information in this way, Gradle can infer creation/consumption/destruction relationships between tasks and can ensure that task execution does not violate those relationships.

How does it work?

Before a task is executed for the first time, Gradle takes a fingerprint of the inputs. This fingerprint contains the paths of input files and a hash of the contents of each file. Gradle then executes the task. If the task completes successfully, Gradle takes a fingerprint of the outputs. This fingerprint contains the set of output files and a hash of the contents of each file. Gradle persists both fingerprints for the next time the task is executed.

Each time after that, before the task is executed, Gradle takes a new fingerprint of the inputs and outputs. If the new fingerprints are the same as the previous fingerprints, Gradle assumes that the outputs are up to date and skips the task. If they are not the same, Gradle executes the task. Gradle persists both fingerprints for the next time the task is executed.

If the stats of a file (i.e. `lastModified` and `size`) did not change, Gradle will reuse the file's fingerprint from the previous run. That means that Gradle does not detect changes when the stats of a file did not change.

Gradle also considers the *code* of the task as part of the inputs to the task. When a task, its actions,

or its dependencies change between executions, Gradle considers the task as out-of-date.

Gradle understands if a file property (e.g. one holding a Java classpath) is order-sensitive. When comparing the fingerprint of such a property, even a change in the order of the files will result in the task becoming out-of-date.

Note that if a task has an output directory specified, any files added to that directory since the last time it was executed are ignored and will NOT cause the task to be out of date. This is so unrelated tasks may share an output directory without interfering with each other. If this is not the behaviour you want for some reason, consider using TaskOutputs.upToDateWhen(groovy.lang.Closure)

Note also that changing the availability of an unavailable file (e.g. modifying the target of a broken symlink to a valid file, or vice versa), will be detected and handled by up-to-date check.

The inputs for the task are also used to calculate the build cache key used to load task outputs when enabled. For more details see Task output caching.

NOTE

For tracking the implementation of tasks, task actions and nested inputs, Gradle uses the class name and an identifier for the classpath which contains the implementation. There are some situations when Gradle is not able to track the implementation precisely:

Unknown classloader

When the classloader which loaded the implementation has not been created by Gradle, the classpath cannot be determined.

Java lambda

Java lambda classes are created at runtime with a non-deterministic classname. Therefore, the class name does not identify the implementation of the lambda and changes between different Gradle runs.

When the implementation of a task, task action or a nested input cannot be tracked precisely, Gradle disables any caching for the task. That means that the task will never be up-to-date or loaded from the build cache.

Advanced techniques

Everything you've seen so far in this section will cover most of the use cases you'll encounter, but there are some scenarios that need special treatment. We'll present a few of those next with the appropriate solutions.

Adding your own cached input/output methods

Have you ever wondered how the `from()` method of the `Copy` task works? It's not annotated with `@InputFiles` and yet any files passed to it are treated as formal inputs of the task. What's happening?

The implementation is quite simple and you can use the same technique for your own tasks to improve their APIs. Write your methods so that they add files directly to the appropriate annotated property. As an example, here's how to add a `sources()` method to the custom `ProcessTemplates` class

we introduced earlier:

Example 104. Declaring a method to add task inputs

build.gradle

```
tasks.register('processTemplates', ProcessTemplates) {
    templateEngine = TemplateEngineType.FREEMARKER
    templateData.name = 'test'
    templateData.variables = [year: '2012']
    outputDir = file(layout.buildDirectory.dir('genOutput'))

    sources fileTree('src/templates')
}
```

build.gradle.kts

```
tasks.register<ProcessTemplates>("processTemplates") {
    templateEngine.set(TemplateEngineType.FREEMARKER)
    templateData.name.set("test")
    templateData.variables.set(mapOf("year" to "2012"))
    outputDir.set(file(layout.buildDirectory.dir("genOutput")))

    sources(fileTree("src/templates"))
}
```

ProcessTemplates.java

```
public abstract class ProcessTemplates extends DefaultTask {
    // ...
    @SkipWhenEmpty
    @InputFiles
    @PathSensitive(PathSensitivity.NONE)
    public abstract ConfigurableFileCollection getSourceFiles();

    public void sources(FileCollection sourceFiles) {
        getSourceFiles().from(sourceFiles);
    }

    // ...
}
```

Output of `gradle processTemplates`

```
> gradle processTemplates
> Task :processTemplates

BUILD SUCCESSFUL in 0s
3 actionable tasks: 3 executed
```

In other words, as long as you add values and files to formal task inputs and outputs during the configuration phase, they will be treated as such regardless from where in the build you add them.

If we want to support tasks as arguments as well and treat their outputs as the inputs, we can use the `project.layout.files()` method like so:

Example 105. Declaring a method to add a task as an input

build.gradle

```
def copyTemplates = tasks.register('copyTemplates', Copy) {
    into file(layout.buildDirectory.dir('tmp'))
    from 'src/templates'
}

tasks.register('processTemplates2', ProcessTemplates) {
    // ...
    sources copyTemplates
}
```

build.gradle.kts

```
val copyTemplates by tasks.registering(Copy::class) {
    into(file(layout.buildDirectory.dir("tmp")))
    from("src/templates")
}

tasks.register<ProcessTemplates>("processTemplates2") {
    // ...
    sources(copyTemplates)
}
```

ProcessTemplates.java

```
  // ...
  public void sources(TaskProvider<?> inputTask) {
      getSourceFiles().from(getProject().getLayout().files(inputTask));
  }
  // ...
```

Output of `gradle processTemplates2`

```
> gradle processTemplates2
> Task :copyTemplates
> Task :processTemplates2

BUILD SUCCESSFUL in 0s
4 actionable tasks: 4 executed
```

This technique can make your custom task easier to use and result in cleaner build files. As an

added benefit, our use of `getProject().getLayout().files()` means that our custom method can set up an inferred task dependency.

One last thing to note: if you are developing a task that takes collections of source files as inputs, like this example, consider using the built-in SourceTask. It will save you having to implement some of the plumbing that we put into `ProcessTemplates`.

Linking an `@OutputDirectory` to an `@InputFiles`

When you want to link the output of one task to the input of another, the types often match and a simple property assignment will provide that link. For example, a `File` output property can be assigned to a `File` input.

Unfortunately, this approach breaks down when you want the files in a task's `@OutputDirectory` (of type `File`) to become the source for another task's `@InputFiles` property (of type `FileCollection`). Since the two have different types, property assignment won't work.

As an example, imagine you want to use the output of a Java compilation task — via the `destinationDir` property — as the input of a custom task that instruments a set of files containing Java bytecode. This custom task, which we'll call `Instrument`, has a `classFiles` property annotated with `@InputFiles`. You might initially try to configure the task like so:

Example 106. Failed attempt at setting up an inferred task dependency

build.gradle

```
plugins {
    id 'java-library'
}

tasks.register('badInstrumentClasses', Instrument) {
    classFiles.from fileTree(tasks.named('compileJava').map { it
.destinationDir })
    destinationDir = file(layout.buildDirectory.dir('instrumented'))
}
```

build.gradle.kts

```
plugins {
    id("java-library")
}

tasks.register<Instrument>("badInstrumentClasses") {
    classFiles.from(fileTree(tasks.compileJava.map { it.destinationDir }))
    destinationDir.set(file(layout.buildDirectory.dir("instrumented")))
}
```

Output of `gradle clean badInstrumentClasses`

```
> gradle clean badInstrumentClasses
> Task :clean UP-TO-DATE
> Task :badInstrumentClasses NO-SOURCE

BUILD SUCCESSFUL in 0s
3 actionable tasks: 2 executed, 1 up-to-date
```

There's nothing obviously wrong with this code, but you can see from the console output that the compilation task is missing. In this case you would need to add an explicit task dependency between `instrumentClasses` and `compileJava` via `dependsOn`. The use of `fileTree()` means that Gradle can't infer the task dependency itself.

One solution is to use the `TaskOutputs.files` property, as demonstrated by the following example:

Example 107. Setting up an inferred task dependency between output dir and input files

build.gradle

```
tasks.register('instrumentClasses', Instrument) {
    classFiles.from tasks.named('compileJava').map { it.outputs.files }
    destinationDir = file(layout.buildDirectory.dir('instrumented'))
}
```

build.gradle.kts

```
tasks.register<Instrument>("instrumentClasses") {
    classFiles.from(tasks.compileJava.map { it.outputs.files })
    destinationDir.set(file(layout.buildDirectory.dir("instrumented")))
}
```

Output of `gradle clean instrumentClasses`

```
> gradle clean instrumentClasses
> Task :clean UP-TO-DATE
> Task :compileJava
> Task :instrumentClasses

BUILD SUCCESSFUL in 0s
5 actionable tasks: 4 executed, 1 up-to-date
```

Alternatively, you can get Gradle to access the appropriate property itself by using one of `project.files()`, `project.layout.files()` or `project.objects.fileCollection()` in place of `project.fileTree()`:

Example 108. Setting up an inferred task dependency with `layout.files()`

build.gradle

```
tasks.register('instrumentClasses2', Instrument) {
    classFiles.from layout.files(tasks.named('compileJava'))
    destinationDir = file(layout.buildDirectory.dir('instrumented'))
}
```

build.gradle.kts

```
tasks.register<Instrument>("instrumentClasses2") {
    classFiles.from(layout.files(tasks.compileJava))
    destinationDir.set(file(layout.buildDirectory.dir("instrumented")))
}
```

Output of `gradle clean instrumentClasses2`

```
> gradle clean instrumentClasses2
> Task :clean UP-TO-DATE
> Task :compileJava
> Task :instrumentClasses2

BUILD SUCCESSFUL in 0s
5 actionable tasks: 4 executed, 1 up-to-date
```

Remember that `files()`, `layout.files()` and `objects.fileCollection()` can take tasks as arguments, whereas `fileTree()` cannot.

The downside of this approach is that all file outputs of the source task become the input files of the target — `instrumentClasses` in this case. That's fine as long as the source task only has a single file-based output, like the `JavaCompile` task. But if you have to link just one output property among several, then you need to explicitly tell Gradle which task generates the input files using the `builtBy` method:

Example 109. Setting up an inferred task dependency with builtBy()

build.gradle

```
tasks.register('instrumentClassesBuiltBy', Instrument) {
    classFiles.from fileTree(tasks.named('compileJava').map { it
.destinationDir }) {
        builtBy tasks.named('compileJava')
    }
    destinationDir = file(layout.buildDirectory.dir('instrumented'))
}
```

build.gradle.kts

```
tasks.register<Instrument>("instrumentClassesBuiltBy") {
    classFiles.from(fileTree(tasks.compileJava.map { it.destinationDir }) {
        builtBy(tasks.compileJava)
    })
    destinationDir.set(file(layout.buildDirectory.dir("instrumented")))
}
```

Output of `gradle clean instrumentClassesBuiltBy`

```
> gradle clean instrumentClassesBuiltBy
> Task :clean UP-TO-DATE
> Task :compileJava
> Task :instrumentClassesBuiltBy

BUILD SUCCESSFUL in 0s
5 actionable tasks: 4 executed, 1 up-to-date
```

You can of course just add an explicit task dependency via `dependsOn`, but the above approach provides more semantic meaning, explaining why `compileJava` has to run beforehand.

Disabling up-to-date checks

Gradle automatically handles up-to-date checks for output files and directories, but what if the task output is something else entirely? Perhaps it's an update to a web service or a database table. Or sometimes you have a task which should always run.

That's where the `doNotTrackState()` method on `Task` comes in. One can use this to disable up-to-date checks completely for a task, like so:

Example 110. Ignoring up-to-date checks

build.gradle

```
tasks.register('alwaysInstrumentClasses', Instrument) {
    classFiles.from layout.files(tasks.named('compileJava'))
    destinationDir = file(layout.buildDirectory.dir('instrumented'))
    doNotTrackState("Instrumentation needs to re-run every time")
}
```

build.gradle.kts

```
tasks.register<Instrument>("alwaysInstrumentClasses") {
    classFiles.from(layout.files(tasks.compileJava))
    destinationDir.set(file(layout.buildDirectory.dir("instrumented")))
    doNotTrackState("Instrumentation needs to re-run every time")
}
```

Output of `gradle clean alwaysInstrumentClasses`

```
> gradle clean alwaysInstrumentClasses
> Task :compileJava
> Task :alwaysInstrumentClasses

BUILD SUCCESSFUL in 0s
4 actionable tasks: 1 executed, 3 up-to-date
```

Output of `gradle alwaysInstrumentClasses`

```
> gradle alwaysInstrumentClasses
> Task :compileJava UP-TO-DATE
> Task :alwaysInstrumentClasses

BUILD SUCCESSFUL in 0s
4 actionable tasks: 1 executed, 3 up-to-date
```

If you are writing your own task that always should run, then you can also use the `@UntrackedTask` annotation on the task class instead of calling `Task.doNotTrackState()`.

Integrate an external tool which does its own up-to-date checking

Sometimes you want to integrate an external tool like Git or Npm, both of which do their own up-to-date checking. In that case it doesn't make much sense for Gradle to also do up-to-date checks. You can disable Gradle's up-to-date checks by using the `@UntrackedTask` annotation on the task wrapping

the tool. Alternatively, you can use the runtime API method `Task.doNotTrackState()`.

For example, let's say you want to implement a task which clones a Git repository.

Example 111. Task for Git clone

buildSrc/src/main/java/org/example/GitClone.java

```
@UntrackedTask(because = "Git tracks the state")
①
public abstract class GitClone extends DefaultTask {

    @Input
    public abstract Property<String> getRemoteUri();

    @Input
    public abstract Property<String> getCommitId();

    @OutputDirectory
    public abstract DirectoryProperty getDestinationDir();

    @TaskAction
    public void gitClone() throws IOException {
        File destinationDir = getDestinationDir().get().getAsFile()
.getAbsoluteFile(); ②
        String remoteUri = getRemoteUri().get();
        // Fetch origin or clone and checkout
        // ...
    }

}
```

build.gradle

```
tasks.register("cloneGradleProfiler", GitClone) {
    destinationDir = layout.buildDirectory.dir("gradle-profiler")
③
    remoteUri = "https://github.com/gradle/gradle-profiler.git"
    commitId = "d6c18a21ca6c45fd8a9db321de4478948bdf801b"
}
```

build.gradle.kts

```
tasks.register<GitClone>("cloneGradleProfiler") {
    destinationDir.set(layout.buildDirectory.dir("gradle-profiler"))
③
    remoteUri.set("https://github.com/gradle/gradle-profiler.git")
    commitId.set("d6c18a21ca6c45fd8a9db321de4478948bdf801b")
}
```

- (1) Declare the task as untracked.
- (2) Use the output directory to run the external tool.
- (3) Add the task and configure the output directory in your build.

Configure input normalization

For up to date checks and the build cache Gradle needs to determine if two task input properties have the same value. In order to do so, Gradle first normalizes both inputs and then compares the result. For example, for a compile classpath, Gradle extracts the ABI signature from the classes on the classpath and then compares signatures between the last Gradle run and the current Gradle run as described in Java compile avoidance.

Normalization applies to all zip files on the classpath (e.g. jars, wars, aars, apks, etc). This allows Gradle to recognize when two zip files are functionally the same, even though the zip files themselves might be slightly different due to metadata (such as timestamps or file order). Normalization applies not only to zip files directly on the classpath, but also to zip files nested inside directories or inside other zip files on the classpath.

It is possible to customize Gradle's built-in strategy for runtime classpath normalization. All inputs annotated with `@Classpath` are considered to be runtime classpaths.

Let's say you want to add a file `build-info.properties` to all your produced jar files which contains information about the build, e.g. the timestamp when the build started or some ID to identify the CI job that published the artifact. This file is only for auditing purposes, and has no effect on the outcome of running tests. Nonetheless, this file is part of the runtime classpath for the `test` task and changes on every build invocation. Therefore, the `test` would be never up-to-date or pulled from the build cache. In order to benefit from incremental builds again, you are able tell Gradle to ignore this file on the runtime classpath at the project level by using Project.normalization(org.gradle.api.Action) (in the *consuming* project):

Example 112. Runtime classpath normalization

build.gradle

```
normalization {
    runtimeClasspath {
        ignore 'build-info.properties'
    }
}
```

build.gradle.kts

```
normalization {
    runtimeClasspath {
        ignore("build-info.properties")
    }
}
```

If adding such a file to your jar files is something you do for all of the projects in your build, and you want to filter this file for all consumers, you should consider configuring such normalization in a convention plugin to share it between subprojects.

The effect of this configuration would be that changes to `build-info.properties` would be ignored for up-to-date checks and build cache key calculations. Note that this will not change the runtime behavior of the `test` task — i.e. any test is still able to load `build-info.properties` and the runtime classpath is still the same as before.

Properties file normalization

By default, properties files (i.e. files that end in a `.properties` extension) will be normalized to ignore differences in comments, whitespace and the order of properties. Gradle does this by loading the properties files and only considering the individual properties during up-to-date checks or build cache key calculations.

It is sometimes the case, though, that certain properties have a runtime impact, while others do not. If a property is changing that does not have an impact on the runtime classpath, it may be desirable to exclude it from up-to-date checks and build cache key calculations. However, excluding the entire file would also exclude the properties that do have a runtime impact. In this case, properties can be excluded selectively from any or all properties files on the runtime classpath.

A rule for ignoring properties can be applied to a specific set of files using the patterns described in RuntimeClasspathNormalization. In the event that a file matches a rule, but cannot be loaded as a properties file (e.g. because it is not formatted properly or uses a non-standard encoding), it will be incorporated into the up-to-date or build cache key calculation as a normal file. In other words, if the file cannot be loaded as a properties file, any changes to whitespace, property order, or

comments may cause the task to become out-of-date or cause a cache miss.

Example 113. Ignore a property in selected properties files

build.gradle

```
normalization {
    runtimeClasspath {
        properties('**/build-info.properties') {
            ignoreProperty 'timestamp'
        }
    }
}
```

build.gradle.kts

```
normalization {
    runtimeClasspath {
        properties("**/build-info.properties") {
            ignoreProperty("timestamp")
        }
    }
}
```

Example 114. Ignore a property in all properties files

build.gradle

```
normalization {
    runtimeClasspath {
        properties {
            ignoreProperty 'timestamp'
        }
    }
}
```

build.gradle.kts

```
normalization {
    runtimeClasspath {
        properties {
            ignoreProperty("timestamp")
        }
    }
}
```

Java `META-INF` normalization

For files in the `META-INF` directory of jar archives it's not always possible to ignore files completely due to their runtime impact.

Manifest files within `META-INF` are normalized to ignore comments, whitespace and order differences. Manifest attribute names are compared case-and-order insensitively. Manifest properties files are normalized according to Properties File Normalization.

Example 115. Ignore `META-INF` manifest attributes

build.gradle

```
normalization {
    runtimeClasspath {
        metaInf {
            ignoreAttribute("Implementation-Version")
        }
    }
}
```

build.gradle.kts

```
normalization {
    runtimeClasspath {
        metaInf {
            ignoreAttribute("Implementation-Version")
        }
    }
}
```

Example 116. Ignore `META-INF` property keys

build.gradle

```
normalization {
    runtimeClasspath {
        metaInf {
            ignoreProperty("app.version")
        }
    }
}
```

build.gradle.kts

```
normalization {
    runtimeClasspath {
        metaInf {
            ignoreProperty("app.version")
        }
    }
}
```

Example 117. Ignore `META-INF/MANIFEST.MF`

build.gradle

```
normalization {
    runtimeClasspath {
        metaInf {
            ignoreManifest()
        }
    }
}
```

build.gradle.kts

```
normalization {
    runtimeClasspath {
        metaInf {
            ignoreManifest()
        }
    }
}
```

Example 118. Ignore all files and directories inside `META-INF`

build.gradle

```
normalization {
    runtimeClasspath {
        metaInf {
            ignoreCompletely()
        }
    }
}
```

build.gradle.kts

```
normalization {
    runtimeClasspath {
        metaInf {
            ignoreCompletely()
        }
    }
}
```

Providing custom up-to-date logic

Gradle automatically handles up-to-date checks for output files and directories, but what if the task output is something else entirely? Perhaps it's an update to a web service or a database table. Gradle has no way of knowing how to check whether the task is up to date in such cases.

That's where the `upToDateWhen()` method on `TaskOutputs` comes in. This takes a predicate function that is used to determine whether a task is up to date or not. For example, you could read the version number of your database schema from the database. Or, you could check whether a particular record in a database table exists or has changed for example.

Just be aware that up-to-date checks should *save* you time. Don't add checks that cost as much or more time than the standard execution of the task. In fact, if a task ends up running frequently anyway, because it's rarely up to date, then it may not be worth having no up-to-date checks at all as described in Disabling up-to-date checks. Remember that your checks will always run if the task is in the execution task graph.

One common mistake is to use `upToDateWhen()` instead of `Task.onlyIf()`. If you want to skip a task on the basis of some condition unrelated to the task inputs and outputs, then you should use `onlyIf()`. For example, in cases where you want to skip a task when a particular property is set or not set.

Stale task outputs

When the Gradle version changes, Gradle detects that outputs from tasks that ran with older versions of Gradle need to be removed to ensure that the newest version of the tasks are starting from a known clean state.

NOTE	Automatic clean-up of stale output directories has only been implemented for the output of source sets (Java/Groovy/Scala compilation).

Task rules

Sometimes you want to have a task whose behavior depends on a large or infinite number value range of parameters. A very nice and expressive way to provide such tasks are task rules:

Example 119. Task rule

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->

    if (taskName.startsWith("ping")) {
        task(taskName) {
            doLast {
                println "Pinging: " + (taskName - 'ping')
            }
        }
    }
}
```

build.gradle.kts

```
tasks.addRule("Pattern: ping<ID>") {
    val taskName = this
    if (startsWith("ping")) {
        task(taskName) {
            doLast {
                println("Pinging: " + (taskName.replace("ping", "")))
            }
        }
    }
}
```

Output of `gradle -q pingServer1`

```
> gradle -q pingServer1
Pinging: Server1
```

The String parameter is used as a description for the rule, which is shown with `gradle tasks`.

Rules are not only used when calling tasks from the command line. You can also create dependsOn relations on rule based tasks:

Example 120. Dependency on rule based tasks

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->

    if (taskName.startsWith("ping")) {
        task(taskName) {
            doLast {
                println "Pinging: " + (taskName - 'ping')
            }
        }
    }
}

tasks.register('groupPing') {
    dependsOn 'pingServer1', 'pingServer2'
}
```

build.gradle.kts

```
tasks.addRule("Pattern: ping<ID>") {
    val taskName = this
    if (startsWith("ping")) {
        task(taskName) {
            doLast {
                println("Pinging: " + (taskName.replace("ping", "")))
            }
        }
    }
}

tasks.register("groupPing") {
    dependsOn("pingServer1", "pingServer2")
}
```

Output of `gradle -q groupPing`

```
> gradle -q groupPing
Pinging: Server1
Pinging: Server2
```

If you run `gradle -q tasks` you won't find a task named `pingServer1` or `pingServer2`, but this script is executing logic based on the request to run those tasks.

Finalizer tasks

Finalizer tasks are automatically added to the task graph when the finalized task is scheduled to run.

Example 121. Adding a task finalizer

build.gradle

```
def taskX = tasks.register('taskX') {
    doLast {
        println 'taskX'
    }
}
def taskY = tasks.register('taskY') {
    doLast {
        println 'taskY'
    }
}

taskX.configure { finalizedBy taskY }
```

build.gradle.kts

```
val taskX by tasks.registering {
    doLast {
        println("taskX")
    }
}
val taskY by tasks.registering {
    doLast {
        println("taskY")
    }
}

taskX { finalizedBy(taskY) }
```

Output of `gradle -q taskX`

```
> gradle -q taskX
taskX
taskY
```

Finalizer tasks will be executed even if the finalized task fails or if the finalized task is considered up to date.

Example 122. Task finalizer for a failing task

build.gradle

```
def taskX = tasks.register('taskX') {
    doLast {
        println 'taskX'
        throw new RuntimeException()
    }
}
def taskY = tasks.register('taskY') {
    doLast {
        println 'taskY'
    }
}

taskX.configure { finalizedBy taskY }
```

build.gradle.kts

```
val taskX by tasks.registering {
    doLast {
        println("taskX")
        throw RuntimeException()
    }
}
val taskY by tasks.registering {
    doLast {
        println("taskY")
    }
}

taskX { finalizedBy(taskY) }
```

Output of `gradle -q taskX`

```
> gradle -q taskX
taskX
taskY

FAILURE: Build failed with an exception.

* Where:
Build file '/home/user/gradle/samples/build.gradle' line: 4

* What went wrong:
Execution failed for task ':taskX'.
> java.lang.RuntimeException (no error message)

* Try:
> Run with --stacktrace option to get the stack trace.
> Run with --info or --debug option to get more log output.
> Run with --scan to get full insights.

* Get more help at https://help.gradle.org

BUILD FAILED in 0s
```

Finalizer tasks are useful in situations where the build creates a resource that has to be cleaned up regardless of the build failing or succeeding. An example of such a resource is a web container that is started before an integration test task and which should be always shut down, even if some of the tests fail.

To specify a finalizer task you use the Task.finalizedBy(java.lang.Object...) method. This method accepts a task instance, a task name, or any other input accepted by Task.dependsOn(java.lang.Object...).

Lifecycle tasks

Lifecycle tasks are tasks that do not do work themselves. They typically do not have any task actions. Lifecycle tasks can represent several concepts:

- a work-flow step (e.g., run all checks with `check`)
- a buildable thing (e.g., create a debug 32-bit executable for native components with `debug32MainExecutable`)
- a convenience task to execute many of the same logical tasks (e.g., run all compilation tasks with `compileAll`)

The Base Plugin defines several standard lifecycle tasks, such as `build`, `assemble`, and `check`. All the core language plugins, like the Java Plugin, apply the Base Plugin and hence have the same base set of lifecycle tasks.

Unless a lifecycle task has actions, its outcome is determined by its task dependencies. If any of those dependencies are executed, the lifecycle task will be considered `EXECUTED`. If all of the task dependencies are up to date, skipped or from cache, the lifecycle task will be considered `UP-TO-DATE`.

Summary

If you are coming from Ant, an enhanced Gradle task like *Copy* seems like a cross between an Ant target and an Ant task. Although Ant's tasks and targets are really different entities, Gradle combines these notions into a single entity. Simple Gradle tasks are like Ant's targets, but enhanced Gradle tasks also include aspects of Ant tasks. All of Gradle's tasks share a common API and you can create dependencies between them. These tasks are much easier to configure than an Ant task. They make full use of the type system, and are more expressive and easier to maintain.

Writing Build Scripts

This chapter looks at some of the details of writing a build script.

The Gradle build language

Gradle provides a *domain specific language*, or DSL, for describing builds. This build language is available in Groovy and Kotlin.

A Groovy build script can contain any Groovy language element.[4] A Kotlin build script can contain any Kotlin language element. Gradle assumes that each build script is encoded using UTF-8.

The Project API

Build scripts describe your build by configuring *projects*. A project is an abstract concept, but you typically map a Gradle project to a software component that needs to be built, like a library or an application. Each build script you have is associated with an object of type Project and as the build script executes, it configures this `Project`.

In fact, almost all top-level properties and blocks in a build script are part of the `Project` API. To demonstrate, take a look at this example build script that prints the name of its project, which is accessed via the Project.name property:

Example 123. Accessing property of the Project object

build.gradle

```
println name
println project.name
```

build.gradle.kts

```
println(name)
println(project.name)
```

Output of `gradle -q check`

```
> gradle -q check
project-api
project-api
```

Both `println` statements print out the same property. The first uses the top-level reference to the `name` property of the `Project` object. The other statement uses the `project` property available to any build script, which returns the associated `Project` object. Only if you define a property or a method which has the same name as a member of the `Project` object, would you need to use the `project` property.

Standard project properties

The `Project` object provides some standard properties, which are available in your build script. The following table lists a few of the commonly used ones.

Table 5. Project Properties

Name	Type	Default Value
`project`	Project	The `Project` instance
`name`	`String`	The name of the project directory.
`path`	`String`	The absolute path of the project.
`description`	`String`	A description for the project.
`projectDir`	`File`	The directory containing the build script.
`buildDir`	`File`	`projectDir/build`
`group`	`Object`	`unspecified`
`version`	`Object`	`unspecified`
`ant`	AntBuilder	An `AntBuilder` instance

IMPORTANT	*Script with other targets* The *build scripts* described here target `Project` objects. There are also settings scripts and init scripts that respectively target Settings and Gradle objects.

The script API

When Gradle executes a Groovy build script (`.gradle`), it compiles the script into a class which implements Script. This means that all of the properties and methods declared by the `Script` interface are available in your script.

When Gradle executes a Kotlin build script (`.gradle.kts`), it compiles the script into a subclass of KotlinBuildScript. This means that all of the visible properties and functions declared by the `KotlinBuildScript` type are available in your script. Also see the KotlinSettingsScript and KotlinInitScript types respectively for settings scripts and init scripts.

Declaring variables

There are two kinds of variables that can be declared in a build script: local variables and extra properties.

Local variables

Local variables are declared with the `def` keyword. They are only visible in the scope where they have been declared. Local variables are a feature of the underlying Groovy language.

Local variables are declared with the `val` keyword. They are only visible in the scope where they have been declared. Local variables are a feature of the underlying Kotlin language.

Example 124. Using local variables

build.gradle

```
def dest = 'dest'

tasks.register('copy', Copy) {
    from 'source'
    into dest
}
```

build.gradle.kts

```
val dest = "dest"

tasks.register<Copy>("copy") {
    from("source")
    into(dest)
}
```

Extra properties

All enhanced objects in Gradle's domain model can hold extra user-defined properties. This includes, but is not limited to, projects, tasks, and source sets.

Extra properties can be added, read and set via the owning object's `ext` property. Alternatively, an `ext` block can be used to add multiple properties at once.

Extra properties can be added, read and set via the owning object's `extra` property. Alternatively, they can be addressed via Kotlin delegated properties using `by extra`.

Example 125. Using extra properties

build.gradle

```
plugins {
    id 'java-library'
}

ext {
    springVersion = "3.1.0.RELEASE"
    emailNotification = "build@master.org"
}

sourceSets.all { ext.purpose = null }

sourceSets {
    main {
        purpose = "production"
    }
    test {
        purpose = "test"
    }
    plugin {
        purpose = "production"
    }
}

tasks.register('printProperties') {
    doLast {
        println springVersion
        println emailNotification
        sourceSets.matching { it.purpose == "production" }.each { println it
.name }
    }
}
```

build.gradle.kts

```
plugins {
    id("java-library")
}

val springVersion by extra("3.1.0.RELEASE")
val emailNotification by extra { "build@master.org" }

sourceSets.all { extra["purpose"] = null }

sourceSets {
    main {
        extra["purpose"] = "production"
    }
    test {
        extra["purpose"] = "test"
    }
    create("plugin") {
        extra["purpose"] = "production"
    }
}

tasks.register("printProperties") {
    doLast {
        println(springVersion)
        println(emailNotification)
        sourceSets.matching { it.extra["purpose"] == "production" }.forEach {
println(it.name) }
    }
}
```

Output of `gradle -q printProperties`

```
> gradle -q printProperties
3.1.0.RELEASE
build@master.org
main
plugin
```

In this example, an `ext` block adds two extra properties to the `project` object. Additionally, a property named `purpose` is added to each source set by setting `ext.purpose` to `null` (`null` is a permissible value). Once the properties have been added, they can be read and set like predefined properties.

In this example, two extra properties are added to the `project` object using `by extra`. Additionally, a property named `purpose` is added to each source set by setting `extra["purpose"]` to `null` (`null` is a

permissible value). Once the properties have been added, they can be read and set on `extra`.

By requiring special syntax for adding a property, Gradle can fail fast when an attempt is made to set a (predefined or extra) property but the property is misspelled or does not exist. Extra properties can be accessed from anywhere their owning object can be accessed, giving them a wider scope than local variables. Extra properties on a project are visible from its subprojects.

For further details on extra properties and their API, see the ExtraPropertiesExtension class in the API documentation.

Configuring arbitrary objects

You can configure arbitrary objects in the following very readable way.

Example 126. Configuring arbitrary objects

build.gradle

```
import java.text.FieldPosition

tasks.register('configure') {
    doLast {
        def pos = configure(new FieldPosition(10)) {
            beginIndex = 1
            endIndex = 5
        }
        println pos.beginIndex
        println pos.endIndex
    }
}
```

build.gradle.kts

```
import java.text.FieldPosition

tasks.register("configure") {
    doLast {
        val pos = FieldPosition(10).apply {
            beginIndex = 1
            endIndex = 5
        }
        println(pos.beginIndex)
        println(pos.endIndex)
    }
}
```

Output of `gradle -q configure`

```
> gradle -q configure
1
5
```

Configuring arbitrary objects using an external script

You can also configure arbitrary objects using an external script.

CAUTION	*Only supported from a Groovy script* Configuring arbitrary objects using an external script is not yet supported by the Kotlin DSL. See gradle/kotlin-dsl#659 for more information.

Example 127. Configuring arbitrary objects using a script

build.gradle

```
tasks.register('configure') {
    doLast {
        def pos = new java.text.FieldPosition(10)
        // Apply the script
        apply from: 'other.gradle', to: pos
        println pos.beginIndex
        println pos.endIndex
    }
}
```

other.gradle

```
// Set properties.
beginIndex = 1
endIndex = 5
```

Output of `gradle -q configure`

```
> gradle -q configure
1
5
```

Some Groovy basics

TIP	Looking for some Kotlin basics, the Kotlin reference documentation and Kotlin Koans should be useful to you.

The Groovy language provides plenty of features for creating DSLs, and the Gradle build language takes advantage of these. Understanding how the build language works will help you when you write your build script, and in particular, when you start to write custom plugins and tasks.

Groovy JDK

Groovy adds lots of useful methods to the standard Java classes. For example, `Iterable` gets an `each` method, which iterates over the elements of the `Iterable`:

Example 128. Groovy JDK methods

build.gradle

```
// Iterable gets an each() method
configurations.runtimeClasspath.each { File f -> println f }
```

Have a look at https://groovy-lang.org/gdk.html for more details.

Property accessors

Groovy automatically converts a property reference into a call to the appropriate getter or setter method.

Example 129. Property accessors

build.gradle

```
// Using a getter method
println project.buildDir
println getProject().getBuildDir()

// Using a setter method
project.buildDir = 'target'
getProject().setBuildDir('target')
```

Optional parentheses on method calls

Parentheses are optional for method calls.

Example 130. Method call without parentheses

build.gradle

```
test.systemProperty 'some.prop', 'value'
test.systemProperty('some.prop', 'value')
```

List and map literals

Groovy provides some shortcuts for defining `List` and `Map` instances. Both kinds of literals are straightforward, but map literals have some interesting twists.

For instance, the "`apply`" method (where you typically apply plugins) actually takes a map parameter. However, when you have a line like "`apply plugin:'java'`", you aren't actually using a map literal, you're actually using "named parameters", which have almost exactly the same syntax as a map literal (without the wrapping brackets). That named parameter list gets converted to a map when the method is called, but it doesn't start out as a map.

Example 131. List and map literals

build.gradle

```
// List literal
test.includes = ['org/gradle/api/**', 'org/gradle/internal/**']

List<String> list = new ArrayList<String>()
list.add('org/gradle/api/**')
list.add('org/gradle/internal/**')
test.includes = list

// Map literal.
Map<String, String> map = [key1:'value1', key2: 'value2']

// Groovy will coerce named arguments
// into a single map argument
apply plugin: 'java'
```

Closures as the last parameter in a method

The Gradle DSL uses closures in many places. You can find out more about closures here. When the last parameter of a method is a closure, you can place the closure after the method call:

Example 132. Closure as method parameter

build.gradle

```
repositories {
    println "in a closure"
}
repositories() { println "in a closure" }
repositories({ println "in a closure" })
```

Closure delegate

Each closure has a `delegate` object, which Groovy uses to look up variable and method references which are not local variables or parameters of the closure. Gradle uses this for *configuration*

closures, where the `delegate` object is set to the object to be configured.

Example 133. Closure delegates

build.gradle

```
dependencies {
    assert delegate == project.dependencies
    testImplementation('junit:junit:4.13')
    delegate.testImplementation('junit:junit:4.13')
}
```

Default imports

To make build scripts more concise, Gradle automatically adds a set of import statements to the Gradle scripts. This means that instead of using `throw new org.gradle.api.tasks.StopExecutionException()` you can just type `throw new StopExecutionException()` instead.

Listed below are the imports added to each script:

Gradle default imports

```
import org.gradle.*
import org.gradle.api.*
import org.gradle.api.artifacts.*
import org.gradle.api.artifacts.component.*
import org.gradle.api.artifacts.dsl.*
import org.gradle.api.artifacts.ivy.*
import org.gradle.api.artifacts.maven.*
import org.gradle.api.artifacts.query.*
import org.gradle.api.artifacts.repositories.*
import org.gradle.api.artifacts.result.*
import org.gradle.api.artifacts.transform.*
import org.gradle.api.artifacts.type.*
import org.gradle.api.artifacts.verification.*
import org.gradle.api.attributes.*
import org.gradle.api.attributes.java.*
import org.gradle.api.attributes.plugin.*
import org.gradle.api.capabilities.*
import org.gradle.api.component.*
import org.gradle.api.credentials.*
import org.gradle.api.distribution.*
import org.gradle.api.distribution.plugins.*
import org.gradle.api.execution.*
import org.gradle.api.file.*
import org.gradle.api.initialization.*
import org.gradle.api.initialization.definition.*
```

```
import org.gradle.api.initialization.dsl.*
import org.gradle.api.initialization.resolve.*
import org.gradle.api.invocation.*
import org.gradle.api.java.archives.*
import org.gradle.api.jvm.*
import org.gradle.api.launcher.cli.*
import org.gradle.api.logging.*
import org.gradle.api.logging.configuration.*
import org.gradle.api.model.*
import org.gradle.api.plugins.*
import org.gradle.api.plugins.antlr.*
import org.gradle.api.plugins.catalog.*
import org.gradle.api.plugins.jvm.*
import org.gradle.api.plugins.quality.*
import org.gradle.api.plugins.scala.*
import org.gradle.api.provider.*
import org.gradle.api.publish.*
import org.gradle.api.publish.ivy.*
import org.gradle.api.publish.ivy.plugins.*
import org.gradle.api.publish.ivy.tasks.*
import org.gradle.api.publish.maven.*
import org.gradle.api.publish.maven.plugins.*
import org.gradle.api.publish.maven.tasks.*
import org.gradle.api.publish.plugins.*
import org.gradle.api.publish.tasks.*
import org.gradle.api.reflect.*
import org.gradle.api.reporting.*
import org.gradle.api.reporting.components.*
import org.gradle.api.reporting.dependencies.*
import org.gradle.api.reporting.dependents.*
import org.gradle.api.reporting.model.*
import org.gradle.api.reporting.plugins.*
import org.gradle.api.resources.*
import org.gradle.api.services.*
import org.gradle.api.specs.*
import org.gradle.api.tasks.*
import org.gradle.api.tasks.ant.*
import org.gradle.api.tasks.application.*
import org.gradle.api.tasks.bundling.*
import org.gradle.api.tasks.compile.*
import org.gradle.api.tasks.diagnostics.*
import org.gradle.api.tasks.diagnostics.configurations.*
import org.gradle.api.tasks.incremental.*
import org.gradle.api.tasks.javadoc.*
import org.gradle.api.tasks.options.*
import org.gradle.api.tasks.scala.*
import org.gradle.api.tasks.testing.*
import org.gradle.api.tasks.testing.junit.*
import org.gradle.api.tasks.testing.junitplatform.*
import org.gradle.api.tasks.testing.testng.*
import org.gradle.api.tasks.util.*
```

```
import org.gradle.api.tasks.wrapper.*
import org.gradle.api.toolchain.management.*
import org.gradle.authentication.*
import org.gradle.authentication.aws.*
import org.gradle.authentication.http.*
import org.gradle.build.event.*
import org.gradle.buildinit.*
import org.gradle.buildinit.plugins.*
import org.gradle.buildinit.tasks.*
import org.gradle.caching.*
import org.gradle.caching.configuration.*
import org.gradle.caching.http.*
import org.gradle.caching.local.*
import org.gradle.concurrent.*
import org.gradle.external.javadoc.*
import org.gradle.ide.visualstudio.*
import org.gradle.ide.visualstudio.plugins.*
import org.gradle.ide.visualstudio.tasks.*
import org.gradle.ide.xcode.*
import org.gradle.ide.xcode.plugins.*
import org.gradle.ide.xcode.tasks.*
import org.gradle.ivy.*
import org.gradle.jvm.*
import org.gradle.jvm.application.scripts.*
import org.gradle.jvm.application.tasks.*
import org.gradle.jvm.tasks.*
import org.gradle.jvm.toolchain.*
import org.gradle.language.*
import org.gradle.language.assembler.*
import org.gradle.language.assembler.plugins.*
import org.gradle.language.assembler.tasks.*
import org.gradle.language.base.*
import org.gradle.language.base.artifact.*
import org.gradle.language.base.compile.*
import org.gradle.language.base.plugins.*
import org.gradle.language.base.sources.*
import org.gradle.language.c.*
import org.gradle.language.c.plugins.*
import org.gradle.language.c.tasks.*
import org.gradle.language.cpp.*
import org.gradle.language.cpp.plugins.*
import org.gradle.language.cpp.tasks.*
import org.gradle.language.java.artifact.*
import org.gradle.language.jvm.tasks.*
import org.gradle.language.nativeplatform.*
import org.gradle.language.nativeplatform.tasks.*
import org.gradle.language.objectivec.*
import org.gradle.language.objectivec.plugins.*
import org.gradle.language.objectivec.tasks.*
import org.gradle.language.objectivecpp.*
import org.gradle.language.objectivecpp.plugins.*
```

```
import org.gradle.language.objectivecpp.tasks.*
import org.gradle.language.plugins.*
import org.gradle.language.rc.*
import org.gradle.language.rc.plugins.*
import org.gradle.language.rc.tasks.*
import org.gradle.language.scala.tasks.*
import org.gradle.language.swift.*
import org.gradle.language.swift.plugins.*
import org.gradle.language.swift.tasks.*
import org.gradle.maven.*
import org.gradle.model.*
import org.gradle.nativeplatform.*
import org.gradle.nativeplatform.platform.*
import org.gradle.nativeplatform.plugins.*
import org.gradle.nativeplatform.tasks.*
import org.gradle.nativeplatform.test.*
import org.gradle.nativeplatform.test.cpp.*
import org.gradle.nativeplatform.test.cpp.plugins.*
import org.gradle.nativeplatform.test.cunit.*
import org.gradle.nativeplatform.test.cunit.plugins.*
import org.gradle.nativeplatform.test.cunit.tasks.*
import org.gradle.nativeplatform.test.googletest.*
import org.gradle.nativeplatform.test.googletest.plugins.*
import org.gradle.nativeplatform.test.plugins.*
import org.gradle.nativeplatform.test.tasks.*
import org.gradle.nativeplatform.test.xctest.*
import org.gradle.nativeplatform.test.xctest.plugins.*
import org.gradle.nativeplatform.test.xctest.tasks.*
import org.gradle.nativeplatform.toolchain.*
import org.gradle.nativeplatform.toolchain.plugins.*
import org.gradle.normalization.*
import org.gradle.platform.*
import org.gradle.platform.base.*
import org.gradle.platform.base.binary.*
import org.gradle.platform.base.component.*
import org.gradle.platform.base.plugins.*
import org.gradle.plugin.devel.*
import org.gradle.plugin.devel.plugins.*
import org.gradle.plugin.devel.tasks.*
import org.gradle.plugin.management.*
import org.gradle.plugin.use.*
import org.gradle.plugins.ear.*
import org.gradle.plugins.ear.descriptor.*
import org.gradle.plugins.ide.*
import org.gradle.plugins.ide.api.*
import org.gradle.plugins.ide.eclipse.*
import org.gradle.plugins.ide.idea.*
import org.gradle.plugins.signing.*
import org.gradle.plugins.signing.signatory.*
import org.gradle.plugins.signing.signatory.pgp.*
import org.gradle.plugins.signing.type.*
```

```
import org.gradle.plugins.signing.type.pgp.*
import org.gradle.process.*
import org.gradle.swiftpm.*
import org.gradle.swiftpm.plugins.*
import org.gradle.swiftpm.tasks.*
import org.gradle.testing.base.*
import org.gradle.testing.base.plugins.*
import org.gradle.testing.jacoco.plugins.*
import org.gradle.testing.jacoco.tasks.*
import org.gradle.testing.jacoco.tasks.rules.*
import org.gradle.testkit.runner.*
import org.gradle.util.*
import org.gradle.vcs.*
import org.gradle.vcs.git.*
import org.gradle.work.*
import org.gradle.workers.*
```

Working With Files

Almost every Gradle build interacts with files in some way: think source files, file dependencies, reports and so on. That's why Gradle comes with a comprehensive API that makes it simple to perform the file operations you need.

The API has two parts to it:

- Specifying which files and directories to process
- Specifying what to do with them

The File paths in depth section covers the first of these in detail, while subsequent sections, like File copying in depth, cover the second. To begin with, we'll show you examples of the most common scenarios that users encounter.

Copying a single file

You copy a file by creating an instance of Gradle's builtin Copy task and configuring it with the location of the file and where you want to put it. This example mimics copying a generated report into a directory that will be packed into an archive, such as a ZIP or TAR:

Example 134. How to copy a single file

build.gradle

```
tasks.register('copyReport', Copy) {
    from layout.buildDirectory.file("reports/my-report.pdf")
    into layout.buildDirectory.dir("toArchive")
}
```

build.gradle.kts

```
tasks.register<Copy>("copyReport") {
    from(layout.buildDirectory.file("reports/my-report.pdf"))
    into(layout.buildDirectory.dir("toArchive"))
}
```

The Project.file(java.lang.Object) method is used to create a file or directory path relative to the current project and is a common way to make build scripts work regardless of the project path. The file and directory paths are then used to specify what file to copy using Copy.from(java.lang.Object...) and which directory to copy it to using Copy.into(java.lang.Object).

You can even use the path directly without the `file()` method, as explained early in the section File copying in depth:

Example 135. Using implicit string paths

build.gradle

```
tasks.register('copyReport2', Copy) {
    from "$buildDir/reports/my-report.pdf"
    into "$buildDir/toArchive"
}
```

build.gradle.kts

```
tasks.register<Copy>("copyReport2") {
    from("$buildDir/reports/my-report.pdf")
    into("$buildDir/toArchive")
}
```

Although hard-coded paths make for simple examples, they also make the build brittle. It's better to use a reliable, single source of truth, such as a task or shared project property. In the following modified example, we use a report task defined elsewhere that has the report's location stored in its `outputFile` property:

Example 136. Prefer task/project properties over hard-coded paths

build.gradle

```
tasks.register('copyReport3', Copy) {
    from myReportTask.outputFile
    into archiveReportsTask.dirToArchive
}
```

build.gradle.kts

```
tasks.register<Copy>("copyReport3") {
    val outputFile: File by myReportTask.get().extra
    val dirToArchive: File by archiveReportsTask.get().extra
    from(outputFile)
    into(dirToArchive)
}
```

We have also assumed that the reports will be archived by `archiveReportsTask`, which provides us with the directory that will be archived and hence where we want to put the copies of the reports.

Copying multiple files

You can extend the previous examples to multiple files very easily by providing multiple arguments to `from()`:

Example 137. Using multiple arguments with from()

build.gradle

```
tasks.register('copyReportsForArchiving', Copy) {
    from layout.buildDirectory.file("reports/my-report.pdf"), layout
.projectDirectory.file("src/docs/manual.pdf")
    into layout.buildDirectory.dir("toArchive")
}
```

build.gradle.kts

```
tasks.register<Copy>("copyReportsForArchiving") {
    from(layout.buildDirectory.file("reports/my-report.pdf"),
layout.projectDirectory.file("src/docs/manual.pdf"))
    into(layout.buildDirectory.dir("toArchive"))
}
```

Two files are now copied into the archive directory. You can also use multiple `from()` statements to do the same thing, as shown in the first example of the section File copying in depth.

Now consider another example: what if you want to copy all the PDFs in a directory without having to specify each one? To do this, attach inclusion and/or exclusion patterns to the copy specification. Here we use a string pattern to include PDFs only:

Example 138. Using a flat filter

build.gradle

```
tasks.register('copyPdfReportsForArchiving', Copy) {
    from layout.buildDirectory.dir("reports")
    include "*.pdf"
    into layout.buildDirectory.dir("toArchive")
}
```

build.gradle.kts

```
tasks.register<Copy>("copyPdfReportsForArchiving") {
    from(layout.buildDirectory.dir("reports"))
    include("*.pdf")
    into(layout.buildDirectory.dir("toArchive"))
}
```

One thing to note, as demonstrated in the following diagram, is that only the PDFs that reside directly in the `reports` directory are copied:

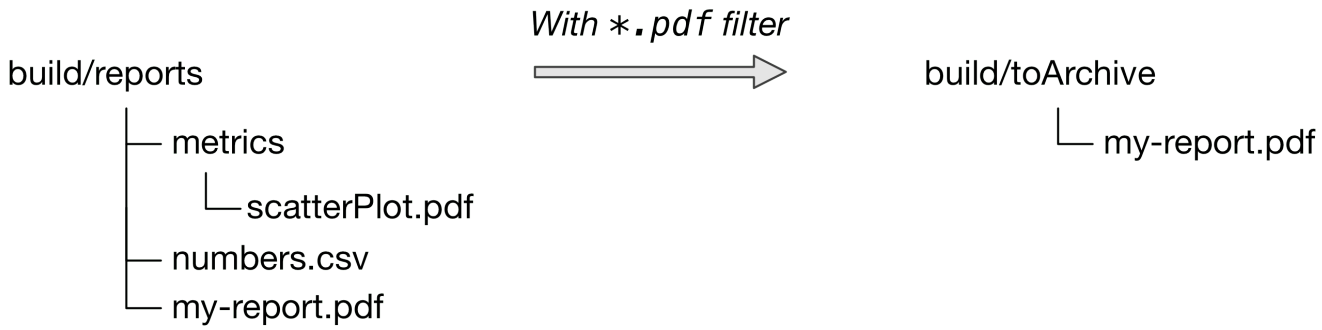


Figure 7. The effect of a flat filter on copying

You can include files in subdirectories by using an Ant-style glob pattern (`**/*`), as done in this updated example:

Example 139. Using a deep filter

build.gradle

```
tasks.register('copyAllPdfReportsForArchiving', Copy) {
    from layout.buildDirectory.dir("reports")
    include "**/*.pdf"
    into layout.buildDirectory.dir("toArchive")
}
```

build.gradle.kts

```
tasks.register<Copy>("copyAllPdfReportsForArchiving") {
    from(layout.buildDirectory.dir("reports"))
    include("**/*.pdf")
    into(layout.buildDirectory.dir("toArchive"))
}
```

This task has the following effect:

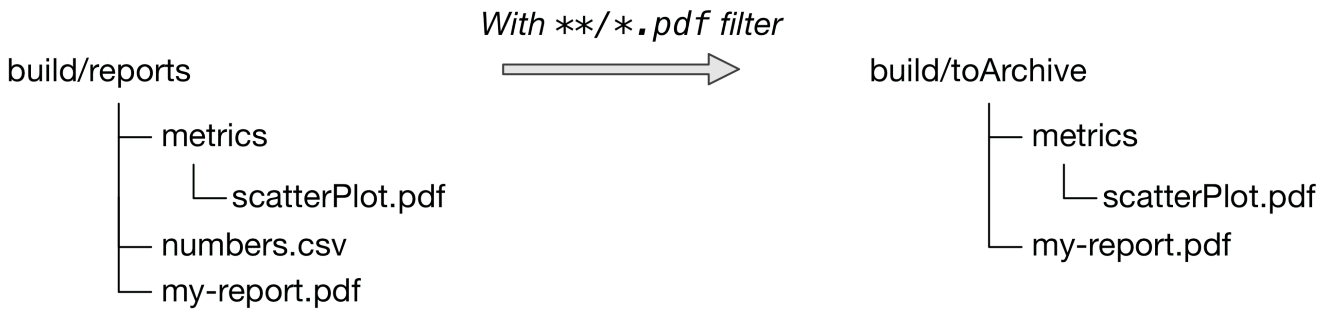


Figure 8. The effect of a deep filter on copying

One thing to bear in mind is that a deep filter like this has the side effect of copying the directory structure below `reports` as well as the files. If you just want to copy the files without the directory structure, you need to use an explicit `fileTree(dir) { includes }.files` expression. We talk more about the difference between file trees and file collections in the File trees section.

This is just one of the variations in behavior you're likely to come across when dealing with file operations in Gradle builds. Fortunately, Gradle provides elegant solutions to almost all those use cases. Read the *in-depth* sections later in the chapter for more detail on how the file operations work in Gradle and what options you have for configuring them.

Copying directory hierarchies

You may have a need to copy not just files, but the directory structure they reside in as well. This is the default behavior when you specify a directory as the `from()` argument, as demonstrated by the following example that copies everything in the `reports` directory, including all its subdirectories, to

the destination:

Example 140. Copying an entire directory

build.gradle

```
tasks.register('copyReportsDirForArchiving', Copy) {
    from layout.buildDirectory.dir("reports")
    into layout.buildDirectory.dir("toArchive")
}
```

build.gradle.kts

```
tasks.register<Copy>("copyReportsDirForArchiving") {
    from(layout.buildDirectory.dir("reports"))
    into(layout.buildDirectory.dir("toArchive"))
}
```

The key aspect that users struggle with is controlling how much of the directory structure goes to the destination. In the above example, do you get a `toArchive/reports` directory or does everything in `reports` go straight into `toArchive`? The answer is the latter. If a directory is part of the `from()` path, then it *won't* appear in the destination.

So how do you ensure that `reports` itself is copied across, but not any other directory in `$buildDir`? The answer is to add it as an include pattern:

Example 141. Copying an entire directory, including itself

build.gradle

```
tasks.register('copyReportsDirForArchiving2', Copy) {
    from(layout.buildDirectory) {
        include "reports/**"
    }
    into layout.buildDirectory.dir("toArchive")
}
```

build.gradle.kts

```
tasks.register<Copy>("copyReportsDirForArchiving2") {
    from(layout.buildDirectory) {
        include("reports/**")
    }
    into(layout.buildDirectory.dir("toArchive"))
}
```

You'll get the same behavior as before except with one extra level of directory in the destination, i.e. `toArchive/reports`.

One thing to note is how the `include()` directive applies only to the `from()`, whereas the directive in the previous section applied to the whole task. These different levels of granularity in the copy specification allow you to easily handle most requirements that you will come across. You can learn more about this in the section on child specifications.

Creating archives (zip, tar, etc.)

From the perspective of Gradle, packing files into an archive is effectively a copy in which the destination is the archive file rather than a directory on the file system. This means that creating archives looks a lot like copying, with all of the same features!

The simplest case involves archiving the entire contents of a directory, which this example demonstrates by creating a ZIP of the `toArchive` directory:

Example 142. Archiving a directory as a ZIP

build.gradle

```
tasks.register('packageDistribution', Zip) {
    archiveFileName = "my-distribution.zip"
    destinationDirectory = layout.buildDirectory.dir('dist')

    from layout.buildDirectory.dir("toArchive")
}
```

build.gradle.kts

```
tasks.register<Zip>("packageDistribution") {
    archiveFileName.set("my-distribution.zip")
    destinationDirectory.set(layout.buildDirectory.dir("dist"))

    from(layout.buildDirectory.dir("toArchive"))
}
```

Notice how we specify the destination and name of the archive instead of an `into()`: both are required. You often won't see them explicitly set, because most projects apply the Base Plugin. It provides some conventional values for those properties. The next example demonstrates this and you can learn more about the conventions in the archive naming section.

Each type of archive has its own task type, the most common ones being Zip, Tar and Jar. They all share most of the configuration options of `Copy`, including filtering and renaming.

One of the most common scenarios involves copying files into specified subdirectories of the archive. For example, let's say you want to package all PDFs into a `docs` directory in the root of the archive. This `docs` directory doesn't exist in the source location, so you have to create it as part of the archive. You do this by adding an `into()` declaration for just the PDFs:

Example 143. Using the Base Plugin for its archive name convention

build.gradle

```
plugins {
    id 'base'
}

version = "1.0.0"

tasks.register('packageDistribution', Zip) {
    from(layout.buildDirectory.dir("toArchive")) {
        exclude "**/*.pdf"
    }

    from(layout.buildDirectory.dir("toArchive")) {
        include "**/*.pdf"
        into "docs"
    }
}
```

build.gradle.kts

```
plugins {
    base
}

version = "1.0.0"

tasks.register<Zip>("packageDistribution") {
    from(layout.buildDirectory.dir("toArchive")) {
        exclude("**/*.pdf")
    }

    from(layout.buildDirectory.dir("toArchive")) {
        include("**/*.pdf")
        into("docs")
    }
}
```

As you can see, you can have multiple `from()` declarations in a copy specification, each with its own configuration. See Using child copy specifications for more information on this feature.

Unpacking archives

Archives are effectively self-contained file systems, so unpacking them is a case of copying the files from that file system onto the local file system — or even into another archive. Gradle enables this by providing some wrapper functions that make archives available as hierarchical collections of files (file trees).

The two functions of interest are Project.zipTree(java.lang.Object) and Project.tarTree(java.lang.Object), which produce a FileTree from a corresponding archive file. That file tree can then be used in a `from()` specification, like so:

Example 144. Unpacking a ZIP file

build.gradle

```
tasks.register('unpackFiles', Copy) {
    from zipTree("src/resources/thirdPartyResources.zip")
    into layout.buildDirectory.dir("resources")
}
```

build.gradle.kts

```
tasks.register<Copy>("unpackFiles") {
    from(zipTree("src/resources/thirdPartyResources.zip"))
    into(layout.buildDirectory.dir("resources"))
}
```

As with a normal copy, you can control which files are unpacked via filters and even rename files as they are unpacked.

More advanced processing can be handled by the eachFile() method. For example, you might need to extract different subtrees of the archive into different paths within the destination directory. The following sample uses the method to extract the files within the archive's `libs` directory into the root destination directory, rather than into a `libs` subdirectory:

Example 145. Unpacking a subset of a ZIP file

build.gradle

```
tasks.register('unpackLibsDirectory', Copy) {
    from(zipTree("src/resources/thirdPartyResources.zip")) {
        include "libs/**"  ①
        eachFile { fcd ->
            fcd.relativePath = new RelativePath(true, fcd.relativePath
.segments.drop(1))  ②
        }
        includeEmptyDirs = false  ③
    }
    into layout.buildDirectory.dir("resources")
}
```

build.gradle.kts

```
tasks.register<Copy>("unpackLibsDirectory") {
    from(zipTree("src/resources/thirdPartyResources.zip")) {
        include("libs/**")  ①
        eachFile {
            relativePath = RelativePath(true,
*relativePath.segments.drop(1).toTypedArray())  ②
        }
        includeEmptyDirs = false  ③
    }
    into(layout.buildDirectory.dir("resources"))
}
```

① Extracts only the subset of files that reside in the `libs` directory

② Remaps the path of the extracting files into the destination directory by dropping the `libs` segment from the file path

③ Ignores the empty directories resulting from the remapping, see Caution note below

CAUTION	You can not change the destination path of empty directories with this technique. You can learn more in this issue.

If you're a Java developer and are wondering why there is no `jarTree()` method, that's because `zipTree()` works perfectly well for JARs, WARs and EARs.

Creating "uber" or "fat" JARs

In the Java space, applications and their dependencies typically used to be packaged as separate

JARs within a single distribution archive. That still happens, but there is another approach that is now common: placing the classes and resources of the dependencies directly into the application JAR, creating what is known as an uber or fat JAR.

Gradle makes this approach easy to accomplish. Consider the aim: to copy the contents of other JAR files into the application JAR. All you need for this is the Project.zipTree(java.lang.Object) method and the Jar task, as demonstrated by the `uberJar` task in the following example:

Example 146. Creating a Java uber or fat JAR

build.gradle

```
plugins {
    id 'java'
}

version = '1.0.0'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'commons-io:commons-io:2.6'
}

tasks.register('uberJar', Jar) {
    archiveClassifier = 'uber'

    from sourceSets.main.output

    dependsOn configurations.runtimeClasspath
    from {
        configurations.runtimeClasspath.findAll { it.name.endsWith('jar') }
.collect { zipTree(it) }
    }
}
```

build.gradle.kts

```
plugins {
    java
}

version = "1.0.0"

repositories {
    mavenCentral()
}

dependencies {
    implementation("commons-io:commons-io:2.6")
}

tasks.register<Jar>("uberJar") {
    archiveClassifier.set("uber")

    from(sourceSets.main.get().output)

    dependsOn(configurations.runtimeClasspath)
    from({
        configurations.runtimeClasspath.get().filter {
it.name.endsWith("jar") }.map { zipTree(it) }
    })
}
```

In this case, we're taking the runtime dependencies of the project — `configurations.runtimeClasspath.files` — and wrapping each of the JAR files with the `zipTree()` method. The result is a collection of ZIP file trees, the contents of which are copied into the uber JAR alongside the application classes.

Creating directories

Many tasks need to create directories to store the files they generate, which is why Gradle automatically manages this aspect of tasks when they explicitly define file and directory outputs. You can learn about this feature in the incremental build section of the user manual. All core Gradle tasks ensure that any output directories they need are created if necessary using this mechanism.

In cases where you need to create a directory manually, you can use the Project.mkdir(java.lang.Object) method from within your build scripts or custom task implementations. Here's a simple example that creates a single `images` directory in the project folder:

Example 147. Manually creating a directory

build.gradle

```
tasks.register('ensureDirectory') {
    doLast {
        mkdir "images"
    }
}
```

build.gradle.kts

```
tasks.register("ensureDirectory") {
    doLast {
        mkdir("images")
    }
}
```

As described in the Apache Ant manual, the `mkdir` task will automatically create all necessary directories in the given path and will do nothing if the directory already exists.

Moving files and directories

Gradle has no API for moving files and directories around, but you can use the Apache Ant integration to easily do that, as shown in this example:

Example 148. Moving a directory using the Ant task

build.gradle

```
tasks.register('moveReports') {
    doLast {
        ant.move file: "${buildDir}/reports",
                 todir: "${buildDir}/toArchive"
    }
}
```

build.gradle.kts

```
tasks.register("moveReports") {
    doLast {
        ant.withGroovyBuilder {
            "move"("file" to "${buildDir}/reports", "todir" to
"${buildDir}/toArchive")
        }
    }
}
```

This is not a common requirement and should be used sparingly as you lose information and can easily break a build. It's generally preferable to copy directories and files instead.

Renaming files on copy

The files used and generated by your builds sometimes don't have names that suit, in which case you want to rename those files as you copy them. Gradle allows you to do this as part of a copy specification using the `rename()` configuration.

The following example removes the "-staging-" marker from the names of any files that have it:

Example 149. Renaming files as they are copied

build.gradle

```
tasks.register('copyFromStaging', Copy) {
    from "src/main/webapp"
    into layout.buildDirectory.dir('explodedWar')

    rename '(.+)-staging(.+)', '$1$2'
}
```

build.gradle.kts

```
tasks.register<Copy>("copyFromStaging") {
    from("src/main/webapp")
    into(layout.buildDirectory.dir("explodedWar"))

    rename("(.+)-staging(.+)", "$1$2")
}
```

You can use regular expressions for this, as in the above example, or closures that use more complex logic to determine the target filename. For example, the following task truncates filenames:

Example 150. Truncating filenames as they are copied

build.gradle

```
tasks.register('copyWithTruncate', Copy) {
    from layout.buildDirectory.dir("reports")
    rename { String filename ->
        if (filename.size() > 10) {
            return filename[0..7] + "~" + filename.size()
        }
        else return filename
    }
    into layout.buildDirectory.dir("toArchive")
}
```

build.gradle.kts

```
tasks.register<Copy>("copyWithTruncate") {
    from(layout.buildDirectory.dir("reports"))
    rename { filename: String ->
        if (filename.length > 10) {
            filename.slice(0..7) + "~" + filename.length
        }
        else filename
    }
    into(layout.buildDirectory.dir("toArchive"))
}
```

As with filtering, you can also apply renaming to a subset of files by configuring it as part of a child specification on a `from()`.

Deleting files and directories

You can easily delete files and directories using either the Delete task or the Project.delete(org.gradle.api.Action) method. In both cases, you specify which files and directories to delete in a way supported by the Project.files(java.lang.Object...) method.

For example, the following task deletes the entire contents of a build's output directory:

Example 151. Deleting a directory

build.gradle

```
tasks.register('myClean', Delete) {
    delete buildDir
}
```

build.gradle.kts

```
tasks.register<Delete>("myClean") {
    delete(buildDir)
}
```

If you want more control over which files are deleted, you can't use inclusions and exclusions in the same way as for copying files. Instead, you have to use the builtin filtering mechanisms of `FileCollection` and `FileTree`. The following example does just that to clear out temporary files from a source directory:

Example 152. Deleting files matching a specific pattern

build.gradle

```
tasks.register('cleanTempFiles', Delete) {
    delete fileTree("src").matching {
        include "**/*.tmp"
    }
}
```

build.gradle.kts

```
tasks.register<Delete>("cleanTempFiles") {
    delete(fileTree("src").matching {
        include("**/*.tmp")
    })
}
```

You'll learn more about file collections and file trees in the next section.

File paths in depth

In order to perform some action on a file, you need to know where it is, and that's the information provided by file paths. Gradle builds on the standard Java `File` class, which represents the location of a single file, and provides new APIs for dealing with collections of paths. This section shows you how to use the Gradle APIs to specify file paths for use in tasks and file operations.

But first, an important note on using hard-coded file paths in your builds.

On hard-coded file paths

Many examples in this chapter use hard-coded paths as string literals. This makes them easy to understand, but it's not good practice for real builds. The problem is that paths often change and the more places you need to change them, the more likely you are to miss one and break the build.

Where possible, you should use tasks, task properties, and project properties — in that order of preference — to configure file paths. For example, if you were to create a task that packages the compiled classes of a Java application, you should aim for something like this:

Example 153. How to minimize the number of hard-coded paths in your build

build.gradle

```
def archivesDirPath = layout.buildDirectory.dir('archives')

tasks.register('packageClasses', Zip) {
    archiveAppendix = "classes"
    destinationDirectory = archivesDirPath

    from compileJava
}
```

build.gradle.kts

```
val archivesDirPath = layout.buildDirectory.dir("archives")

tasks.register<Zip>("packageClasses") {
    archiveAppendix.set("classes")
    destinationDirectory.set(archivesDirPath)

    from(tasks.compileJava)
}
```

See how we're using the `compileJava` task as the source of the files to package and we've created a project property `archivesDirPath` to store the location where we put archives, on the basis we're

likely to use it elsewhere in the build.

Using a task directly as an argument like this relies on it having defined outputs, so it won't always be possible. In addition, this example could be improved further by relying on the Java plugin's convention for `destinationDirectory` rather than overriding it, but it does demonstrate the use of project properties.

Single files and directories

Gradle provides the Project.file(java.lang.Object) method for specifying the location of a single file or directory. Relative paths are resolved relative to the project directory, while absolute paths remain unchanged.

CAUTION	Never use `new File(relative path)` unless passed to `file()` or `files()` or `from()` or other methods being defined in terms of `file()` or `files()`. Otherwise this creates a path relative to the current working directory (CWD). Gradle can make no guarantees about the location of the CWD, which means builds that rely on it may break at any time.

Here are some examples of using the `file()` method with different types of argument:

Example 154. Locating files

build.gradle

```
// Using a relative path
File configFile = file('src/config.xml')

// Using an absolute path
configFile = file(configFile.absolutePath)

// Using a File object with a relative path
configFile = file(new File('src/config.xml'))

// Using a java.nio.file.Path object with a relative path
configFile = file(Paths.get('src', 'config.xml'))

// Using an absolute java.nio.file.Path object
configFile = file(Paths.get(System.getProperty('user.home')).resolve('global-
config.xml'))
```

build.gradle.kts

```
// Using a relative path
var configFile = file("src/config.xml")

// Using an absolute path
configFile = file(configFile.absolutePath)

// Using a File object with a relative path
configFile = file(File("src/config.xml"))

// Using a java.nio.file.Path object with a relative path
configFile = file(Paths.get("src", "config.xml"))

// Using an absolute java.nio.file.Path object
configFile = file(Paths.get(System.getProperty("user.home")).resolve("global-
config.xml"))
```

As you can see, you can pass strings, `File` instances and `Path` instances to the `file()` method, all of which result in an absolute `File` object. You can find other options for argument types in the reference guide, linked in the previous paragraph.

What happens in the case of multi-project builds? The `file()` method will always turn relative paths into paths that are relative to the current project directory, which may be a child project. If you want to use a path that's relative to the *root project* directory, then you need to use the special

Project.getRootDir() property to construct an absolute path, like so:

Example 155. Creating a path relative to a parent project

build.gradle

```
File configFile = file("$rootDir/shared/config.xml")
```

build.gradle.kts

```
val configFile = file("$rootDir/shared/config.xml")
```

Let's say you're working on a multi-project build in a `dev/projects/AcmeHealth` directory. You use the above example in the build of the library you're fixing — at `AcmeHealth/subprojects/AcmePatientRecordLib/build.gradle`. The file path will resolve to the absolute version of `dev/projects/AcmeHealth/shared/config.xml`.

The `file()` method can be used to configure any task that has a property of type `File`. Many tasks, though, work on multiple files, so we look at how to specify sets of files next.

File collections

A *file collection* is simply a set of file paths that's represented by the FileCollection interface. *Any* file paths. It's important to understand that the file paths don't have to be related in any way, so they don't have to be in the same directory or even have a shared parent directory. You will also find that many parts of the Gradle API use `FileCollection`, such as the copying API discussed later in this chapter and dependency configurations.

The recommended way to specify a collection of files is to use the ProjectLayout.files(java.lang.Object...) method, which returns a `FileCollection` instance. This method is very flexible and allows you to pass multiple strings, `File` instances, collections of strings, collections of `File`s, and more. You can even pass in tasks as arguments if they have defined outputs. Learn about all the supported argument types in the reference guide.

CAUTION `files()` properly handle relative paths and `File(relative path)` instances, resolving them relative to the project directory.

As with the Project.file(java.lang.Object) method covered in the previous section, all relative paths are evaluated relative to the current project directory. The following example demonstrates some of the variety of argument types you can use — strings, `File` instances, a list and a `Path`:

Example 156. Creating a file collection

build.gradle

```
FileCollection collection = layout.files('src/file1.txt',
                                         new File('src/file2.txt'),
                                         ['src/file3.csv', 'src/file4.csv'],
                                         Paths.get('src', 'file5.txt'))
```

build.gradle.kts

```
val collection: FileCollection = layout.files(
    "src/file1.txt",
    File("src/file2.txt"),
    listOf("src/file3.csv", "src/file4.csv"),
    Paths.get("src", "file5.txt")
)
```

File collections have some important attributes in Gradle. They can be:

- created lazily
- iterated over
- filtered
- combined

Lazy creation of a file collection is useful when you need to evaluate the files that make up a collection at the time a build runs. In the following example, we query the file system to find out what files exist in a particular directory and then make those into a file collection:

Example 157. Implementing a file collection

build.gradle

```
tasks.register('list') {
    doLast {
        File srcDir

        // Create a file collection using a closure
        collection = layout.files { srcDir.listFiles() }

        srcDir = file('src')
        println "Contents of $srcDir.name"
        collection.collect { relativePath(it) }.sort().each { println it }

        srcDir = file('src2')
        println "Contents of $srcDir.name"
        collection.collect { relativePath(it) }.sort().each { println it }
    }
}
```

build.gradle.kts

```
tasks.register("list") {
    doLast {
        var srcDir: File? = null

        val collection = layout.files({
            srcDir?.listFiles()
        })

        srcDir = file("src")
        println("Contents of ${srcDir.name}")
        collection.map { relativePath(it) }.sorted().forEach { println(it) }

        srcDir = file("src2")
        println("Contents of ${srcDir.name}")
        collection.map { relativePath(it) }.sorted().forEach { println(it) }
    }
}
```

Output of `gradle -q list`

```
> gradle -q list
Contents of src
src/dir1
src/file1.txt
Contents of src2
src2/dir1
src2/dir2
```

The key to lazy creation is passing a closure (in Groovy) or a `Provider` (in Kotlin) to the `files()` method. Your closure/provider simply needs to return a value of a type accepted by `files()`, such as `List<File>`, `String`, `FileCollection`, etc.

Iterating over a file collection can be done through the `each()` method (in Groovy) or `forEach` method (in Kotlin) on the collection or using the collection in a `for` loop. In both approaches, the file collection is treated as a set of `File` instances, i.e. your iteration variable will be of type `File`.

The following example demonstrates such iteration as well as how you can convert file collections to other types using the `as` operator or supported properties:

Example 158. Using a file collection

build.gradle

```
// Iterate over the files in the collection
collection.each { File file ->
    println file.name
}

// Convert the collection to various types
Set set = collection.files
Set set2 = collection as Set
List list = collection as List
String path = collection.asPath
File file = collection.singleFile

// Add and subtract collections
def union = collection + layout.files('src/file2.txt')
def difference = collection - layout.files('src/file2.txt')
```

build.gradle.kts

```
// Iterate over the files in the collection
collection.forEach { file: File ->
    println(file.name)
}

// Convert the collection to various types
val set: Set<File> = collection.files
val list: List<File> = collection.toList()
val path: String = collection.asPath
val file: File = collection.singleFile

// Add and subtract collections
val union = collection + layout.files("src/file2.txt")
val difference = collection - layout.files("src/file2.txt")
```

You can also see at the end of the example *how to combine file collections* using the `+` and `-` operators to merge and subtract them. An important feature of the resulting file collections is that they are *live*. In other words, when you combine file collections in this way, the result always reflects what's currently in the source file collections, even if they change during the build.

For example, imagine `collection` in the above example gains an extra file or two after `union` is created. As long as you use `union` after those files are added to `collection`, `union` will also contain those additional files. The same goes for the `different` file collection.

Live collections are also important when it comes to *filtering*. If you want to use a subset of a file collection, you can take advantage of the FileCollection.filter(org.gradle.api.specs.Spec) method to determine which files to "keep". In the following example, we create a new collection that consists of only the files that end with .txt in the source collection:

Example 159. Filtering a file collection

build.gradle

```
FileCollection textFiles = collection.filter { File f ->
    f.name.endsWith(".txt")
}
```

build.gradle.kts

```
val textFiles: FileCollection = collection.filter { f: File ->
    f.name.endsWith(".txt")
}
```

Output of `gradle -q filterTextFiles`

```
> gradle -q filterTextFiles
src/file1.txt
src/file2.txt
src/file5.txt
```

If `collection` changes at any time, either by adding or removing files from itself, then `textFiles` will immediately reflect the change because it is also a live collection. Note that the closure you pass to `filter()` takes a `File` as an argument and should return a boolean.

File trees

A *file tree* is a file collection that retains the directory structure of the files it contains and has the type FileTree. This means that all the paths in a file tree must have a shared parent directory. The following diagram highlights the distinction between file trees and file collections in the common case of copying files:

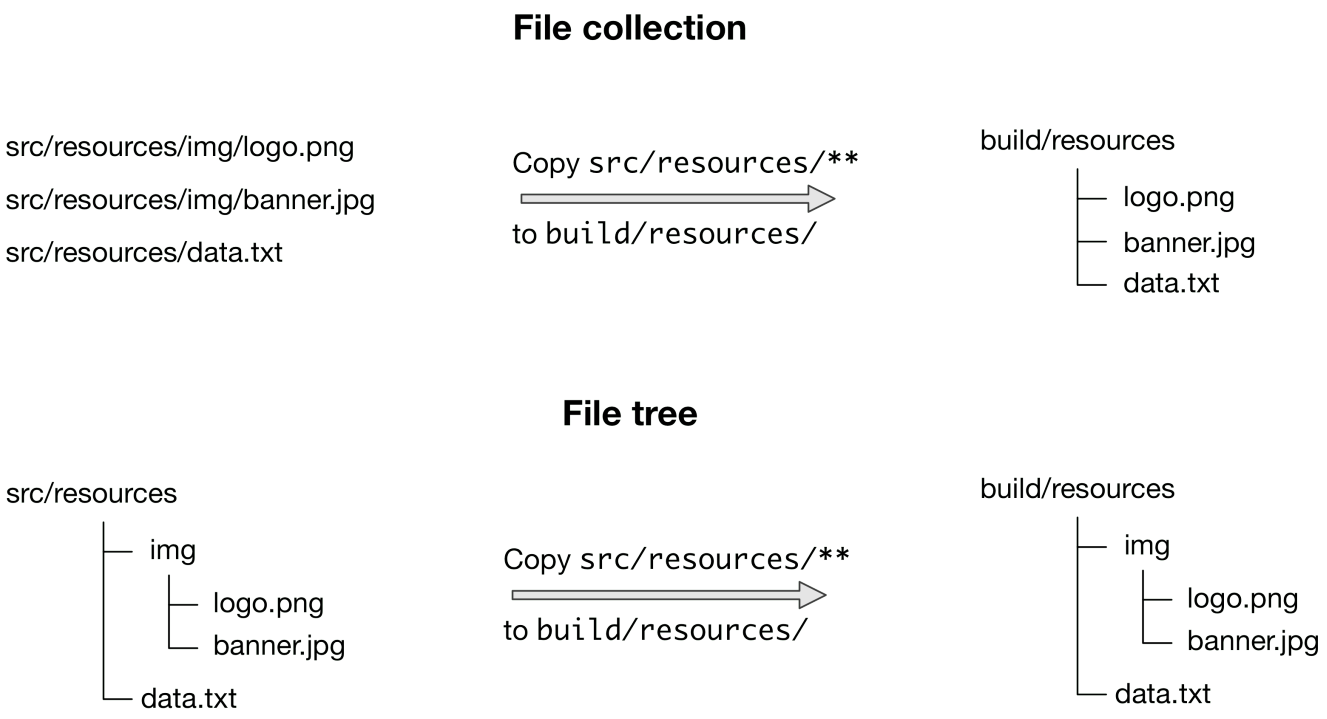


Figure 9. The differences in how file trees and file collections behave when copying files

NOTE	Although `FileTree` extends `FileCollection` (an is-a relationship), their behaviors do differ. In other words, you can use a file tree wherever a file collection is required, but remember: a file collection is a flat list/set of files, while a file tree is a file and directory hierarchy. To convert a file tree to a flat collection, use the FileTree.getFiles() property.

The simplest way to create a file tree is to pass a file or directory path to the Project.fileTree(java.lang.Object) method. This will create a tree of all the files and directories in that base directory (but not the base directory itself). The following example demonstrates how to use the basic method and, in addition, how to filter the files and directories using Ant-style patterns:

Example 160. Creating a file tree

build.gradle

```
// Create a file tree with a base directory
ConfigurableFileTree tree = fileTree(dir: 'src/main')

// Add include and exclude patterns to the tree
tree.include '**/*.java'
tree.exclude '**/Abstract*'

// Create a tree using closure
tree = fileTree('src') {
    include '**/*.java'
}

// Create a tree using a map
tree = fileTree(dir: 'src', include: '**/*.java')
tree = fileTree(dir: 'src', includes: ['**/*.java', '**/*.xml'])
tree = fileTree(dir: 'src', include: '**/*.java', exclude: '**/*test*/**')
```

build.gradle.kts

```
// Create a file tree with a base directory
var tree: ConfigurableFileTree = fileTree("src/main")

// Add include and exclude patterns to the tree
tree.include("**/*.java")
tree.exclude("**/Abstract*")

// Create a tree using closure
tree = fileTree("src") {
    include("**/*.java")
}

// Create a tree using a map
tree = fileTree("dir" to "src", "include" to "**/*.java")
tree = fileTree("dir" to "src", "includes" to listOf("**/*.java",
"**/*.xml"))
tree = fileTree("dir" to "src", "include" to "**/*.java", "exclude" to
"**/*test*/**")
```

You can see more examples of supported patterns in the API docs for PatternFilterable. Also, see the API documentation for `fileTree()` to see what types you can pass as the base directory.

By default, `fileTree()` returns a `FileTree` instance that applies some default exclude patterns for

convenience — the same defaults as Ant in fact. For the complete default exclude list, see the Ant manual.

If those default excludes prove problematic, you can workaround the issue by changing the default excludes in the settings script:

Example 161. Changing default excludes in the settings script

settings.gradle

```
import org.apache.tools.ant.DirectoryScanner

DirectoryScanner.removeDefaultExclude('**/.git')
DirectoryScanner.removeDefaultExclude('**/.git/**')
```

settings.gradle.kts

```
import org.apache.tools.ant.DirectoryScanner

DirectoryScanner.removeDefaultExclude("**/.git")
DirectoryScanner.removeDefaultExclude("**/.git/**")
```

NOTE Currently, Gradle's default excludes are configured via Ant's `DirectoryScanner` class.

NOTE Gradle does not support changing default excludes during the execution phase.

You can do many of the same things with file trees that you can with file collections:

- iterate over them (depth first)
- filter them (using FileTree.matching(org.gradle.api.Action) and Ant-style patterns)
- merge them

You can also traverse file trees using the FileTree.visit(org.gradle.api.Action) method. All of these techniques are demonstrated in the following example:

Example 162. Using a file tree

build.gradle

```
// Iterate over the contents of a tree
tree.each {File file ->
    println file
}

// Filter a tree
FileTree filtered = tree.matching {
    include 'org/gradle/api/**'
}

// Add trees together
FileTree sum = tree + fileTree(dir: 'src/test')

// Visit the elements of the tree
tree.visit {element ->
    println "$element.relativePath => $element.file"
}
```

build.gradle.kts

```
// Iterate over the contents of a tree
tree.forEach{ file: File ->
    println(file)
}

// Filter a tree
val filtered: FileTree = tree.matching {
    include("org/gradle/api/**")
}

// Add trees together
val sum: FileTree = tree + fileTree("src/test")

// Visit the elements of the tree
tree.visit {
    println("${this.relativePath} => ${this.file}")
}
```

We’ve discussed how to create your own file trees and file collections, but it’s also worth bearing in mind that many Gradle plugins provide their own instances of file trees, such as Java’s source sets. These can be used and manipulated in exactly the same way as the file trees you create yourself.

Another specific type of file tree that users commonly need is the archive, i.e. ZIP files, TAR files, etc. We look at those next.

Using archives as file trees

An archive is a directory and file hierarchy packed into a single file. In other words, it's a special case of a file tree, and that's exactly how Gradle treats archives. Instead of using the `fileTree()` method, which only works on normal file systems, you use the Project.zipTree(java.lang.Object) and Project.tarTree(java.lang.Object) methods to wrap archive files of the corresponding type (note that JAR, WAR and EAR files are ZIPs). Both methods return `FileTree` instances that you can then use in the same way as normal file trees. For example, you can extract some or all of the files of an archive by copying its contents to some directory on the file system. Or you can merge one archive into another.

Here are some simple examples of creating archive-based file trees:

Example 163. Using an archive as a file tree

build.gradle

```
// Create a ZIP file tree using path
FileTree zip = zipTree('someFile.zip')

// Create a TAR file tree using path
FileTree tar = tarTree('someFile.tar')

//tar tree attempts to guess the compression based on the file extension
//however if you must specify the compression explicitly you can:
FileTree someTar = tarTree(resources.gzip('someTar.ext'))
```

build.gradle.kts

```
// Create a ZIP file tree using path
val zip: FileTree = zipTree("someFile.zip")

// Create a TAR file tree using path
val tar: FileTree = tarTree("someFile.tar")

// tar tree attempts to guess the compression based on the file extension
// however if you must specify the compression explicitly you can:
val someTar: FileTree = tarTree(resources.gzip("someTar.ext"))
```

You can see a practical example of extracting an archive file in among the common scenarios we cover.

Understanding implicit conversion to file collections

Many objects in Gradle have properties which accept a set of input files. For example, the JavaCompile task has a `source` property that defines the source files to compile. You can set the value of this property using any of the types supported by the files() method, as mentioned in the API docs. This means you can, for example, set the property to a `File`, `String`, collection, `FileCollection` or even a closure or `Provider`.

This is a feature of specific tasks! That means implicit conversion will not happen for just any task that has a `FileCollection` or `FileTree` property. If you want to know whether implicit conversion happens in a particular situation, you will need to read the relevant documentation, such as the corresponding task's API docs. Alternatively, you can remove all doubt by explicitly using ProjectLayout.files(java.lang.Object...) in your build.

Here are some examples of the different types of arguments that the `source` property can take:

Example 164. Specifying a set of files

build.gradle

```
tasks.register('compile', JavaCompile) {

    // Use a File object to specify the source directory
    source = file('src/main/java')

    // Use a String path to specify the source directory
    source = 'src/main/java'

    // Use a collection to specify multiple source directories
    source = ['src/main/java', '../shared/java']

    // Use a FileCollection (or FileTree in this case) to specify the source
files
    source = fileTree(dir: 'src/main/java').matching { include
'org/gradle/api/**' }

    // Using a closure to specify the source files.
    source = {
        // Use the contents of each zip file in the src dir
        file('src').listFiles().findAll {it.name.endsWith('.zip')}.collect {
zipTree(it) }
    }
}
```

build.gradle.kts

```
tasks.register<JavaCompile>("compile") {
    // Use a File object to specify the source directory
    source = fileTree(file("src/main/java"))

    // Use a String path to specify the source directory
    source = fileTree("src/main/java")

    // Use a collection to specify multiple source directories
    source = fileTree(listOf("src/main/java", "../shared/java"))

    // Use a FileCollection (or FileTree in this case) to specify the source
files
    source = fileTree("src/main/java").matching {
include("org/gradle/api/**") }

    // Using a closure to specify the source files.
    setSource({
        // Use the contents of each zip file in the src dir
        file("src").listFiles().filter { it.name.endsWith(".zip") }.map {
zipTree(it) }
    })
}
```

One other thing to note is that properties like `source` have corresponding methods in core Gradle tasks. Those methods follow the convention of *appending* to collections of values rather than replacing them. Again, this method accepts any of the types supported by the files() method, as shown here:

Example 165. Appending a set of files

build.gradle

```
compile {
    // Add some source directories use String paths
    source 'src/main/java', 'src/main/groovy'

    // Add a source directory using a File object
    source file('../shared/java')

    // Add some source directories using a closure
    source { file('src/test/').listFiles() }
}
```

build.gradle.kts

```
tasks.named<JavaCompile>("compile") {
    // Add some source directories use String paths
    source("src/main/java", "src/main/groovy")

    // Add a source directory using a File object
    source(file("../shared/java"))

    // Add some source directories using a closure
    setSource({ file("src/test/").listFiles() })
}
```

As this is a common convention, we recommend that you follow it in your own custom tasks. Specifically, if you plan to add a method to configure a collection-based property, make sure the method appends rather than replaces values.

File copying in depth

The basic process of copying files in Gradle is a simple one:

- Define a task of type Copy
- Specify which files (and potentially directories) to copy
- Specify a destination for the copied files

But this apparent simplicity hides a rich API that allows fine-grained control of which files are copied, where they go, and what happens to them as they are copied — renaming of the files and token substitution of file content are both possibilities, for example.

Let's start with the last two items on the list, which form what is known as a *copy specification*. This is formally based on the CopySpec interface, which the `Copy` task implements, and offers:

- A CopySpec.from(java.lang.Object...) method to define what to copy
- An CopySpec.into(java.lang.Object) method to define the destination

`CopySpec` has several additional methods that allow you to control the copying process, but these two are the only required ones. `into()` is straightforward, requiring a directory path as its argument in any form supported by the Project.file(java.lang.Object) method. The `from()` configuration is far more flexible.

Not only does `from()` accept multiple arguments, it also allows several different types of argument. For example, some of the most common types are:

- A `String` — treated as a file path or, if it starts with "file://", a file URI
- A `File` — used as a file path
- A `FileCollection` or `FileTree` — all files in the collection are included in the copy
- A task — the files or directories that form a task's defined outputs are included

In fact, `from()` accepts all the same arguments as Project.files(java.lang.Object...) so see that method for a more detailed list of acceptable types.

Something else to consider is what type of thing a file path refers to:

- A file — the file is copied as is
- A directory — this is effectively treated as a file tree: everything in it, including subdirectories, is copied. However, the directory itself is not included in the copy.
- A non-existent file — the path is ignored

Here is an example that uses multiple `from()` specifications, each with a different argument type. You will probably also notice that `into()` is configured lazily using a closure (in Groovy) or a Provider (in Kotlin) — a technique that also works with `from()`:

Example 166. Specifying copy task source files and destination directory

build.gradle

```
tasks.register('anotherCopyTask', Copy) {
    // Copy everything under src/main/webapp
    from 'src/main/webapp'
    // Copy a single file
    from 'src/staging/index.html'
    // Copy the output of a task
    from copyTask
    // Copy the output of a task using Task outputs explicitly.
    from copyTaskWithPatterns.outputs
    // Copy the contents of a Zip file
    from zipTree('src/main/assets.zip')
    // Determine the destination directory later
    into { getDestDir() }
}
```

build.gradle.kts

```
tasks.register<Copy>("anotherCopyTask") {
    // Copy everything under src/main/webapp
    from("src/main/webapp")
    // Copy a single file
    from("src/staging/index.html")
    // Copy the output of a task
    from(copyTask)
    // Copy the output of a task using Task outputs explicitly.
    from(tasks["copyTaskWithPatterns"].outputs)
    // Copy the contents of a Zip file
    from(zipTree("src/main/assets.zip"))
    // Determine the destination directory later
    into({ getDestDir() })
}
```

Note that the lazy configuration of `into()` is different from a child specification, even though the syntax is similar. Keep an eye on the number of arguments to distinguish between them.

Filtering files

You've already seen that you can filter file collections and file trees directly in a `Copy` task, but you can also apply filtering in any copy specification through the CopySpec.include(java.lang.String...) and CopySpec.exclude(java.lang.String...) methods.

Both of these methods are normally used with Ant-style include or exclude patterns, as described in

PatternFilterable. You can also perform more complex logic by using a closure that takes a FileTreeElement and returns `true` if the file should be included or `false` otherwise. The following example demonstrates both forms, ensuring that only .html and .jsp files are copied, except for those .html files with the word "DRAFT" in their content:

Example 167. Selecting the files to copy

build.gradle

```
tasks.register('copyTaskWithPatterns', Copy) {
    from 'src/main/webapp'
    into layout.buildDirectory.dir('explodedWar')
    include '**/*.html'
    include '**/*.jsp'
    exclude { FileTreeElement details ->
        details.file.name.endsWith('.html') &&
            details.file.text.contains('DRAFT')
    }
}
```

build.gradle.kts

```
tasks.register<Copy>("copyTaskWithPatterns") {
    from("src/main/webapp")
    into(layout.buildDirectory.dir("explodedWar"))
    include("**/*.html")
    include("**/*.jsp")
    exclude { details: FileTreeElement ->
        details.file.name.endsWith(".html") &&
            details.file.readText().contains("DRAFT")
    }
}
```

A question you may ask yourself at this point is what happens when inclusion and exclusion patterns overlap? Which pattern wins? Here are the basic rules:

- If there are no explicit inclusions or exclusions, everything is included
- If at least one inclusion is specified, only files and directories matching the patterns are included
- Any exclusion pattern overrides any inclusions, so if a file or directory matches at least one exclusion pattern, it won't be included, regardless of the inclusion patterns

Bear these rules in mind when creating combined inclusion and exclusion specifications so that you end up with the exact behavior you want.

Note that the inclusions and exclusions in the above example will apply to *all* `from()` configurations. If you want to apply filtering to a subset of the copied files, you'll need to use child specifications.

Renaming files

The example of how to rename files on copy gives you most of the information you need to perform this operation. It demonstrates the two options for renaming:

- Using a regular expression
- Using a closure

Regular expressions are a flexible approach to renaming, particularly as Gradle supports regex groups that allow you to remove and replaces parts of the source filename. The following example shows how you can remove the string "-staging-" from any filename that contains it using a simple regular expression:

Example 168. Renaming files as they are copied

build.gradle

```
tasks.register('rename', Copy) {
    from 'src/main/webapp'
    into layout.buildDirectory.dir('explodedWar')
    // Use a closure to convert all file names to upper case
    rename { String fileName ->
        fileName.toUpperCase()
    }
    // Use a regular expression to map the file name
    rename '(.+)-staging-(.+)', '$1$2'
    rename(/(.+)-staging-(.+)/, '$1$2')
}
```

build.gradle.kts

```
tasks.register<Copy>("rename") {
    from("src/main/webapp")
    into(layout.buildDirectory.dir("explodedWar"))
    // Use a closure to convert all file names to upper case
    rename { fileName: String ->
        fileName.toUpperCase()
    }
    // Use a regular expression to map the file name
    rename("(.+)-staging-(.+)", "$1$2")
    rename("(.+)-staging-(.+)".toRegex().pattern, "$1$2")
}
```

You can use any regular expression supported by the Java `Pattern` class and the substitution string (the second argument of `rename()` works on the same principles as the `Matcher.appendReplacement()` method.

NOTE

Regular expressions in Groovy build scripts

There are two common issues people come across when using regular expressions in this context:

1. If you use a slashy string (those delimited by '/') for the first argument, you *must* include the parentheses for `rename()` as shown in the above example.
2. It's safest to use single quotes for the second argument, otherwise you need to escape the '$' in group substitutions, i.e. `"\$1\$2"`

The first is a minor inconvenience, but slashy strings have the advantage that you don't have to escape backslash ('\') characters in the regular expression. The second issue stems from Groovy's support for embedded expressions using `${ }` syntax in double-quoted and slashy strings.

The closure syntax for `rename()` is straightforward and can be used for any requirements that simple regular expressions can't handle. You're given the name of a file and you return a new name for that file, or `null` if you don't want to change the name. Do be aware that the closure will be executed for every file that's copied, so try to avoid expensive operations where possible.

Filtering file content (token substitution, templating, etc.)

Not to be confused with filtering which files are copied, *file content filtering* allows you to transform the content of files while they are being copied. This can involve basic templating that uses token substitution, removal of lines of text, or even more complex filtering using a full-blown template engine.

The following example demonstrates several forms of filtering, including token substitution using the CopySpec.expand(java.util.Map) method and another using CopySpec.filter(java.lang.Class) with an Ant filter:

Example 169. Filtering files as they are copied

build.gradle

```
import org.apache.tools.ant.filters.FixCrLfFilter
import org.apache.tools.ant.filters.ReplaceTokens

tasks.register('filter', Copy) {
    from 'src/main/webapp'
    into layout.buildDirectory.dir('explodedWar')
    // Substitute property tokens in files
    expand(copyright: '2009', version: '2.3.1')
    expand(project.properties)
    // Use some of the filters provided by Ant
    filter(FixCrLfFilter)
    filter(ReplaceTokens, tokens: [copyright: '2009', version: '2.3.1'])
    // Use a closure to filter each line
    filter { String line ->
        "[$line]"
    }
    // Use a closure to remove lines
    filter { String line ->
        line.startsWith('-') ? null : line
    }
    filteringCharset = 'UTF-8'
}
```

build.gradle.kts

```
import org.apache.tools.ant.filters.FixCrLfFilter
import org.apache.tools.ant.filters.ReplaceTokens
tasks.register<Copy>("filter") {
    from("src/main/webapp")
    into(layout.buildDirectory.dir("explodedWar"))
    // Substitute property tokens in files
    expand("copyright" to "2009", "version" to "2.3.1")
    expand(project.properties)
    // Use some of the filters provided by Ant
    filter(FixCrLfFilter::class)
    filter(ReplaceTokens::class, "tokens" to mapOf("copyright" to "2009",
"version" to "2.3.1"))
    // Use a closure to filter each line
    filter { line: String ->
        "[$line]"
    }
    // Use a closure to remove lines
    filter { line: String ->
        if (line.startsWith('-')) null else line
    }
    filteringCharset = "UTF-8"
}
```

The `filter()` method has two variants, which behave differently:

- one takes a `FilterReader` and is designed to work with Ant filters, such as `ReplaceTokens`
- one takes a closure or Transformer that defines the transformation for each line of the source file

Note that both variants assume the source files are text based. When you use the `ReplaceTokens` class with `filter()`, the result is a template engine that replaces tokens of the form `@tokenName@` (the Ant-style token) with values that you define.

The `expand()` method treats the source files as Groovy templates, which evaluate and expand expressions of the form `${expression}`. You can pass in property names and values that are then expanded in the source files. `expand()` allows for more than basic token substitution as the embedded expressions are full-blown Groovy expressions.

NOTE It's good practice to specify the character set when reading and writing the file, otherwise the transformations won't work properly for non-ASCII text. You configure the character set with the CopySpec.getFilteringCharset() property. If it's not specified, the JVM default character set is used, which is likely to be different from the one you want.

Using the `CopySpec` class

A copy specification (or copy spec for short) determines what gets copied to where, and what happens to files during the copy. You've alread seen many examples in the form of configuration for `Copy` and archiving tasks. But copy specs have two attributes that are worth covering in more detail:

1. They can be independent of tasks
2. They are hierarchical

The first of these attributes allows you to *share copy specs within a build*. The second provides fine-grained control within the overall copy specification.

Sharing copy specs

Consider a build that has several tasks that copy a project's static website resources or add them to an archive. One task might copy the resources to a folder for a local HTTP server and another might package them into a distribution. You could manually specify the file locations and appropriate inclusions each time they are needed, but human error is more likely to creep in, resulting in inconsistencies between tasks.

One solution Gradle provides is the Project.copySpec(org.gradle.api.Action) method. This allows you to create a copy spec outside of a task, which can then be attached to an appropriate task using the CopySpec.with(org.gradle.api.file.CopySpec...) method. The following example demonstrates how this is done:

Example 170. Sharing copy specifications

build.gradle

```
CopySpec webAssetsSpec = copySpec {
    from 'src/main/webapp'
    include '**/*.html', '**/*.png', '**/*.jpg'
    rename '(.+)-staging(.+)', '$1$2'
}

tasks.register('copyAssets', Copy) {
    into layout.buildDirectory.dir("inPlaceApp")
    with webAssetsSpec
}

tasks.register('distApp', Zip) {
    archiveFileName = 'my-app-dist.zip'
    destinationDirectory = layout.buildDirectory.dir('dists')

    from appClasses
    with webAssetsSpec
}
```

build.gradle.kts

```
val webAssetsSpec: CopySpec = copySpec {
    from("src/main/webapp")
    include("**/*.html", "**/*.png", "**/*.jpg")
    rename("(.+)-staging(.+)", "$1$2")
}

tasks.register<Copy>("copyAssets") {
    into(layout.buildDirectory.dir("inPlaceApp"))
    with(webAssetsSpec)
}

tasks.register<Zip>("distApp") {
    archiveFileName.set("my-app-dist.zip")
    destinationDirectory.set(layout.buildDirectory.dir("dists"))

    from(appClasses)
    with(webAssetsSpec)
}
```

Both the `copyAssets` and `distApp` tasks will process the static resources under `src/main/webapp`, as

specified by `webAssetsSpec`.

NOTE

The configuration defined by `webAssetsSpec` will *not* apply to the app classes included by the `distApp` task. That's because `from appClasses` is its own child specification independent of `with webAssetsSpec`.

This can be confusing to understand, so it's probably best to treat `with()` as an extra `from()` specification in the task. Hence it doesn't make sense to define a standalone copy spec without at least one `from()` defined.

If you encounter a scenario in which you want to apply the same copy configuration to *different* sets of files, then you can share the configuration block directly without using `copySpec()`. Here's an example that has two independent tasks that happen to want to process image files only:

Example 171. Sharing copy patterns only

build.gradle

```
def webAssetPatterns = {
    include '**/*.html', '**/*.png', '**/*.jpg'
}

tasks.register('copyAppAssets', Copy) {
    into layout.buildDirectory.dir("inPlaceApp")
    from 'src/main/webapp', webAssetPatterns
}

tasks.register('archiveDistAssets', Zip) {
    archiveFileName = 'distribution-assets.zip'
    destinationDirectory = layout.buildDirectory.dir('dists')

    from 'distResources', webAssetPatterns
}
```

build.gradle.kts

```
val webAssetPatterns = Action<CopySpec> {
    include("**/*.html", "**/*.png", "**/*.jpg")
}

tasks.register<Copy>("copyAppAssets") {
    into(layout.buildDirectory.dir("inPlaceApp"))
    from("src/main/webapp", webAssetPatterns)
}

tasks.register<Zip>("archiveDistAssets") {
    archiveFileName.set("distribution-assets.zip")
    destinationDirectory.set(layout.buildDirectory.dir("dists"))

    from("distResources", webAssetPatterns)
}
```

In this case, we assign the copy configuration to its own variable and apply it to whatever `from()` specification we want. This doesn't just work for inclusions, but also exclusions, file renaming, and file content filtering.

Using child specifications

If you only use a single copy spec, the file filtering and renaming will apply to *all* the files that are copied. Sometimes this is what you want, but not always. Consider the following example that

copies files into a directory structure that can be used by a Java Servlet container to deliver a website:

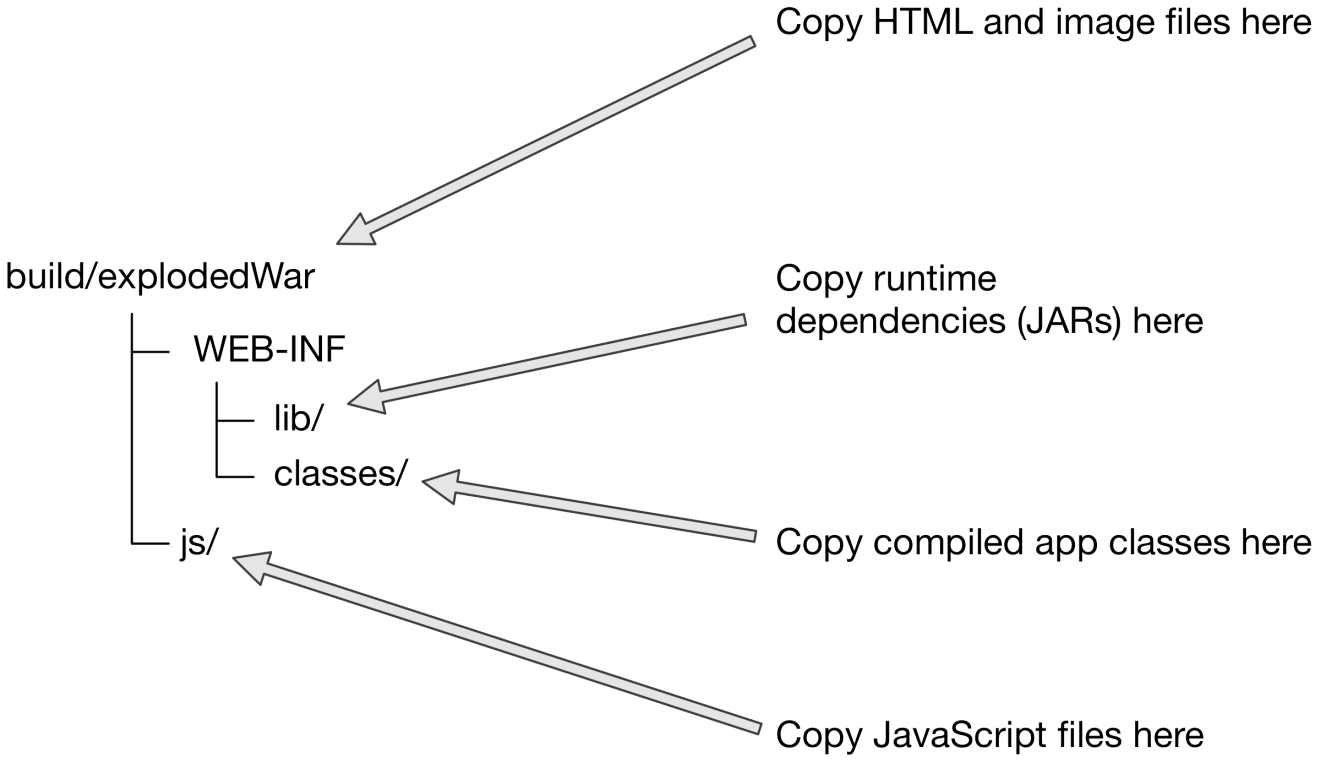


Figure 10. Creating an exploded WAR for a Servlet container

This is not a straightforward copy as the `WEB-INF` directory and its subdirectories don't exist within the project, so they must be created during the copy. In addition, we only want HTML and image files going directly into the root folder — `build/explodedWar` — and only JavaScript files going into the `js` directory. So we need separate filter patterns for those two sets of files.

The solution is to use *child specifications*, which can be applied to both `from()` and `into()` declarations. The following task definition does the necessary work:

Example 172. Nested copy specs

build.gradle

```
tasks.register('nestedSpecs', Copy) {
    into layout.buildDirectory.dir("explodedWar")
    exclude '**/*staging*'
    from('src/dist') {
        include '**/*.html', '**/*.png', '**/*.jpg'
    }
    from(sourceSets.main.output) {
        into 'WEB-INF/classes'
    }
    into('WEB-INF/lib') {
        from configurations.runtimeClasspath
    }
}
```

build.gradle.kts

```
tasks.register<Copy>("nestedSpecs") {
    into(layout.buildDirectory.dir("explodedWar"))
    exclude("**/*staging*")
    from("src/dist") {
        include("**/*.html", "**/*.png", "**/*.jpg")
    }
    from(sourceSets.main.get().output) {
        into("WEB-INF/classes")
    }
    into("WEB-INF/lib") {
        from(configurations.runtimeClasspath)
    }
}
```

Notice how the `src/dist` configuration has a nested inclusion specification: that's the child copy spec. You can of course add content filtering and renaming here as required. A child copy spec is still a copy spec.

The above example also demonstrates how you can copy files into a subdirectory of the destination either by using a child `into()` on a `from()` or a child `from()` on an `into()`. Both approaches are acceptable, but you may want to create and follow a convention to ensure consistency across your build files.

NOTE	Don't get your `into()` specifications mixed up! For a normal copy — one to the filesystem rather than an archive — there should always be *one* "root" `into()` that simply specifies the overall destination directory of the copy. Any other `into()` should have a child spec attached and its path will be relative to the root `into()`.

One final thing to be aware of is that a child copy spec inherits its destination path, include patterns, exclude patterns, copy actions, name mappings and filters from its parent. So be careful where you place your configuration.

Copying files in your own tasks

There might be occasions when you want to copy files or directories as *part* of a task. For example, a custom archiving task based on an unsupported archive format might want to copy files to a temporary directory before they are then archived. You still want to take advantage of Gradle's copy API, but without introducing an extra `Copy` task.

The solution is to use the Project.copy(org.gradle.api.Action) method. It works the same way as the `Copy` task by configuring it with a copy spec. Here's a trivial example:

Example 173. Copying files using the copy() method without up-to-date check

build.gradle

```
tasks.register('copyMethod') {
    doLast {
        copy {
            from 'src/main/webapp'
            into layout.buildDirectory.dir('explodedWar')
            include '**/*.html'
            include '**/*.jsp'
        }
    }
}
```

build.gradle.kts

```
tasks.register("copyMethod") {
    doLast {
        copy {
            from("src/main/webapp")
            into(layout.buildDirectory.dir("explodedWar"))
            include("**/*.html")
            include("**/*.jsp")
        }
    }
}
```

The above example demonstrates the basic syntax and also highlights two major limitations of using the `copy()` method:

1. The `copy()` method is not incremental. The example's `copyMethod` task will *always* execute because it has no information about what files make up the task's inputs. You have to manually define the task inputs and outputs.
2. Using a task as a copy source, i.e. as an argument to `from()`, won't set up an automatic task dependency between your task and that copy source. As such, if you are using the `copy()` method as part of a task action, you must explicitly declare all inputs and outputs in order to get the correct behavior.

The following example shows you how to workaround these limitations by using the dynamic API for task inputs and outputs:

Example 174. Copying files using the copy() method with up-to-date check

build.gradle

```
tasks.register('copyMethodWithExplicitDependencies') {
    // up-to-date check for inputs, plus add copyTask as dependency
    inputs.files(copyTask)
        .withPropertyName("inputs")
        .withPathSensitivity(PathSensitivity.RELATIVE)
    outputs.dir('some-dir') // up-to-date check for outputs
        .withPropertyName("outputDir")
    doLast{
        copy {
            // Copy the output of copyTask
            from copyTask
            into 'some-dir'
        }
    }
}
```

build.gradle.kts

```
tasks.register("copyMethodWithExplicitDependencies") {
    // up-to-date check for inputs, plus add copyTask as dependency
    inputs.files(copyTask)
        .withPropertyName("inputs")
        .withPathSensitivity(PathSensitivity.RELATIVE)
    outputs.dir("some-dir") // up-to-date check for outputs
        .withPropertyName("outputDir")
    doLast {
        copy {
            // Copy the output of copyTask
            from(copyTask)
            into("some-dir")
        }
    }
}
```

These limitations make it preferable to use the `Copy` task wherever possible, because of its builtin support for incremental building and task dependency inference. That is why the `copy()` method is intended for use by custom tasks that need to copy files as part of their function. Custom tasks that use the `copy()` method should declare the necessary inputs and outputs relevant to the copy action.

Mirroring directories and file collections with the `Sync` task

The Sync task, which extends the `Copy` task, copies the source files into the destination directory and then removes any files from the destination directory which it did not copy. In other words, it synchronizes the contents of a directory with its source. This can be useful for doing things such as installing your application, creating an exploded copy of your archives, or maintaining a copy of the project's dependencies.

Here is an example which maintains a copy of the project's runtime dependencies in the `build/libs` directory.

Example 175. Using the Sync task to copy dependencies

build.gradle

```
tasks.register('libs', Sync) {
    from configurations.runtime
    into layout.buildDirectory.dir('libs')
}
```

build.gradle.kts

```
tasks.register<Sync>("libs") {
    from(configurations["runtime"])
    into(layout.buildDirectory.dir("libs"))
}
```

You can also perform the same function in your own tasks with the Project.sync(org.gradle.api.Action) method.

Deploying single files into application servers

When working with application servers, you can use a `Copy` task to deploy the application archive (e.g. a WAR file). Since you are deploying a single file, the destination directory of the `Copy` is the whole deployment directory. The deployment directory sometimes does contain unreadable files like named pipes, so Gradle may have problems doing up-to-date checks. In order to support this use-case, you can use Task.doNotTrackState().

Example 176. Using Copy to deploy a WAR file

build.gradle

```
plugins {
    id 'war'
}

tasks.register("deployToTomcat", Copy) {
    from war
    into layout.projectDirectory.dir('tomcat/webapps')
    doNotTrackState("Deployment directory contains unreadable files")
}
```

build.gradle.kts

```
plugins {
    war
}

tasks.register<Copy>("deployToTomcat") {
    from(tasks.war)
    into(layout.projectDirectory.dir("tomcat/webapps"))
    doNotTrackState("Deployment directory contains unreadable files")
}
```

Installing executables

When you are building a standalone executable, you may want to install this file on your system, so it ends up in your path. You can use a `Copy` task to install the executable into shared directories like `/usr/local/bin`. The installation directory probably contains many other executables, some of which may even be unreadable by Gradle. To support the unreadable files in the `Copy` task's destination directory and to avoid time consuming up-to-date checks, you can use Task.doNotTrackState().

Example 177. Using Copy to install an executable

build.gradle

```
tasks.register("installExecutable", Copy) {
    from "build/my-binary"
    into "/usr/local/bin"
    doNotTrackState("Installation directory contains unrelated files")
}
```

build.gradle.kts

```
tasks.register<Copy>("installExecutable") {
    from("build/my-binary")
    into("/usr/local/bin")
    doNotTrackState("Installation directory contains unrelated files")
}
```

Archive creation in depth

Archives are essentially self-contained file systems and Gradle treats them as such. This is why working with archives is very similar to working with files and directories, including such things as file permissions.

Out of the box, Gradle supports creation of both ZIP and TAR archives, and by extension Java's JAR, WAR and EAR formats — Java's archive formats are all ZIPs. Each of these formats has a corresponding task type to create them: Zip, Tar, Jar, War, and Ear. These all work the same way and are based on copy specifications, just like the `Copy` task.

Creating an archive file is essentially a file copy in which the destination is implicit, i.e. the archive file itself. Here's a basic example that specifies the path and name of the target archive file:

Example 178. Archiving a directory as a ZIP

build.gradle

```
tasks.register('packageDistribution', Zip) {
    archiveFileName = "my-distribution.zip"
    destinationDirectory = layout.buildDirectory.dir('dist')

    from layout.buildDirectory.dir("toArchive")
}
```

build.gradle.kts

```
tasks.register<Zip>("packageDistribution") {
    archiveFileName.set("my-distribution.zip")
    destinationDirectory.set(layout.buildDirectory.dir("dist"))

    from(layout.buildDirectory.dir("toArchive"))
}
```

In the next section you'll learn about convention-based archive names, which can save you from always configuring the destination directory and archive name.

The full power of copy specifications are available to you when creating archives, which means you can do content filtering, file renaming or anything else that is covered in the previous section. A particularly common requirement is copying files into subdirectories of the archive that don't exist in the source folders, something that can be achieved with `into()` child specifications.

Gradle does of course allow you create as many archive tasks as you want, but it's worth bearing in mind that many convention-based plugins provide their own. For example, the Java plugin adds a `jar` task for packaging a project's compiled classes and resources in a JAR. Many of these plugins provide sensible conventions for the names of archives as well as the copy specifications used. We recommend you use these tasks wherever you can, rather than overriding them with your own.

Archive naming

Gradle has several conventions around the naming of archives and where they are created based on the plugins your project uses. The main convention is provided by the Base Plugin, which defaults to creating archives in the `$buildDir/distributions` directory and typically uses archive names of the form *[projectName]-[version].[type]*.

The following example comes from a project named `archive-naming`, hence the `myZip` task creates an archive named `archive-naming-1.0.zip`:

Example 179. Creation of ZIP archive

build.gradle

```
plugins {
    id 'base'
}

version = 1.0

tasks.register('myZip', Zip) {
    from 'somedir'

    doLast {
        println archiveFileName.get()
        println relativePath(destinationDirectory)
        println relativePath(archiveFile)
    }
}
```

build.gradle.kts

```
plugins {
    base
}

version = "1.0"

tasks.register<Zip>("myZip") {
    from("somedir")

    doLast {
        println(archiveFileName.get())
        println(relativePath(destinationDirectory))
        println(relativePath(archiveFile))
    }
}
```

Output of `gradle -q myZip`

```
> gradle -q myZip
archive-naming-1.0.zip
build/distributions
build/distributions/archive-naming-1.0.zip
```

Note that the name of the archive does *not* derive from the name of the task that creates it.

If you want to change the name and location of a generated archive file, you can provide values for the `archiveFileName` and `destinationDirectory` properties of the corresponding task. These override any conventions that would otherwise apply.

Alternatively, you can make use of the default archive name pattern provided by AbstractArchiveTask.getArchiveFileName(): *[archiveBaseName]-[archiveAppendix]-[archiveVersion]-[archiveClassifier].[archiveExtension]*. You can set each of these properties on the task separately if you wish. Note that the Base Plugin uses the convention of project name for *archiveBaseName*, project version for *archiveVersion* and the archive type for *archiveExtension*. It does not provide values for the other properties.

This example — from the same project as the one above — configures just the `archiveBaseName` property, overriding the default value of the project name:

Example 180. Configuration of archive task - custom archive name

build.gradle

```
tasks.register('myCustomZip', Zip) {
    archiveBaseName = 'customName'
    from 'somedir'

    doLast {
        println archiveFileName.get()
    }
}
```

build.gradle.kts

```
tasks.register<Zip>("myCustomZip") {
    archiveBaseName.set("customName")
    from("somedir")

    doLast {
        println(archiveFileName.get())
    }
}
```

Output of `gradle -q myCustomZip`

```
> gradle -q myCustomZip
customName-1.0.zip
```

You can also override the default `archiveBaseName` value for *all* the archive tasks in your build by using the *project* property `archivesBaseName`, as demonstrated by the following example:

Example 181. Configuration of archive task - appendix & classifier

build.gradle

```
plugins {
    id 'base'
}

version = 1.0
// tag::base-plugin-config[]
base {
    archivesName = "gradle"
    distsDirectory = layout.buildDirectory.dir('custom-dist')
    libsDirectory = layout.buildDirectory.dir('custom-libs')
}
// end::base-plugin-config[]

tasks.register('myZip', Zip) {
    from 'somedir'
}

tasks.register('myOtherZip', Zip) {
    archiveAppendix = 'wrapper'
    archiveClassifier = 'src'
    from 'somedir'
}

tasks.register('echoNames') {
    doLast {
        println "Project name: ${project.name}"
        println myZip.archiveFileName.get()
        println myOtherZip.archiveFileName.get()
    }
}
```

build.gradle.kts

```
plugins {
    base
}

version = "1.0"

// tag::base-plugin-config[]
base {
    archivesName.set("gradle")
    distsDirectory.set(layout.buildDirectory.dir("custom-dist"))
    libsDirectory.set(layout.buildDirectory.dir("custom-libs"))
}
// end::base-plugin-config[]

val myZip by tasks.registering(Zip::class) {
    from("somedir")
}

val myOtherZip by tasks.registering(Zip::class) {
    archiveAppendix.set("wrapper")
    archiveClassifier.set("src")
    from("somedir")
}

tasks.register("echoNames") {
    doLast {
        println("Project name: ${project.name}")
        println(myZip.get().archiveFileName.get())
        println(myOtherZip.get().archiveFileName.get())
    }
}
```

Output of `gradle -q echoNames`

```
> gradle -q echoNames
Project name: archives-changed-base-name
gradle-1.0.zip
gradle-wrapper-1.0-src.zip
```

You can find all the possible archive task properties in the API documentation for AbstractArchiveTask, but we have also summarized the main ones here:

`archiveFileName` — `Property<String>`, **default:** `archiveBaseName-archiveAppendix-archiveVersion-archiveClassifier.archiveExtension`

The complete file name of the generated archive. If any of the properties in the default value are

empty, their '-' separator is dropped.

`archiveFile` — `Provider<RegularFile>`, ***read-only***, **default:** `destinationDirectory/archiveFileName`

The absolute file path of the generated archive.

`destinationDirectory` — `DirectoryProperty`, **default: depends on archive type**

The target directory in which to put the generated archive. By default, JARs and WARs go into `$buildDir/libs`. ZIPs and TARs go into `$buildDir/distributions`.

`archiveBaseName` — `Property<String>`, **default:** `project.name`

The base name portion of the archive file name, typically a project name or some other descriptive name for what it contains.

`archiveAppendix` — `Property<String>`, **default:** `null`

The appendix portion of the archive file name that comes immediately after the base name. It is typically used to distinguish between different forms of content, such as code and docs, or a minimal distribution versus a full or complete one.

`archiveVersion` — `Property<String>`, **default:** `project.version`

The version portion of the archive file name, typically in the form of a normal project or product version.

`archiveClassifier` — `Property<String>`, **default:** `null`

The classifier portion of the archive file name. Often used to distinguish between archives that target different platforms.

`archiveExtension` — `Property<String>`, **default: depends on archive type and compression type**

The filename extension for the archive. By default, this is set based on the archive task type and the compression type (if you're creating a TAR). Will be one of: `zip`, `jar`, `war`, `tar`, `tgz` or `tbz2`. You can of course set this to a custom extension if you wish.

Sharing content between multiple archives

As described earlier, you can use the Project.copySpec(org.gradle.api.Action) method to share content between archives.

Reproducible builds

Sometimes it's desirable to recreate archives exactly the same, byte for byte, on different machines. You want to be sure that building an artifact from source code produces the same result no matter when and where it is built. This is necessary for projects like reproducible-builds.org.

Reproducing the same byte-for-byte archive poses some challenges since the order of the files in an archive is influenced by the underlying file system. Each time a ZIP, TAR, JAR, WAR or EAR is built from source, the order of the files inside the archive may change. Files that only have a different timestamp also causes differences in archives from build to build. All AbstractArchiveTask (e.g. Jar, Zip) tasks shipped with Gradle include support for producing reproducible archives.

For example, to make a `Zip` task reproducible you need to set Zip.isReproducibleFileOrder() to `true`

and Zip.isPreserveFileTimestamps() to `false`. In order to make all archive tasks in your build reproducible, consider adding the following configuration to your build file:

Example 182. Activating reproducible archives

build.gradle

```
tasks.withType(AbstractArchiveTask).configureEach {
    preserveFileTimestamps = false
    reproducibleFileOrder = true
}
```

build.gradle.kts

```
tasks.withType<AbstractArchiveTask>().configureEach {
    isPreserveFileTimestamps = false
    isReproducibleFileOrder = true
}
```

Often you will want to publish an archive, so that it is usable from another project. This process is described in Cross-Project publications.

Using Gradle Plugins

Gradle at its core intentionally provides very little for real world automation. All of the useful features, like the ability to compile Java code, are added by *plugins*. Plugins add new tasks (e.g. JavaCompile), domain objects (e.g. SourceSet), conventions (e.g. Java source is located at `src/main/java`) as well as extending core objects and objects from other plugins.

In this chapter we discuss how to use plugins and the terminology and concepts surrounding plugins.

What plugins do

Applying a plugin to a project allows the plugin to extend the project's capabilities. It can do things such as:

- Extend the Gradle model (e.g. add new DSL elements that can be configured)
- Configure the project according to conventions (e.g. add new tasks or configure sensible defaults)
- Apply specific configuration (e.g. add organizational repositories or enforce standards)

By applying plugins, rather than adding logic to the project build script, we can reap a number of benefits. Applying plugins:

- Promotes reuse and reduces the overhead of maintaining similar logic across multiple projects
- Allows a higher degree of modularization, enhancing comprehensibility and organization
- Encapsulates imperative logic and allows build scripts to be as declarative as possible

Types of plugins

There are two general types of plugins in Gradle, *binary* plugins and *script* plugins. Binary plugins are written either programmatically by implementing Plugin interface or declaratively using one of Gradle's DSL languages. Binary plugins can reside within a build script, within the project hierarchy or externally in a plugin jar. Script plugins are additional build scripts that further configure the build and usually implement a declarative approach to manipulating the build. They are typically used within a build although they can be externalized and accessed from a remote location.

A plugin often starts out as a script plugin (because they are easy to write) and then, as the code becomes more valuable, it's migrated to a binary plugin that can be easily tested and shared between multiple projects or organizations.

Using plugins

To use the build logic encapsulated in a plugin, Gradle needs to perform two steps. First, it needs to *resolve* the plugin, and then it needs to *apply* the plugin to the target, usually a Project.

Resolving a plugin means finding the correct version of the jar which contains a given plugin and adding it to the script classpath. Once a plugin is resolved, its API can be used in a build script. Script plugins are self-resolving in that they are resolved from the specific file path or URL provided when applying them. Core binary plugins provided as part of the Gradle distribution are automatically resolved.

Applying a plugin means actually executing the plugin's Plugin.apply(T) on the Project you want to enhance with the plugin. Applying plugins is *idempotent*. That is, you can safely apply any plugin multiple times without side effects.

The most common use case for using a plugin is to both resolve the plugin and apply it to the current project. Since this is such a common use case, it's recommended that build authors use the plugins DSL to both resolve and apply plugins in one step.

Binary plugins

You apply plugins by their *plugin id*, which is a globally unique identifier, or name, for plugins. Core Gradle plugins are special in that they provide short names, such as `'java'` for the core JavaPlugin. All other binary plugins must use the fully qualified form of the plugin id (e.g. `com.github.foo.bar`), although some legacy plugins may still utilize a short, unqualified form. Where you put the plugin id depends on whether you are using the plugins DSL or the buildscript block.

Locations of binary plugins

A plugin is simply any class that implements the Plugin interface. Gradle provides the core plugins

(e.g. `JavaPlugin`) as part of its distribution which means they are automatically resolved. However, non-core binary plugins need to be resolved before they can be applied. This can be achieved in a number of ways:

- Including the plugin from the plugin portal or a custom repository using the plugins DSL (see Applying plugins using the plugins DSL).
- Including the plugin from an external jar defined as a buildscript dependency (see Applying plugins using the buildscript block).
- Defining the plugin as a source file under the buildSrc directory in the project (see Using buildSrc to extract functional logic).
- Defining the plugin as an inline class declaration inside a build script.

For more on defining your own plugins, see Custom Plugins.

Applying plugins with the plugins DSL

The plugins DSL provides a succinct and convenient way to declare plugin dependencies. It works with the Gradle plugin portal to provide easy access to both core and community plugins. The plugins DSL block configures an instance of PluginDependenciesSpec.

To apply a core plugin, the short name can be used:

Example 183. Applying a core plugin

build.gradle

```
plugins {
    id 'java'
}
```

build.gradle.kts

```
plugins {
    java
}
```

To apply a community plugin from the portal, the fully qualified plugin id must be used:

Example 184. Applying a community plugin

build.gradle

```
plugins {
    id 'org.springframework.boot' version '2.4.1'
}
```

build.gradle.kts

```
plugins {
    id("org.springframework.boot") version "2.4.1"
}
```

See PluginDependenciesSpec for more information on using the Plugin DSL.

Limitations of the plugins DSL

This way of adding plugins to a project is much more than a more convenient syntax. The plugins DSL is processed in a way which allows Gradle to determine the plugins in use very early and very quickly. This allows Gradle to do smart things such as:

- Optimize the loading and reuse of plugin classes.
- Provide editors detailed information about the potential properties and values in the buildscript for editing assistance.

This requires that plugins be specified in a way that Gradle can easily and quickly extract, before executing the rest of the build script. It also requires that the definition of plugins to use be somewhat static.

There are some key differences between the `plugins {}` block mechanism and the "traditional" `apply()` method mechanism. There are also some constraints, some of which are temporary limitations while the mechanism is still being developed and some are inherent to the new approach.

Constrained Syntax

The `plugins {}` block does not support arbitrary code. It is constrained, in order to be idempotent (produce the same result every time) and side effect free (safe for Gradle to execute at any time).

The form is:

build.gradle

```
plugins {
    id «plugin id»                                          ①
    id «plugin id» version «plugin version» [apply «false»]  ②
}
```

① for core Gradle plugins or plugins already available to the build script

② for binary Gradle plugins that need to be resolved

build.gradle.kts

```
plugins {
    `«plugin id»`                                           ①
    id(«plugin id»)                                         ②
    id(«plugin id») version «plugin version» [apply «false»]  ③
}
```

① for core Gradle plugins

② for core Gradle plugins or plugins already available to the build script

③ for binary Gradle plugins that need to be resolved

Where `«plugin id»` and `«plugin version»` must be constant, literal, strings and the `apply` statement with a `boolean` can be used to disable the default behavior of applying the plugin immediately (e.g. you want to apply it only in `subprojects`). No other statements are allowed; their presence will cause a compilation error.

Where `«plugin id»`, in case `#1` is a static Kotlin extension property, named after the core plugin ID ; and in cases `#2` and `#3` is a string. `«plugin version»` is also a string. The `apply` statement with a `boolean` can be used to disable the default behavior of applying the plugin immediately (e.g. you want to apply it only in `subprojects`).

See plugin version management if you want to use a variable to define a plugin version.

The `plugins {}` block must also be a top level statement in the buildscript. It cannot be nested inside another construct (e.g. an if-statement or for-loop).

Can only be used in build scripts and settings file

The `plugins {}` block can currently only be used in a project's build script and the settings.gradle file. It cannot be used in script plugins or init scripts.

Future versions of Gradle will remove this restriction.

If the restrictions of the `plugins {}` block are prohibitive, the recommended approach is to apply

plugins using the buildscript {} block.

Applying external plugins with same version to subprojects

If you have a multi-project build, you probably want to apply plugins to some or all of the subprojects in your build, but not to the `root` project. The default behavior of the `plugins {}` block is to immediately `resolve` *and* `apply` the plugins. But, you can use the `apply false` syntax to tell Gradle not to apply the plugin to the current project and then use the `plugins {}` block without the version in subprojects' build scripts:

Example 185. Applying plugins only on certain subprojects

settings.gradle

```
include 'hello-a'
include 'hello-b'
include 'goodbye-c'
```

build.gradle

```
plugins {
    id 'com.example.hello' version '1.0.0' apply false
    id 'com.example.goodbye' version '1.0.0' apply false
}
```

hello-a/build.gradle

```
plugins {
    id 'com.example.hello'
}
```

hello-b/build.gradle

```
plugins {
    id 'com.example.hello'
}
```

goodbye-c/build.gradle

```
plugins {
    id 'com.example.goodbye'
}
```

settings.gradle.kts

```
include("hello-a")
include("hello-b")
include("goodbye-c")
```

build.gradle.kts

```
plugins {
    id("com.example.hello") version "1.0.0" apply false
    id("com.example.goodbye") version "1.0.0" apply false
}
```

hello-a/build.gradle.kts

```
plugins {
    id("com.example.hello")
}
```

hello-b/build.gradle.kts

```
plugins {
    id("com.example.hello")
}
```

goodbye-c/build.gradle.kts

```
plugins {
    id("com.example.goodbye")
}
```

Even better - you can encapsulate the versions of external plugins by composing the build logic using your own convention plugins.

Applying plugins from the *buildSrc* directory

You can apply plugins that reside in a project's *buildSrc* directory as long as they have a defined ID. The following example shows how to tie a plugin implementation class — `my.MyPlugin` — defined in *buildSrc* to the ID "my-plugin":

Example 186. Defining a buildSrc plugin with an ID

buildSrc/build.gradle

```
plugins {
    id 'java-gradle-plugin'
}

gradlePlugin {
    plugins {
        myPlugins {
            id = 'my-plugin'
            implementationClass = 'my.MyPlugin'
        }
    }
}
```

buildSrc/build.gradle.kts

```
plugins {
    `java-gradle-plugin`
}

gradlePlugin {
    plugins {
        create("myPlugins") {
            id = "my-plugin"
            implementationClass = "my.MyPlugin"
        }
    }
}
```

The plugin can then be applied by ID as normal:

Example 187. Applying a plugin from buildSrc

build.gradle

```
plugins {
    id 'my-plugin'
}
```

build.gradle.kts

```
plugins {
    id("my-plugin")
}
```

Plugin Management

The `pluginManagement {}` block may only appear in either the `settings.gradle` file, where it must be the first block in the file, or in an Initialization Script.

Example 188. Configuring pluginManagement per-project and globally

settings.gradle

```
pluginManagement {
    plugins {
    }
    resolutionStrategy {
    }
    repositories {
    }
}
rootProject.name = 'plugin-management'
```

init.gradle

```
settingsEvaluated { settings ->
    settings.pluginManagement {
        plugins {
        }
        resolutionStrategy {
        }
        repositories {
        }
    }
}
```

settings.gradle.kts

```
pluginManagement {
    plugins {
    }
    resolutionStrategy {
    }
    repositories {
    }
}
rootProject.name = "plugin-management"
```

init.gradle.kts

```
settingsEvaluated {
    pluginManagement {
        plugins {
        }
        resolutionStrategy {
        }
        repositories {
        }
    }
}
```

Custom Plugin Repositories

By default, the `plugins {}` DSL resolves plugins from the public Gradle Plugin Portal. Many build authors would also like to resolve plugins from private Maven or Ivy repositories because the plugins contain proprietary implementation details, or just to have more control over what plugins are available to their builds.

To specify custom plugin repositories, use the `repositories {}` block inside `pluginManagement {}`:

Example 189. Example: Using plugins from custom plugin repositories.

settings.gradle

```
pluginManagement {
    repositories {
        maven {
            url './maven-repo'
        }
        gradlePluginPortal()
        ivy {
            url './ivy-repo'
        }
    }
}
```

settings.gradle.kts

```
pluginManagement {
    repositories {
        maven(url = "./maven-repo")
        gradlePluginPortal()
        ivy(url = "./ivy-repo")
    }
}
```

This tells Gradle to first look in the Maven repository at `../maven-repo` when resolving plugins and then to check the Gradle Plugin Portal if the plugins are not found in the Maven repository. If you don't want the Gradle Plugin Portal to be searched, omit the `gradlePluginPortal()` line. Finally, the Ivy repository at `../ivy-repo` will be checked.

Plugin Version Management

A `plugins {}` block inside `pluginManagement {}` allows all plugin versions for the build to be defined in a single location. Plugins can then be applied by id to any build script via the `plugins {}` block.

One benefit of setting plugin versions this way is that the `pluginManagement.plugins {}` does not have the same constrained syntax as the build script `plugins {}` block. This allows plugin versions to be taken from `gradle.properties`, or loaded via another mechanism.

Example 190. Example: Managing plugin versions via `pluginManagement`.

settings.gradle

```
pluginManagement {
  plugins {
        id 'com.example.hello' version "${helloPluginVersion}"
    }
}
```

build.gradle

```
plugins {
    id 'com.example.hello'
}
```

gradle.properties

```
helloPluginVersion=1.0.0
```

settings.gradle.kts

```
pluginManagement {
  val helloPluginVersion: String by settings
  plugins {
    id("com.example.hello") version "${helloPluginVersion}"
  }
}
```

build.gradle.kts

```
plugins {
    id("com.example.hello")
}
```

gradle.properties

```
helloPluginVersion=1.0.0
```

The plugin version is loaded from `gradle.properties` and configured in the settings script, allowing the plugin to be added to any project without specifying the version.

Plugin Resolution Rules

Plugin resolution rules allow you to modify plugin requests made in `plugins {}` blocks, e.g. changing the requested version or explicitly specifying the implementation artifact coordinates.

To add resolution rules, use the `resolutionStrategy {}` inside the `pluginManagement {}` block:

Example 191. Plugin resolution strategy.

settings.gradle

```
pluginManagement {
    resolutionStrategy {
        eachPlugin {
            if (requested.id.namespace == 'com.example') {
                useModule('com.example:sample-plugins:1.0.0')
            }
        }
    }
    repositories {
        maven {
            url './maven-repo'
        }
        gradlePluginPortal()
        ivy {
            url './ivy-repo'
        }
    }
}
```

settings.gradle.kts

```
pluginManagement {
    resolutionStrategy {
        eachPlugin {
            if (requested.id.namespace == "com.example") {
                useModule("com.example:sample-plugins:1.0.0")
            }
        }
    }
    repositories {
        maven {
            url = uri("./maven-repo")
        }
        gradlePluginPortal()
        ivy {
            url = uri("./ivy-repo")
        }
    }
}
```

This tells Gradle to use the specified plugin implementation artifact instead of using its built-in

default mapping from plugin ID to Maven/Ivy coordinates.

Custom Maven and Ivy plugin repositories must contain plugin marker artifacts in addition to the artifacts which actually implement the plugin. For more information on publishing plugins to custom repositories read Gradle Plugin Development Plugin.

See PluginManagementSpec for complete documentation for using the `pluginManagement {}` block.

Plugin Marker Artifacts

Since the `plugins {}` DSL block only allows for declaring plugins by their globally unique plugin `id` and `version` properties, Gradle needs a way to look up the coordinates of the plugin implementation artifact. To do so, Gradle will look for a Plugin Marker Artifact with the coordinates `plugin.id:plugin.id.gradle.plugin:plugin.version`. This marker needs to have a dependency on the actual plugin implementation. Publishing these markers is automated by the java-gradle-plugin.

For example, the following complete sample from the `sample-plugins` project shows how to publish a `com.example.hello` plugin and a `com.example.goodbye` plugin to both an Ivy and Maven repository using the combination of the java-gradle-plugin, the maven-publish plugin, and the ivy-publish plugin.

Example 192. Complete Plugin Publishing Sample

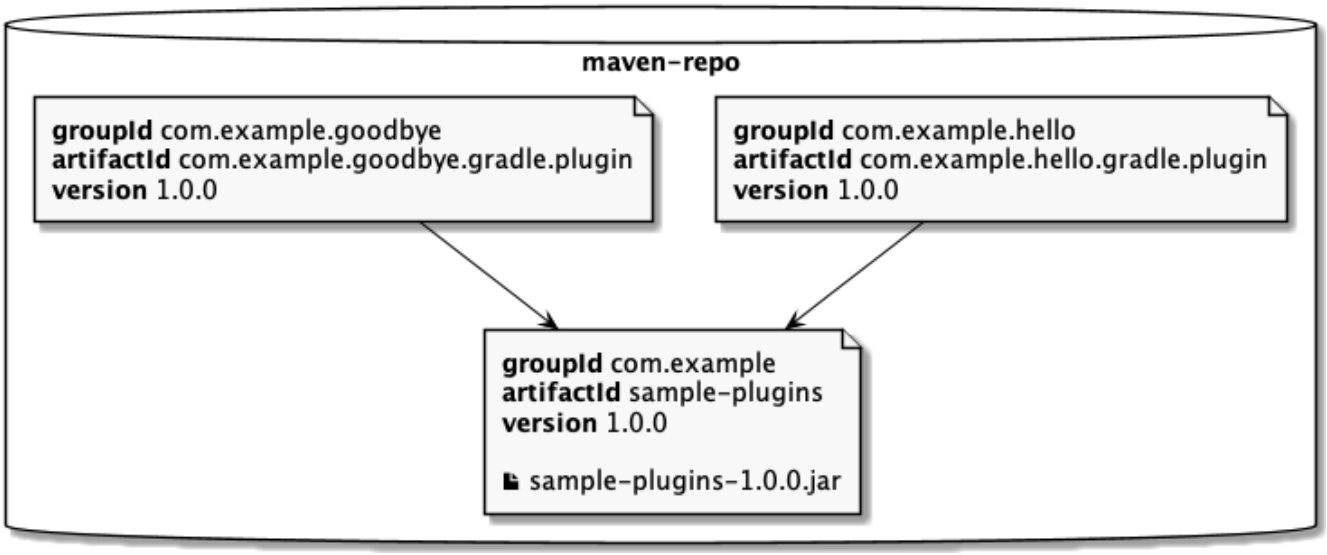
build.gradle

```
plugins {
    id 'java-gradle-plugin'
    id 'maven-publish'
    id 'ivy-publish'
}

group 'com.example'
version '1.0.0'

gradlePlugin {
    plugins {
        hello {
            id = 'com.example.hello'
            implementationClass = 'com.example.hello.HelloPlugin'
        }
        goodbye {
            id = 'com.example.goodbye'
            implementationClass = 'com.example.goodbye.GoodbyePlugin'
        }
    }
}

publishing {
    repositories {
        maven {
            url '../../consuming/maven-repo'
        }
        ivy {
            url '../../consuming/ivy-repo'
        }
    }
}
```

build.gradle.kts

```
plugins {
    `java-gradle-plugin`
    `maven-publish`
    `ivy-publish`
}

group = "com.example"
version = "1.0.0"

gradlePlugin {
    plugins {
        create("hello") {
            id = "com.example.hello"
            implementationClass = "com.example.hello.HelloPlugin"
        }
        create("goodbye") {
            id = "com.example.goodbye"
            implementationClass = "com.example.goodbye.GoodbyePlugin"
        }
    }
}

publishing {
    repositories {
        maven {
            url = uri("../../consuming/maven-repo")
        }
        ivy {
            url = uri("../../consuming/ivy-repo")
        }
    }
}
```

Running `gradle publish` in the sample directory creates the following Maven repository layout (the Ivy layout is similar):



Legacy Plugin Application

With the introduction of the plugins DSL, users should have little reason to use the legacy method of applying plugins. It is documented here in case a build author cannot use the plugins DSL due to restrictions in how it currently works.

Applying Binary Plugins

Example 193. Applying a binary plugin

build.gradle

```
apply plugin: 'java'
```

build.gradle.kts

```
apply(plugin = "java")
```

Plugins can be applied using a *plugin id*. In the above case, we are using the short name '`java`' to apply the JavaPlugin.

Rather than using a plugin id, plugins can also be applied by simply specifying the class of the plugin:

Example 194. Applying a binary plugin by type

build.gradle

```
apply plugin: JavaPlugin
```

build.gradle.kts

```
apply<JavaPlugin>()
```

The `JavaPlugin` symbol in the above sample refers to the JavaPlugin. This class does not strictly need to be imported as the `org.gradle.api.plugins` package is automatically imported in all build scripts (see Default imports).

Furthermore, it is not necessary to append `.class` to identify a class literal in Groovy as it is in Java.

Furthermore, one need to append the `::class` suffix to identify a class literal in Kotlin instead of `.class` in Java.

Applying plugins with the buildscript block

Binary plugins that have been published as external jar files can be added to a project by adding the plugin to the build script classpath and then applying the plugin. External jars can be added to the build script classpath using the `buildscript {}` block as described in External dependencies for the build script.

Example 195. Applying a plugin with the buildscript block

build.gradle

```
buildscript {
    repositories {
        gradlePluginPortal()
    }
    dependencies {
        classpath 'org.springframework.boot:spring-boot-gradle-plugin:2.4.1'
    }
}

apply plugin: 'org.springframework.boot'
```

build.gradle.kts

```
buildscript {
    repositories {
        gradlePluginPortal()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:2.4.1")
    }
}

apply(plugin = "org.springframework.boot")
```

Script plugins

Example 196. Applying a script plugin

build.gradle

```
apply from: 'other.gradle'
```

build.gradle.kts

```
apply(from = "other.gradle.kts")
```

Script plugins are automatically resolved and can be applied from a script on the local filesystem or at a remote location. Filesystem locations are relative to the project directory, while remote script locations are specified with an HTTP URL. Multiple script plugins (of either form) can be applied to a given target.

Finding community plugins

Gradle has a vibrant community of plugin developers who contribute plugins for a wide variety of capabilities. The Gradle plugin portal provides an interface for searching and exploring community plugins.

More on plugins

This chapter aims to serve as an introduction to plugins and Gradle and the role they play. For more information on the inner workings of plugins, see Custom Plugins.

Build Lifecycle

Gradle is an example of **dependency based programming**: you define tasks and dependencies between tasks. Gradle guarantees that these tasks execute in the order of their dependencies. Your build scripts and plugins configure this dependency graph. This page discusses the phases of the lifecycle Gradle passes through as it interprets those scripts. Additionally, this page explains how you can react to events of the lifecycle using notifications.

Task Graphs

Some build tools assemble a task graph as they execute tasks. Gradle builds the task graph before executing any task. With configuration avoidance, Gradle skips configuration for tasks that aren't part of the current build.

Within each project, tasks form a Directed Acyclic Graph (DAG).

This diagram shows two example task graphs: one abstract and the other concrete. The diagram represents dependencies between tasks as arrows:

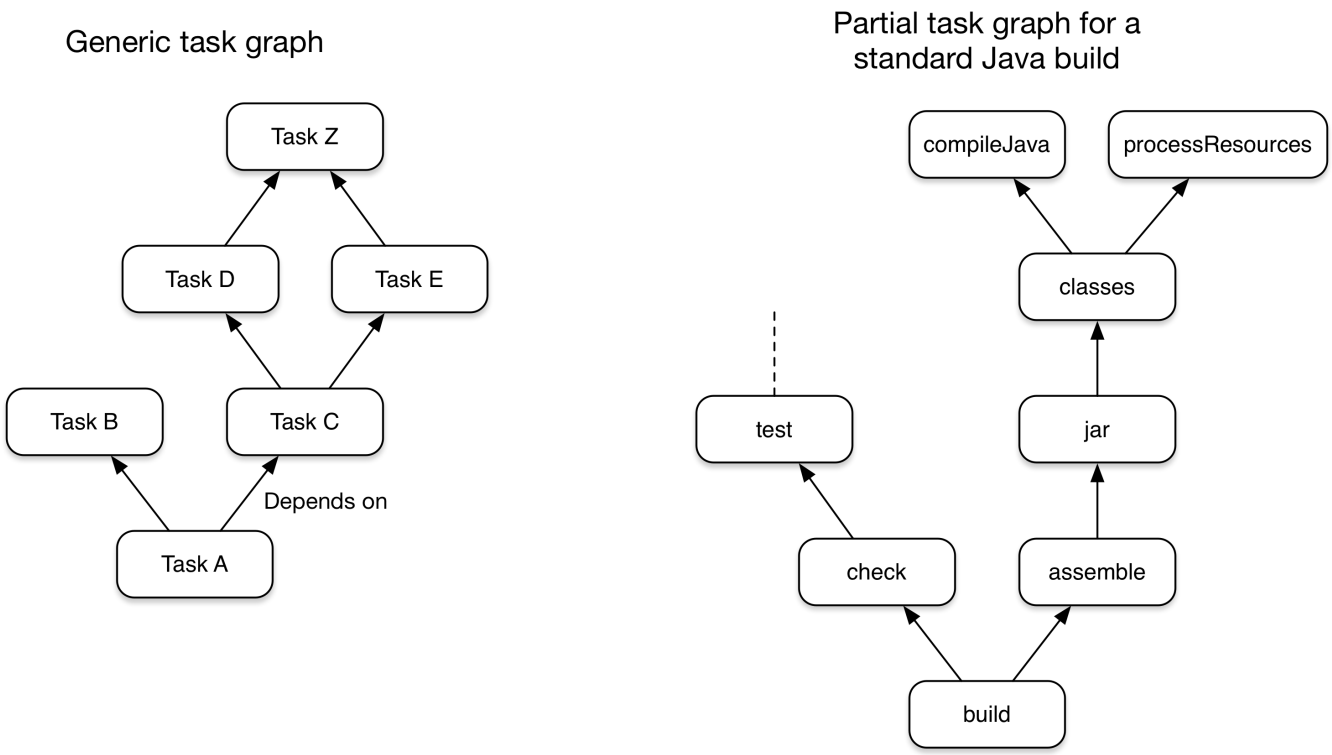


Figure 11. Two examples of Gradle task graphs

Both plugins and your own build scripts contribute to the task graph via the task dependency mechanism.

Build Phases

A Gradle build has three distinct phases. Gradle runs these phases in order: first initialization, then configuration, and finally execution.

Initialization

- Detects the settings file.
- Evaluates the settings file to determine which projects and included builds participate in the build.
- Creates a `Project` instance for every project.

Configuration

- Evaluates the build scripts of every project participating in the build.
- Creates a task graph for requested tasks.

Execution

- Schedules and executes each of the selected tasks in the order of their dependencies.

Example

The following example shows which parts of settings and build files correspond to various build phases:

settings.gradle

```
rootProject.name = 'basic'
println 'This is executed during the initialization phase.'
```

build.gradle

```
println 'This is executed during the configuration phase.'

tasks.register('configured') {
    println 'This is also executed during the configuration phase, because
:configured is used in the build.'
}

tasks.register('test') {
    doLast {
        println 'This is executed during the execution phase.'
    }
}

tasks.register('testBoth') {
    doFirst {
      println 'This is executed first during the execution phase.'
    }
    doLast {
      println 'This is executed last during the execution phase.'
    }
    println 'This is executed during the configuration phase as well, because
:testBoth is used in the build.'
}
```

settings.gradle.kts

```
rootProject.name = "basic"
println("This is executed during the initialization phase.")
```

build.gradle.kts

```
println("This is executed during the configuration phase.")

tasks.register("configured") {
    println("This is also executed during the configuration phase, because
:configured is used in the build.")
}

tasks.register("test") {
    doLast {
        println("This is executed during the execution phase.")
    }
}

tasks.register("testBoth") {
    doFirst {
        println("This is executed first during the execution phase.")
    }
    doLast {
        println("This is executed last during the execution phase.")
    }
    println("This is executed during the configuration phase as well, because
:testBoth is used in the build.")
}
```

The following command configures and executes the `test` and `testBoth` tasks specified above. Because Gradle only configures requested tasks and their dependencies, the `configured` task never configures.

```
> gradle test testBoth
This is executed during the initialization phase.

> Configure project :
This is executed during the configuration phase.
This is executed during the configuration phase as well, because :testBoth is used in
the build.

> Task :test
This is executed during the execution phase.

> Task :testBoth
This is executed first during the execution phase.
This is executed last during the execution phase.

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

```
> gradle test testBoth
This is executed during the initialization phase.

> Configure project :
This is executed during the configuration phase.
This is executed during the configuration phase as well, because :testBoth is used in
the build.

> Task :test
This is executed during the execution phase.

> Task :testBoth
This is executed first during the execution phase.
This is executed last during the execution phase.

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Initialization

In the initialization phase, Gradle detects the set of projects and included builds participating in the build. Gradle first evaluates the settings file. Then, Gradle instantiates `Project` instances for each project.

Detect Settings File

When you run Gradle in a directory that contains a `settings.gradle` file, Gradle uses that `settings.gradle` file to initialize the build. You can run Gradle within any subproject of the build.[5] When you run Gradle in a directory that contains **no** `settings.gradle` file:

1. Gradle looks for a `settings.gradle(.kts)` file in parent directories.
2. If Gradle finds a `settings.gradle(.kts)` file, Gradle checks if the current project is part of the multi-project build. If so, Gradle builds as a multi-project.
3. If Gradle does not find a `settings.gradle(.kts)` file, Gradle builds as a single project.

In the standard Gradle project layout, project paths match the physical subproject layout on disk. The automatic search for a settings file only works for multi-project builds with a standard project layout. To build projects that use a nonstandard layout, execute the build from the directory that contains `settings.gradle(.kts)`.

Evaluate Settings File

During settings file evaluation, Gradle:

- Adds libraries to your build script classpath.
- Defines which included builds participate in a composite build.
- Defines which projects participate in a multi-project build.

Gradle creates a `Project` for every project in the build. By default, each `Project` has a name equal to the name of its top level directory. Every project except the root project has a parent project. Any project may have child projects.

Configuration

In the configuration phase, Gradle adds tasks and other properties to the projects generated by the initialization phase. By the end of the configuration phase, Gradle has a complete task execution graph for the requested tasks.

Project Evaluation

During project evaluation, Gradle evaluates build scripts to build a task hierarchy within a `Project`. This hierarchy includes inputs, actions, and outputs for all tasks.

React to Project Evaluation

You can receive a notification immediately before and immediately after a project evaluates. These notifications work even when project evaluation fails. You can configure project evaluation notifications for all projects or a specific project. For example, you could use these notifications for:

- additional configuration after applying all definitions in a build script
- custom logging
- custom profiling

The following example uses `gradle.beforeProject()` to add `hasTests` property to certain tests. Later, the example uses `gradle.afterProject()` to add a `test` task to each project where the `hasTests` property value is `true`:

build.gradle

```
gradle.beforeProject { project ->
    project.ext.set("hasTests", false)
}

gradle.afterProject { project ->
    if (project.ext.has("hasTests") && project.ext.get("hasTests") as
Boolean) {
        def projectString = project.toString()
        println "Adding test task to $projectString"
        project.task('test') {
            doLast {
                println "Running tests for $projectString"
            }
        }
    }
}
```

project-a.gradle

```
hasTests = true
```

build.gradle.kts

```
gradle.beforeProject {
    // Set a default value
    project.ext.set("hasTests", false)
}

gradle.afterProject {
    if (project.ext.has("hasTests") && project.ext.get("hasTests") as
Boolean) {
        val projectString = project.toString()
        println("Adding test task to $projectString")
        tasks.register("test") {
            doLast {
                println("Running tests for $projectString")
            }
        }
    }
}
```

project-a.gradle.kts

```
extra["hasTests"] = true
```

```
> gradle -q test
Adding test task to project ':project-a'
Running tests for project ':project-a'
```

To receive these events via a listener instead of a closure, add a ProjectEvaluationListener to a build's Gradle instance.

Task Execution Graph Assembly

During project evaluation, Gradle assembles the **task execution graph**: a DAG representing the dependency relationships between tasks.

React to Task Execution Graph Assembly

You can receive a notification immediately after Gradle finishes populating a project's task execution graph.

To receive these events, add a TaskExecutionGraphListener to a project's TaskExecutionGraph.

Task Creation

During project evaluation, Gradle registers tasks and their configuration actions. The configuration

actions define inputs, outputs, and actions for those tasks. They are evaluated if the task is part of the task graph for the requested tasks.

React to Task Creation

You can receive a notification immediately after Gradle adds a task to a project. For example, you could use these notifications to set some default values or methods.

The following example sets a `srcDir` value on each task in a project:

build.gradle

```
tasks.whenTaskAdded { task ->
    task.ext.srcDir = 'src/main/java'
}

tasks.register('a')

println "source dir is $a.srcDir"
```

build.gradle.kts

```
tasks.whenTaskAdded {
    extra["srcDir"] = "src/main/java"
}

val a by tasks.registering

println("source dir is ${a.get().extra["srcDir"]}")
```

```
> gradle -q a
source dir is src/main/java
```

To receive these events via a listener instead of a closure, add an Action to a TaskContainer.

Execution

In the execution phase, Gradle runs tasks. Gradle uses the task execution graphs generated by the configuration phase to determine which tasks to execute.

Task Execution

Task execution includes most of the work you normally associate with a build: downloading libraries, compiling code, reading inputs, and writing outputs.

React to Task Execution Notifications

You can receive a notification immediately before and after Gradle executes any task. These notifications work even when task execution fails. The following example logs the start and end of each task execution:

build.gradle

```
tasks.register('ok')

tasks.register('broken') {
    dependsOn ok
    doLast {
        throw new RuntimeException('broken')
    }
}

gradle.taskGraph.beforeTask { Task task ->
    println "executing $task ..."
}

gradle.taskGraph.afterTask { Task task, TaskState state ->
    if (state.failure) {
        println "FAILED"
    }
    else {
        println "done"
    }
}
```

build.gradle.kts

```
tasks.register("ok")

tasks.register("broken") {
    dependsOn("ok")
    doLast {
        throw RuntimeException("broken")
    }
}

gradle.taskGraph.beforeTask {
    println("executing $this ...")
}

gradle.taskGraph.afterTask {
    if (state.failure != null) {
        println("FAILED")
    } else {
        println("done")
    }
}
```

```
> gradle -q broken
executing task ':ok' ...
done
executing task ':broken' ...
FAILED

FAILURE: Build failed with an exception.

* Where:
Build file '/home/user/gradle/samples/build.gradle' line: 6

* What went wrong:
Execution failed for task ':broken'.
> broken

* Try:
> Run with --stacktrace option to get the stack trace.
> Run with --info or --debug option to get more log output.
> Run with --scan to get full insights.

* Get more help at https://help.gradle.org

BUILD FAILED in 0s
```

```
> gradle -q broken
executing task ':ok' ...
done
executing task ':broken' ...
FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':broken'.
> broken

* Try:
> Run with --stacktrace option to get the stack trace.
> Run with --info or --debug option to get more log output.
> Run with --scan to get full insights.

* Get more help at https://help.gradle.org

BUILD FAILED in 0s
```

To receive these events via a listener instead of a closure, add a TaskExecutionListener to a project's TaskExecutionGraph.

Logging

The log is the main 'UI' of a build tool. If it is too verbose, real warnings and problems are easily hidden by this. On the other hand you need relevant information for figuring out if things have gone wrong. Gradle defines 6 log levels, as shown in Log levels. There are two Gradle-specific log levels, in addition to the ones you might normally see. Those levels are *QUIET* and *LIFECYCLE*. The latter is the default, and is used to report build progress.

Log levels

ERROR	Error messages
QUIET	Important information messages
WARNING	Warning messages
LIFECYCLE	Progress information messages
INFO	Information messages
DEBUG	Debug messages

NOTE	The rich components of the console (build status and work in progress area) are displayed regardless of the log level used. Before Gradle 4.0 those rich components were only displayed at log level `LIFECYCLE` or below.

Choosing a log level

You can use the command line switches shown in Log level command-line options to choose different log levels. You can also configure the log level using `gradle.properties`, see Gradle properties. In Stacktrace command-line options you find the command line switches which affect stacktrace logging.

Table 6. Log level command-line options

Option	Outputs Log Levels
no logging options	LIFECYCLE and higher
`-q` or `--quiet`	QUIET and higher
`-w` or `--warn`	WARN and higher
`-i` or `--info`	INFO and higher
`-d` or `--debug`	DEBUG and higher (that is, all log messages)

CAUTION	The `DEBUG` log level can expose security sensitive information to the console.

Stacktrace command-line options

`-s` or `--stacktrace`

Truncated stacktraces are printed. We recommend this over full stacktraces. Groovy full stacktraces are extremely verbose (Due to the underlying dynamic invocation mechanisms. Yet they usually do not contain relevant information for what has gone wrong in *your* code.) This option renders stacktraces for deprecation warnings.

`-S` or `--full-stacktrace`

The full stacktraces are printed out. This option renders stacktraces for deprecation warnings.

<No stacktrace options>

No stacktraces are printed to the console in case of a build error (e.g. a compile error). Only in case of internal exceptions will stacktraces be printed. If the `DEBUG` log level is chosen, truncated stacktraces are always printed.

Logging Sensitive Information

Running Gradle with the `DEBUG` log level can expose security sensitive information to the console and build log.

This information can include but is not limited to:

- Environment variables

- Private repository credentials
- Build cache & Gradle Enterprise Credentials
- Plugin Portal publishing credentials

The DEBUG log level should **not** be used when running on public Continuous Integration services. Build logs for public Continuous Integration services are world-viewable and can expose this sensitive information. Depending upon your organization's threat model, logging sensitive credentials in private CI may also be a vulnerability. Please discuss this with your organization's security team.

Some CI providers attempt to scrub sensitive credentials from logs; however, this will be imperfect and usually only scrubs exact-matches of pre-configured secrets.

If you believe a Gradle Plugin may be exposing sensitive information, please contact security@gradle.com for disclosure assistance.

Writing your own log messages

A simple option for logging in your build file is to write messages to standard output. Gradle redirects anything written to standard output to its logging system at the `QUIET` log level.

Example 197. Using stdout to write log messages

build.gradle

```
println 'A message which is logged at QUIET level'
```

build.gradle.kts

```
println("A message which is logged at QUIET level")
```

Gradle also provides a `logger` property to a build script, which is an instance of Logger. This interface extends the SLF4J `Logger` interface and adds a few Gradle specific methods to it. Below is an example of how this is used in the build script:

Example 198. Writing your own log messages

build.gradle

```
logger.quiet('An info log message which is always logged.')
logger.error('An error log message.')
logger.warn('A warning log message.')
logger.lifecycle('A lifecycle info log message.')
logger.info('An info log message.')
logger.debug('A debug log message.')
logger.trace('A trace log message.') // Gradle never logs TRACE level logs
```

build.gradle.kts

```
logger.quiet("An info log message which is always logged.")
logger.error("An error log message.")
logger.warn("A warning log message.")
logger.lifecycle("A lifecycle info log message.")
logger.info("An info log message.")
logger.debug("A debug log message.")
logger.trace("A trace log message.") // Gradle never logs TRACE level logs
```

Use the typical SLF4J pattern to replace a placeholder with an actual value as part of the log message.

Example 199. Writing a log message with placeholder

build.gradle

```
logger.info('A {} log message', 'info')
```

build.gradle.kts

```
logger.info("A {} log message", "info")
```

You can also hook into Gradle's logging system from within other classes used in the build (classes from the `buildSrc` directory for example). Simply use an SLF4J logger. You can use this logger the same way as you use the provided logger in the build script.

Example 200. Using SLF4J to write log messages

build.gradle

```
import org.slf4j.LoggerFactory

def slf4jLogger = LoggerFactory.getLogger('some-logger')
slf4jLogger.info('An info log message logged using SLF4j')
```

build.gradle.kts

```
import org.slf4j.LoggerFactory

val slf4jLogger = LoggerFactory.getLogger("some-logger")
slf4jLogger.info("An info log message logged using SLF4j")
```

Logging from external tools and libraries

Internally, Gradle uses Ant and Ivy. Both have their own logging system. Gradle redirects their logging output into the Gradle logging system. There is a 1:1 mapping from the Ant/Ivy log levels to the Gradle log levels, except the Ant/Ivy `TRACE` log level, which is mapped to Gradle `DEBUG` log level. This means the default Gradle log level will not show any Ant/Ivy output unless it is an error or a warning.

There are many tools out there which still use standard output for logging. By default, Gradle redirects standard output to the `QUIET` log level and standard error to the `ERROR` level. This behavior is configurable. The project object provides a LoggingManager, which allows you to change the log levels that standard out or error are redirected to when your build script is evaluated.

Example 201. Configuring standard output capture

build.gradle

```
logging.captureStandardOutput LogLevel.INFO
println 'A message which is logged at INFO level'
```

build.gradle.kts

```
logging.captureStandardOutput(LogLevel.INFO)
println("A message which is logged at INFO level")
```

To change the log level for standard out or error during task execution, tasks also provide a LoggingManager.

Example 202. Configuring standard output capture for a task

build.gradle

```
tasks.register('logInfo') {
    logging.captureStandardOutput LogLevel.INFO
    doFirst {
        println 'A task message which is logged at INFO level'
    }
}
```

build.gradle.kts

```
tasks.register("logInfo") {
    logging.captureStandardOutput(LogLevel.INFO)
    doFirst {
        println("A task message which is logged at INFO level")
    }
}
```

Gradle also provides integration with the Java Util Logging, Jakarta Commons Logging and Log4j logging toolkits. Any log messages which your build classes write using these logging toolkits will be redirected to Gradle's logging system.

Changing what Gradle logs

You can replace much of Gradle's logging UI with your own. You might do this, for example, if you want to customize the UI in some way - to log more or less information, or to change the formatting. You replace the logging using the Gradle.useLogger(java.lang.Object) method. This is accessible from a build script, or an init script, or via the embedding API. Note that this completely disables Gradle's default output. Below is an example init script which changes how task execution and build completion is logged.

Example 203. Customizing what Gradle logs

customLogger.init.gradle

```
useLogger(new CustomEventLogger())

class CustomEventLogger extends BuildAdapter implements TaskExecutionListener
{

    void beforeExecute(Task task) {
        println "[$task.name]"
    }

    void afterExecute(Task task, TaskState state) {
        println()
    }

    void buildFinished(BuildResult result) {
        println 'build completed'
        if (result.failure != null) {
            result.failure.printStackTrace()
        }
    }
}
```

customLogger.init.gradle.kts

```
useLogger(CustomEventLogger())

class CustomEventLogger() : BuildAdapter(), TaskExecutionListener {

    override fun beforeExecute(task: Task) {
        println("[${task.name}]")
    }

    override fun afterExecute(task: Task, state: TaskState) {
        println()
    }

    override fun buildFinished(result: BuildResult) {
        println("build completed")
        if (result.failure != null) {
            (result.failure as Throwable).printStackTrace()
        }
    }
}
```

```
$ gradle -I customLogger.init.gradle build

> Task :compile
[compile]
compiling source


> Task :testCompile
[testCompile]
compiling test source


> Task :test
[test]
running unit tests


> Task :build
[build]

build completed
3 actionable tasks: 3 executed
```

```
$ gradle -I customLogger.init.gradle.kts build

> Task :compile
[compile]
compiling source


> Task :testCompile
[testCompile]
compiling test source


> Task :test
[test]
running unit tests


> Task :build
[build]

build completed
3 actionable tasks: 3 executed
```

Your logger can implement any of the listener interfaces listed below. When you register a logger, only the logging for the interfaces that it implements is replaced. Logging for the other interfaces is

left untouched. You can find out more about the listener interfaces in Build lifecycle events.

- BuildListener
- ProjectEvaluationListener
- TaskExecutionGraphListener
- TaskExecutionListener
- TaskActionListener

Structuring and Building a Software Component with Gradle

If you are building a software of a certain size with Gradle, you have two basic structuring mechanisms. First, this chapter describes how to structure your software project using a Gradle multi-project. In this documentation, we consider this to be a single *software component* which is structured internally. Second, you may regard your software as a *software product* that is composed of multiple *software components* where each component is represented by a separate Gradle build. This is described in detail in the chapter on structuring software products with Gradle

Creating a multi-project build

A multi-project build in Gradle consists of one root project, and one or more subprojects.

A basic multi-project build contains a root project and a single subproject. This is a structure of a multi-project build that contains a single subproject called `app`:

Example 204. Basic multi-project build

Project layout

```
.
├─── app
│   ...
│   └─── build.gradle
└─── settings.gradle
```

Project layout

```
.
├─── app
│   ...
│   └─── build.gradle.kts
└─── settings.gradle.kts
```

This is the recommended project structure for starting any Gradle project. The build init plugin also generates skeleton projects that follow this structure - a root project with a single subproject.

Note that the root project does not have a Gradle build file, only a settings file that defines the subprojects to include.

settings.gradle

```
rootProject.name = 'basic-multiproject'
include 'app'
```

settings.gradle.kts

```
rootProject.name = "basic-multiproject"
include("app")
```

In this case, Gradle will look for a build file in the `app` directory.

We can view the structure of a multi-project build by running the `gradle projects` command.

```
> gradle -q projects

------------------------------------------------------------
Root project 'basic-multiproject'
------------------------------------------------------------

Root project 'basic-multiproject'
\--- Project ':app'

To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :app:tasks
```

Let's say the `app` subproject is a Java application by applying the application plugin and configuring the main class:

app/build.gradle

```
plugins {
    id 'application'
}

application {
    mainClass = 'com.example.Hello'
}
```

app/build.gradle.kts

```
plugins {
    id("application")
}

application {
    mainClass.set("com.example.Hello")
}
```

app/src/main/java/com/example/Hello.java

```
package com.example;

public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

We can then run the application by executing the `run` task from the application plugin.

```
> gradle -q run
Hello, world!
```

And that's how simple it is to create a basic multi-project build.

Adding subprojects

Let's say we want to add another subproject called `lib` to the previously created project. All we need to do is add another `include` statement in the root settings file:

settings.gradle

```
rootProject.name = 'basic-multiproject'
include 'app'
include 'lib'
```

settings.gradle.kts

```
rootProject.name = "basic-multiproject"
include("app")
include("lib")
```

Gradle will then look for the build file for the new subproject in the `lib/` subdirectory of the project:

Example 205. Multi-project build with two subprojects

Project layout

```
.
├─── app
│   ...
│   └─── build.gradle
├─── lib
│   ...
│   └─── build.gradle
└─── settings.gradle
```

Project layout

```
.
├─── app
│   ...
│   └─── build.gradle.kts
├─── lib
│   ...
│   └─── build.gradle.kts
└─── settings.gradle.kts
```

Next, will explore how build logic can be shared between subprojects and how subprojects can depend on one another.

Naming recommendations

As your project grows, naming and consistency gets increasingly more important. To keep your builds maintainable, we recommend the following:

1. *Keep default project names for subprojects*: It is possible to configure custom project names in the settings file. However, it's an unnecessary extra effort for the developers to keep track of which project belongs to what folders.
2. *Use kebab case formatting for all project names*: A kebab case formatting is when all letters lowercase, words separated with a dash ('-') character (e.g.`kebab-case-formatting`). This is already the de-facto pattern for many large projects. Besides, Gradle supports name abbreviation for kebab case names.
3. *Define the root project name in the settings file*: The ´rootProject.name´ effectively assigns a name to the build as a whole, which is used in reports like build scans. If the root project name is not set, the name will be the container directory name, which can be unstable (i.e. you can check out your project to any directory).

Sharing Build Logic between Subprojects

Convention Plugins

Usually, subprojects in a multi-project build share some common traits. For example, several subprojects may contain *code* in a particular programming language while another subproject may be dedicated for *documentation*. Code quality rules apply to all of the *code* subprojects but not the *documentation* subproject. At the same time, the subprojects that share one common trait may serve different purposes - they may produce different *artifact types* that further differentiate them, for example:

- *public libraries* - libraries that are published to some repository
- *internal libraries* - libraries on which other subprojects depend on internally within the project
- *command line applications* - applications with specific packaging requirements
- *web services* - applications with specific packaging requirements that are different from above
- etc

Some other *code* subprojects may be dedicated for *testing purposes* and so on.

The traits above identify a subproject's *type*. Or in other words, a subproject's *type* tells us what traits the project has.

Gradle's recommended way of organizing build logic is to use its plugin system. A plugin should define the *type* of a subproject. In fact, Gradle core plugins are modeled in the same way - for example, the Java Plugin configures a generic `java` project, while Java Library Plugin internally applies the Java Plugin and configures aspects specific to a Java library in addition. Similarly, the

Application Plugin applies and configures the Java Plugin and the Distribution Plugin.

You can compose custom build logic by applying and configuring both core and external plugins and create custom plugins that define new project *types* and configure conventions specific to your project or organization. For each of the example traits from the beginning of this section, we can write a plugin that encapsulates the logic common to the subproject of a given *type*.

We recommend putting source code and tests for the convention plugins in the special `buildSrc` directory in the root directory of the project. For more information about `buildSrc`, consult Using buildSrc to organize build logic.

Have a look at the sample that demonstrates a multi-project build that models the build logic using convention plugins.

Another, more complex and real-world example of a multi-project build that composes build logic using convention plugins is the build of the Gradle Build Tool itself.

Cross project configuration

Another, discouraged, way to share build logic between subproject is *cross project configuration* via the `subprojects {}` and `allprojects {}` DSL constructs. With cross configuration, build logic can be injected into a subproject and this is not obvious when looking at the subproject's build script, making it harder to understand the logic of a particular subproject. In the long run, cross configuration usually grows complex with more and more conditional logic and a higher maintenance burden. Cross configuration can also introduce configuration-time coupling between projects, which can prevent optimizations like configuration-on-demand from working properly.

There are two most common uses of cross-configuration that can be better modelled using convention plugins:

- Applying plugins or other configuration to subprojects of certain type. Often the cross-configuration section will do `if subproject is of type X, then configure Y`. This is equivalent to applying `X-conventions` plugin directly to a subproject.
- Extracting information from subprojects of a certain type. This use case can be modelled using outgoing configuration variants.

Fine Tuning the Project Layout

A multi-project build is a build where you build more than one project during a single execution of Gradle. You have to declare the projects taking part in the multi-project build in the settings file.

Project locations

Multi-project builds are always represented by a tree with a single root. Each element in the tree represents a project. A project has a path which denotes the position of the project in the multi-project build tree. In most cases the project path is consistent with the physical location of the project in the file system. However, this behavior is configurable. The project tree is created in the `settings.gradle` file. The location of the settings file is also the location of the root project.

Building the tree

In the settings file you can use the `include` method to build the project tree.

Example 206. Project layout

settings.gradle

```
include 'project1', 'project2:child', 'project3:child1'
```

settings.gradle.kts

```
include("project1", "project2:child", "project3:child1")
```

The `include` method takes project paths as arguments. The project path is assumed to be equal to the relative physical file system path. For example, a path 'services:api' is mapped by default to a folder 'services/api' (relative from the project root). You only need to specify the leaves of the tree. This means that the inclusion of the path 'services:hotels:api' will result in creating 3 projects: 'services', 'services:hotels' and 'services:hotels:api'. More examples of how to work with the project path can be found in the DSL documentation of Settings.include(java.lang.String[]).

Modifying elements of the project tree

The multi-project tree created in the settings file is made up of so called *project descriptors*. You can modify these descriptors in the settings file at any time. To access a descriptor you can do:

Example 207. Lookup of elements of the project tree

settings.gradle

```
include('project-a')
println rootProject.name
println project(':project-a').name
```

settings.gradle.kts

```
println(rootProject.name)
println(project(":project-a").name)
```

Using this descriptor you can change the name, project directory and build file of a project.

Example 208. Modification of elements of the project tree

settings.gradle

```
rootProject.name = 'main'
include('project-a')
project(':project-a').projectDir = file('../my-project-a')
project(':project-a').buildFileName = 'project-a.gradle'
```

settings.gradle.kts

```
rootProject.name = "main"
project(":project-a").projectDir = file("../my-project-a")
project(":project-a").buildFileName = "project-a.gradle"
```

Look at the ProjectDescriptor class in the API documentation for more information.

Declaring Dependencies between Subprojects

Project dependencies

What if one project needs the jar produced by another project on its compile classpath? What if it also requires the transitive dependencies of the other project? Obviously this is a very common use case for Java multi-project builds. As mentioned in Project dependencies, Gradle offers project dependencies for this.

Example 209. Project dependencies

Project layout

```
.
├─── buildSrc
│   ...
├─── api
│   ├─── src
│   │   └───...
│   └─── build.gradle
├─── services
│   └─── person-service
│       ├─── src
│       │   └───...
│       └─── build.gradle
├─── shared
│   ├─── src
│   │   └───...
│   └─── build.gradle
└─── settings.gradle
```

Project layout

```
.
├─── buildSrc
│   ...
├─── api
│   ├─── src
│   │   └───...
│   └─── build.gradle.kts
├─── services
│   └─── person-service
│       ├─── src
│       │   └───...
│       └─── build.gradle.kts
├─── shared
│   ├─── src
│   │   └───...
│   └─── build.gradle.kts
└─── settings.gradle.kts
```

We have the projects `shared`, `api` and `person-service`. The `person-service` project has a dependency on the other two projects. The `api` project has a dependency on the `shared` project. It has no build script and gets nothing injected by another build script. We use the `:` separator to define a project

path. Consult the DSL documentation of Settings.include(java.lang.String[]) for more information about defining project paths.

settings.gradle

```
rootProject.name = 'dependencies-java'
include 'api', 'shared', 'services:person-service'
```

buildSrc/src/main/groovy/myproject.java-conventions.gradle

```
plugins {
    id 'java'
}

group = 'com.example'
version = '1.0'

repositories {
    mavenCentral()
}

dependencies {
    testImplementation "junit:junit:4.13"
}
```

api/build.gradle

```
plugins {
    id 'myproject.java-conventions'
}

dependencies {
    implementation project(':shared')
}
```

shared/build.gradle

```
plugins {
    id 'myproject.java-conventions'
}
```

services/person-service/build.gradle

```
plugins {
    id 'myproject.java-conventions'
}

dependencies {
    implementation project(':shared')
    implementation project(':api')
}
```

settings.gradle.kts

```
rootProject.name = "dependencies-java"
include("api", "shared", "services:person-service")
```

buildSrc/src/main/kotlin/myproject.java-conventions.gradle.kts

```
plugins {
    id("java")
}

group = "com.example"
version = "1.0"

repositories {
    mavenCentral()
}

dependencies {
    testImplementation("junit:junit:4.13")
}
```

api/build.gradle.kts

```
plugins {
    id("myproject.java-conventions")
}

dependencies {
    implementation(project(":shared"))
}
```

shared/build.gradle.kts

```
plugins {
    id("myproject.java-conventions")
}
```

services/person-service/build.gradle.kts

```
plugins {
    id("myproject.java-conventions")
}

dependencies {
    implementation(project(":shared"))
    implementation(project(":api"))
}
```

Shared build logic is extracted into a *convention plugin* that is applied in the subprojects' build scripts that also define project dependencies. A project dependency is a special form of an execution dependency. It causes the other project to be built first and adds the jar with the classes of the other project to the classpath. It also adds the dependencies of the other project to the classpath. You can trigger a `gradle :api:compile`. First the `shared` project is built and then the `api` project is built. Project dependencies enable partial multi-project builds.

Depending on artifacts produced by another project

Project dependencies model dependencies between modules. Effectively, you are saying that you depend on the main output of another project. In a Java-based project that's usually a JAR file.

Sometimes you may want to depend on an output produced by another task. In turn, you'll want to make sure that the task is executed beforehand to produce that very output. Declaring a task dependency from one project to another is a poor way to model this kind of relationship and introduces unnecessary coupling. The recommended way to model such a dependency is to produce the output, mark it as an "outgoing" artifact or add it to the output of the `main` source set which you can depend on in the consuming project.

Let's say you are working in a multi-project build with the two subprojects `producer` and `consumer`. The subproject `producer` defines a task named `buildInfo` that generates a properties file containing build information e.g. the project version. You can then map the task provider to its output file and Gradle will automatically establish a task dependency.

Example 210. Task generating a property file containing build information

build.gradle

```
plugins {
    id 'java-library'
}

version = '1.0'

def buildInfo = tasks.register("buildInfo", BuildInfo) {
    version = project.version
    outputFile = layout.buildDirectory.file('generated-resources/build-
info.properties')
}

sourceSets {
    main {
        output.dir(buildInfo.map { it.outputFile.asFile.get().parentFile })
    }
}
```

build.gradle.kts

```
plugins {
    id("java-library")
}

version = "1.0"

val buildInfo by tasks.registering(BuildInfo::class) {
    version.set(project.version.toString())
    outputFile.set(layout.buildDirectory.file("generated-resources/build-
info.properties"))
}

sourceSets {
    main {
        output.dir(buildInfo.map { it.outputFile.asFile.get().parentFile })
    }
}
```

buildSrc/src/main/java/BuildInfo.java

```
public abstract class BuildInfo extends DefaultTask {

    @Input
    public abstract Property<String> getVersion();

    @OutputFile
    public abstract RegularFileProperty getOutputFile();

    @TaskAction
    public void create() throws IOException {
        Properties prop = new Properties();
        prop.setProperty("version", getVersion().get());
        try (OutputStream output = new FileOutputStream(getOutputFile().getAsFile
().get())) {
            prop.store(output, null);
        }
    }
}
```

The consuming project is supposed to be able to read the properties file at runtime. Declaring a project dependency on the producing project takes care of creating the properties beforehand and making it available to the runtime classpath.

Example 211. Declaring a project dependency on the project producing the properties file

build.gradle

```
dependencies {
    runtimeOnly project(':producer')
}
```

build.gradle.kts

```
dependencies {
    runtimeOnly(project(":producer"))
}
```

In the example above, the consumer now declares a dependency on the outputs of the `producer` project.

Depending on the main output artifact from another project is only one example. Gradle has one of the most powerful dependency management engines that allows you to share arbitrary artifacts

between projects and let Gradle build them on demand. For more details see the section on sharing outputs between projects.

Configuration time and execution time

Build phases describes the phases of every Gradle build. Let's zoom into the configuration and execution phases of a multi-project build. Configuration here means evaluating the build script file of a project, which includes downloading all plugins and build script dependencies. By default, the configuration of all projects happens before any task is executed. This means that when a single task, from a single project is requested, *all* projects of a multi-project build are configured first. The reason every project needs to be configured is to support the flexibility of accessing and changing any part of the Gradle project model.

Parallel project execution

With more and more CPU cores available on developer desktops and CI servers, it is important that Gradle is able to fully utilise these processing resources. More specifically, parallel execution attempts to:

- Reduce total build time for a multi-project build where execution is IO bound or otherwise does not consume all available CPU resources.
- Provide faster feedback for execution of small projects without awaiting completion of other projects.

Although Gradle already offers parallel test execution via Test.setMaxParallelForks(int) the feature described in this section is parallel execution at a project level.

Parallel project execution allows the separate projects in a decoupled multi-project build to be executed in parallel (see also Decoupled projects). While parallel execution does not strictly require decoupling at configuration time, the long-term goal is to provide a powerful set of features that will be available for fully decoupled projects. Such features include:

- Configuration on-demand.
- Configuration of projects in parallel.
- Re-use of configuration for unchanged projects.
- Project-level up-to-date checks.
- Using pre-built artifacts in the place of building dependent projects.

How does parallel execution work? First, you need to tell Gradle to use parallel mode. You can use the `--parallel` command line argument or configure your build environment (Gradle properties). Unless you provide a specific number of parallel threads, Gradle attempts to choose the right number based on available CPU cores. Every parallel worker exclusively owns a given project while executing a task. Task dependencies are fully supported and parallel workers will start executing upstream tasks first. Bear in mind that the alphabetical ordering of decoupled tasks, as can be seen during sequential execution, is not guaranteed in parallel mode. In other words, in parallel mode tasks will run as soon as their dependencies complete *and a task worker is available to run them*, which may be earlier than they would start during a sequential build. You should make sure that

task dependencies and task inputs/outputs are declared correctly to avoid ordering issues.

Decoupled Projects

Gradle allows any project to access any other project during both the configuration and execution phases. While this provides a great deal of power and flexibility to the build author, it also limits the flexibility that Gradle has when building those projects. For instance, this effectively prevents Gradle from correctly building multiple projects in parallel, configuring only a subset of projects, or from substituting a pre-built artifact in place of a project dependency.

Two projects are said to be *decoupled* if they do not directly access each other's project model. Decoupled projects may only interact in terms of declared dependencies: project dependencies and/or task dependencies. Any other form of project interaction (i.e. by modifying another project object or by reading a value from another project object) causes the projects to be coupled. The consequence of coupling during the configuration phase is that if gradle is invoked with the 'configuration on demand' option, the result of the build can be flawed in several ways. The consequence of coupling during execution phase is that if gradle is invoked with the parallel option, one project task runs too late to influence a task of a project building in parallel. Gradle does not attempt to detect coupling and warn the user, as there are too many possibilities to introduce coupling.

A very common way for projects to be coupled is by using configuration injection. It may not be immediately apparent, but using key Gradle features like the `allprojects` and `subprojects` keywords automatically cause your projects to be coupled. This is because these keywords are used in a `build.gradle` file, which defines a project. Often this is a "root project" that does nothing more than define common configuration, but as far as Gradle is concerned this root project is still a fully-fledged project, and by using `allprojects` that project is effectively coupled to all other projects. Coupling of the root project to subprojects does not impact configuration on-demand, but using the `allprojects` and `subprojects` in any subproject's `build.gradle` file will have an impact.

This means that using any form of shared build script logic or configuration injection (`allprojects`, `subprojects`, etc.) will cause your projects to be coupled. As we extend the concept of project decoupling and provide features that take advantage of decoupled projects, we will also introduce new features to help you to solve common use cases (like configuration injection) without causing your projects to be coupled.

In order to make good use of cross project configuration without running into issues for parallel and 'configuration on demand' options, follow these recommendations:

- Avoid a subproject's build script referencing other subprojects; preferring cross configuration from the root project.
- Avoid changing the configuration of other projects at execution time.

Configuration on demand

The *Configuration injection* feature and access to the complete project model are possible because every project is configured before the execution phase. Yet, this approach may not be the most efficient in a very large multi-project build. There are Gradle builds with a hierarchy of hundreds of subprojects. The configuration time of huge multi-project builds may become noticeable.

Configuration on demand attempts to configure only projects that are relevant for requested tasks, i.e. it only executes the build script file of projects that are participating in the build. This way, the configuration time of a large multi-project build can be reduced.

The configuration on demand feature is incubating, so not every build is guaranteed to work correctly. The feature should work very well for multi-project builds that have decoupled projects. In “configuration on demand” mode, projects are configured as follows:

- The root project is always configured.
- The project in the directory where the build is executed is also configured, but only when Gradle is executed without any tasks. This way the default tasks behave correctly when projects are configured on demand.
- The standard project dependencies are supported and makes relevant projects configured. If project A has a compile dependency on project B then building A causes configuration of both projects.
- The task dependencies declared via task path are supported and cause relevant projects to be configured. Example: `someTask.dependsOn(":some-other-project:someOtherTask")`
- A task requested via task path from the command line (or Tooling API) causes the relevant project to be configured. For example, building 'project-a:project-b:someTask' causes configuration of project-b.

To configure on demand with every build run see Gradle properties. To configure on demand just for a given build, see command-line performance-oriented options.

Organizing Gradle Projects

Source code and build logic of every software project should be organized in a meaningful way. This page lays out the best practices that lead to readable, maintainable projects. The following sections also touch on common problems and how to avoid them.

Separate language-specific source files

Gradle’s language plugins establish conventions for discovering and compiling source code. For example, a project applying the Java plugin will automatically compile the code in the directory `src/main/java`. Other language plugins follow the same pattern. The last portion of the directory path usually indicates the expected language of the source files.

Some compilers are capable of cross-compiling multiple languages in the same source directory. The Groovy compiler can handle the scenario of mixing Java and Groovy source files located in `src/main/groovy`. Gradle recommends that you place sources in directories according to their language, because builds are more performant and both the user and build can make stronger assumptions.

The following source tree contains Java and Kotlin source files. Java source files live in `src/main/java`, whereas Kotlin source files live in `src/main/kotlin`.

```
.
├─── build.gradle
└─── src
    └─── main
        ├─── java
        │    └─── HelloWorld.java
        └─── kotlin
             └─── Utils.kt
```

```
.
├─── build.gradle.kts
└─── src
    └─── main
        ├─── java
        │    └─── HelloWorld.java
        └─── kotlin
             └─── Utils.kt
```

Separate source files per test type

It's very common that a project defines and executes different types of tests e.g. unit tests, integration tests, functional tests or smoke tests. Optimally, the test source code for each test type should be stored in dedicated source directories. Separated test source code has a positive impact on maintainability and separation of concerns as you can run test types independent from each other.

Have a look at the sample that demonstrates how a separate integration tests configuration can be added to a Java-based project.

Use standard conventions as much as possible

All Gradle core plugins follow the software engineering paradigm convention over configuration. The plugin logic provides users with sensible defaults and standards, the conventions, in a certain context. Let's take the Java plugin as an example.

- It defines the directory `src/main/java` as the default source directory for compilation.
- The output directory for compiled source code and other artifacts (like the JAR file) is `build`.

By sticking to the default conventions, new developers to the project immediately know how to find their way around. While those conventions can be reconfigured, it makes it harder to build script users and authors to manage the build logic and its outcome. Try to stick to the default conventions as much as possible except if you need to adapt to the layout of a legacy project. Refer to the

reference page of the relevant plugin to learn about its default conventions.

Always define a settings file

Gradle tries to locate a `settings.gradle` (Groovy DSL) or a `settings.gradle.kts` (Kotlin DSL) file with every invocation of the build. For that purpose, the runtime walks the hierarchy of the directory tree up to the root directory. The algorithm stops searching as soon as it finds the settings file.

Always add a `settings.gradle` to the root directory of your build to avoid the initial performance impact. The file can either be empty or define the desired name of the project.

A multi-project build must have a `settings.gradle(.kts)` file in the root project of the multi-project hierarchy. It is required because the settings file defines which projects are taking part in a multi-project build. Besides defining included projects, you might need it to add libraries to your build script classpath.

The following example shows a standard Gradle project layout:

```
.
├─── settings.gradle
├─── subproject-one
│    └─── build.gradle
└─── subproject-two
     └─── build.gradle
```

```
.
├─── settings.gradle.kts
├─── subproject-one
│    └─── build.gradle.kts
└─── subproject-two
     └─── build.gradle.kts
```

Use `buildSrc` to abstract imperative logic

Complex build logic is usually a good candidate for being encapsulated either as custom task or binary plugin. Custom task and plugin implementations should not live in the build script. It is very convenient to use `buildSrc` for that purpose as long as the code does not need to be shared among multiple, independent projects.

The directory `buildSrc` is treated as an included build. Upon discovery of the directory, Gradle automatically compiles and tests this code and puts it in the classpath of your build script. For multi-project builds there can be only one `buildSrc` directory, which has to sit in the root project directory. `buildSrc` should be preferred over script plugins as it is easier to maintain, refactor and

test the code.

`buildSrc` uses the same source code conventions applicable to Java and Groovy projects. It also provides direct access to the Gradle API. Additional dependencies can be declared in a dedicated `build.gradle` under `buildSrc`.

Example 212. Custom buildSrc build script

buildSrc/build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'junit:junit:4.13'
}
```

buildSrc/build.gradle.kts

```
repositories {
    mavenCentral()
}

dependencies {
    testImplementation("junit:junit:4.13")
}
```

A typical project including `buildSrc` has the following layout. Any code under `buildSrc` should use a package similar to application code. Optionally, the `buildSrc` directory can host a build script if additional configuration is needed (e.g. to apply plugins or to declare dependencies).

```
.
├─── buildSrc
│    ├─── build.gradle
│    └─── src
│        ├─── main
│        │    └─── java
│        │        └─── com
│        │            └─── enterprise
│        │                ├─── Deploy.java
│        │                └─── DeploymentPlugin.java
│        └─── test
│            └─── java
│                └─── com
│                    └─── enterprise
│                        └─── DeploymentPluginTest.java
├─── settings.gradle
├─── subprojecto-one
│    └─── build.gradle.kts
└─── subproject-two
     └─── build.gradle.kts
```

```
.
├─── buildSrc
│    ├─── build.gradle.kts
│    └─── src
│        ├─── main
│        │    └─── java
│        │        └─── com
│        │            └─── enterprise
│        │                ├─── Deploy.java
│        │                └─── DeploymentPlugin.java
│        └─── test
│            └─── java
│                └─── com
│                    └─── enterprise
│                        └─── DeploymentPluginTest.java
├─── settings.gradle.kts
├─── subproject-one
│    └─── build.gradle.kts
└─── subproject-two
     └─── build.gradle.kts
```

NOTE	A change in `buildSrc` causes the whole project to become out-of-date. Thus, when making small incremental changes, the `--no-rebuild` command-line option is often helpful to get faster feedback. Remember to run a full build regularly or at least when you're done, though.

Declare properties in `gradle.properties` file

In Gradle, properties can be defined in the build script, in a `gradle.properties` file or as parameters on the command line.

It's common to declare properties on the command line for ad-hoc scenarios. For example you may want to pass in a specific property value to control runtime behavior just for this one invocation of the build. Properties in a build script can easily become a maintenance headache and convolute the build script logic. The `gradle.properties` helps with keeping properties separate from the build script and should be explored as viable option. It's a good location for placing properties that control the build environment.

A typical project setup places the `gradle.properties` file in the root directory of the build. Alternatively, the file can also live in the `GRADLE_USER_HOME` directory if you want to it apply to all builds on your machine.

```
.
├─── gradle.properties
└─── settings.gradle
├─── subproject-a
│    └─── build.gradle
└─── subproject-b
     └─── build.gradle
```

```
.
├─── gradle.properties
└─── settings.gradle.kts
├─── subproject-a
│    └─── build.gradle.kts
└─── subproject-b
     └─── build.gradle.kts
```

Avoid overlapping task outputs

Tasks should define inputs and outputs to get the performance benefits of incremental build functionality. When declaring the outputs of a task, make sure that the directory for writing outputs is unique among all the tasks in your project.

Intermingling or overwriting output files produced by different tasks compromises up-to-date checking causing slower builds. In turn, these filesystem changes may prevent Gradle's build cache from properly identifying and caching what would otherwise be cacheable tasks.

Standardizing builds with a custom Gradle distribution

Often enterprises want to standardize the build platform for all projects in the organization by defining common conventions or rules. You can achieve that with the help of initialization scripts. Initialization scripts make it extremely easy to apply build logic across all projects on a single machine. For example, to declare a in-house repository and its credentials.

There are some drawbacks to the approach. First of all, you will have to communicate the setup process across all developers in the company. Furthermore, updating the initialization script logic uniformly can prove challenging.

Custom Gradle distributions are a practical solution to this very problem. A custom Gradle distribution is comprised of the standard Gradle distribution plus one or many custom initialization scripts. The initialization scripts come bundled with the distribution and are applied every time the build is run. Developers only need to point their checked-in Wrapper files to the URL of the custom Gradle distribution.

Custom Gradle distributions may also contain a `gradle.properties` file in the root of the distribution, which provide an organization-wide set of properties that control the build environment.

The following steps are typical for creating a custom Gradle distribution:

1. Implement logic for downloading and repackaging a Gradle distribution.
2. Define one or many initialization scripts with the desired logic.
3. Bundle the initialization scripts with the Gradle distribution.
4. Upload the Gradle distribution archive to a HTTP server.
5. Change the Wrapper files of all projects to point to the URL of the custom Gradle distribution.

Example 213. Building a custom Gradle distribution

build.gradle

```
plugins {
    id 'base'
}

// This is defined in buildSrc
import org.gradle.distribution.DownloadGradle

version = '0.1'

tasks.register('downloadGradle', DownloadGradle) {
    description = 'Downloads the Gradle distribution with a given version.'
    gradleVersion = '4.6'
}

tasks.register('createCustomGradleDistribution', Zip) {
    description = 'Builds custom Gradle distribution and bundles
initialization scripts.'

    dependsOn downloadGradle

    def projectVersion = project.version
    archiveFileName = downloadGradle.gradleVersion.map { gradleVersion ->
        "mycompany-gradle-${gradleVersion}-${projectVersion}-bin.zip"
    }

    from zipTree(downloadGradle.destinationFile)

    from('src/init.d') {
        into "${downloadGradle.distributionNameBase.get()}/init.d"
    }
}
```

Best practices for authoring maintainable builds

Gradle has a rich API with several approaches to creating build logic. The associated flexibility can easily lead to unnecessarily complex builds with custom code commonly added directly to build scripts. In this chapter, we present several best practices that will help you develop expressive and maintainable builds that are easy to use.

NOTE The third-party Gradle lint plugin helps with enforcing a desired code style in build scripts if that's something that would interest you.

Avoid using imperative logic in scripts

The Gradle runtime does not enforce a specific style for build logic. For that very reason, it's easy to end up with a build script that mixes declarative DSL elements with imperative, procedural code. Let's talk about some concrete examples.

- *Declarative code:* Built-in, language-agnostic DSL elements (e.g. Project.dependencies{} or Project.repositories{}) or DSLs exposed by plugins
- *Imperative code:* Conditional logic or very complex task action implementations

The end goal of every build script should be to only contain declarative language elements which makes the code easier to understand and maintain. Imperative logic should live in binary plugins and which in turn is applied to the build script. As a side product, you automatically enable your team to reuse the plugin logic in other projects if you publish the artifact to a binary repository.

The following sample build shows a negative example of using conditional logic directly in the build script. While this code snippet is small, it is easy to imagine a full-blown build script using numerous procedural statements and the impact it would have on readability and maintainability. By moving the code into a class, it can also be tested individually.

Example 214. A build script using conditional logic to create a task

build.gradle

```
if (project.findProperty('releaseEngineer') != null) {
    tasks.register('release') {
        doLast {
            logger.quiet 'Releasing to production...'

            // release the artifact to production
        }
    }
}
```

build.gradle.kts

```
if (project.findProperty("releaseEngineer") != null) {
    tasks.register("release") {
        doLast {
            logger.quiet("Releasing to production...")

            // release the artifact to production
        }
    }
}
```

Let's compare the build script with the same logic implemented as a binary plugin. The code might look more involved at first but clearly looks more like typical application code. This particular plugin class lives in the `buildSrc` directory which makes it available to the build script automatically.

Example 215. A binary plugin implementing imperative logic

ReleasePlugin.java

```
package com.enterprise;

import org.gradle.api.Action;
import org.gradle.api.Plugin;
import org.gradle.api.Project;
import org.gradle.api.Task;
import org.gradle.api.tasks.TaskProvider;

public class ReleasePlugin implements Plugin<Project> {
    private static final String RELEASE_ENG_ROLE_PROP = "releaseEngineer";
    private static final String RELEASE_TASK_NAME = "release";

    @Override
    public void apply(Project project) {
        if (project.findProperty(RELEASE_ENG_ROLE_PROP) != null) {
            Task task = project.getTasks().create(RELEASE_TASK_NAME);

            task.doLast(new Action<Task>() {
                @Override
                public void execute(Task task) {
                    task.getLogger().quiet("Releasing to production...");

                    // release the artifact to production
                }
            });
        }
    }
}
```

Now that the build logic has been translated into a plugin, you can apply it in the build script. The build script has been shrunk from 8 lines of code to a one liner.

Example 216. A build script applying a plugin that encapsulates imperative logic

build.gradle

```
plugins {
    id 'com.enterprise.release'
}
```

build.gradle.kts

```
plugins {
    id("com.enterprise.release")
}
```

Avoid using internal Gradle APIs

Use of Gradle internal APIs in plugins and build scripts has the potential to break builds when either Gradle or plugins change.

The following packages are listed in the Gradle public API definition, with the exception of any subpackage with `internal` in the name:

```
org/gradle/*
org/gradle/api/**
org/gradle/authentication/**
org/gradle/buildinit/**
org/gradle/caching/**
org/gradle/concurrent/**
org/gradle/deployment/**
org/gradle/external/javadoc/**
org/gradle/ide/**
org/gradle/includedbuild/**
org/gradle/ivy/**
org/gradle/jvm/**
org/gradle/language/**
org/gradle/maven/**
org/gradle/nativeplatform/**
org/gradle/normalization/**
org/gradle/platform/**
org/gradle/play/**
org/gradle/plugin/devel/**
org/gradle/plugin/repository/*
org/gradle/plugin/use/*
org/gradle/plugin/management/*
org/gradle/plugins/**
org/gradle/process/**
org/gradle/testfixtures/**
org/gradle/testing/jacoco/**
org/gradle/tooling/**
org/gradle/swiftpm/**
org/gradle/model/**
org/gradle/testkit/**
org/gradle/testing/**
org/gradle/vcs/**
org/gradle/workers/**
```

Alternatives for oft-used internal APIs

To provide a nested DSL for your custom task, don't use `org.gradle.internal.reflect.Instantiator`; use ObjectFactory instead. It may also be helpful to read the chapter on lazy configuration.

Don't use `org.gradle.api.internal.ConventionMapping`. Use Provider and/or Property. You can find an example for capturing user input to configure runtime behavior in the implementing plugins section.

Instead of `org.gradle.internal.os.OperatingSystem`, use another method to detect operating system, such as Apache commons-lang SystemUtils or `System.getProperty("os.name")`.

Use other collections or I/O frameworks instead of `org.gradle.util.CollectionUtils`, `org.gradle.util.internal.GFileUtils`, and other classes under `org.gradle.util.*`.

Gradle plugin authors may find the Designing Gradle Plugins subsection on restricting the plugin

implementation to Gradle's public API helpful.

Follow conventions when declaring tasks

The task API gives a build author a lot of flexibility to declare tasks in a build script. For optimal readability and maintainability follow these rules:

- The task type should be the only key-value pair within the parentheses after the task name.
- Other configuration should be done within the task's configuration block.
- Task actions added when declaring a task should only be declared with the methods Task.doFirst{} or Task.doLast{}.
- When declaring an ad-hoc task — one that doesn't have an explicit type — you should use Task.doLast{} if you're only declaring a single action.
- A task should define a group and description.

Example 217. Definition of tasks following best practices

build.gradle

```
import com.enterprise.DocsGenerate

def generateHtmlDocs = tasks.register('generateHtmlDocs', DocsGenerate) {
    group = JavaBasePlugin.DOCUMENTATION_GROUP
    description = 'Generates the HTML documentation for this project.'
    title = 'Project docs'
    outputDir = layout.buildDirectory.dir('docs')
}

tasks.register('allDocs') {
    group = JavaBasePlugin.DOCUMENTATION_GROUP
    description = 'Generates all documentation for this project.'
    dependsOn generateHtmlDocs

    doLast {
        logger.quiet('Generating all documentation...')
    }
}
```

build.gradle.kts

```
import com.enterprise.DocsGenerate

tasks.register<DocsGenerate>("generateHtmlDocs") {
    group = JavaBasePlugin.DOCUMENTATION_GROUP
    description = "Generates the HTML documentation for this project."
    title.set("Project docs")
    outputDir.set(layout.buildDirectory.dir("docs"))
}

tasks.register("allDocs") {
    group = JavaBasePlugin.DOCUMENTATION_GROUP
    description = "Generates all documentation for this project."
    dependsOn("generateHtmlDocs")

    doLast {
        logger.quiet("Generating all documentation...")
    }
}
```

Improve task discoverability

Even new users to a build should to be able to find crucial information quickly and effortlessly. In Gradle you can declare a group and a description for any task of the build. The tasks report uses the assigned values to organize and render the task for easy discoverability. Assigning a group and description is most helpful for any task that you expect build users to invoke.

The example task `generateDocs` generates documentation for a project in the form of HTML pages. The task should be organized underneath the bucket `Documentation`. The description should express its intent.

Example 218. A task declaring the group and description

build.gradle

```
tasks.register('generateDocs') {
    group = 'Documentation'
    description = 'Generates the HTML documentation for this project.'

    doLast {
        // action implementation
    }
}
```

build.gradle.kts

```
tasks.register("generateDocs") {
    group = "Documentation"
    description = "Generates the HTML documentation for this project."

    doLast {
        // action implementation
    }
}
```

The output of the tasks report reflects the assigned values.

```
> gradle tasks

> Task :tasks

Documentation tasks
-------------------
generateDocs - Generates the HTML documentation for this project.
```

Minimize logic executed during the configuration phase

It's important for every build script developer to understand the different phases of the build lifecycle and their implications on performance and evaluation order of build logic. During the configuration phase the project and its domain objects should be *configured*, whereas the execution phase only executes the actions of the task(s) requested on the command line plus their dependencies. Be aware that any code that is not part of a task action will be executed with *every single run* of the build. A build scan can help you with identifying the time spent during each of the lifecycle phases. It's an invaluable tool for diagnosing common performance issues.

Let's consider the following incantation of the anti-pattern described above. In the build script you can see that the dependencies assigned to the configuration `printArtifactNames` are resolved outside of the task action.

Example 219. Executing logic during configuration should be avoided

build.gradle

```
dependencies {
    implementation 'log4j:log4j:1.2.17'
}

tasks.register('printArtifactNames') {
    // always executed
    def libraryNames = configurations.compileClasspath.collect { it.name }

    doLast {
        logger.quiet libraryNames
    }
}
```

build.gradle.kts

```
dependencies {
    implementation("log4j:log4j:1.2.17")
}

tasks.register("printArtifactNames") {
    // always executed
    val libraryNames = configurations.compileClasspath.get().map { it.name }

    doLast {
        logger.quiet(libraryNames.toString())
    }
}
```

The code for resolving the dependencies should be moved into the task action to avoid the performance impact of resolving the dependencies before they are actually needed.

Example 220. Executing logic during execution phase is preferred

build.gradle

```
dependencies {
    implementation 'log4j:log4j:1.2.17'
}

tasks.register('printArtifactNames') {
    doLast {
        def libraryNames = configurations.compileClasspath.collect { it.name
}
        logger.quiet libraryNames
    }
}
```

build.gradle.kts

```
dependencies {
    implementation("log4j:log4j:1.2.17")
}

tasks.register("printArtifactNames") {
    doLast {
        val libraryNames = configurations.compileClasspath.get().map {
it.name }
        logger.quiet(libraryNames.toString())
    }
}
```

Avoid using the `GradleBuild` task type

The GradleBuild task type allows a build script to define a task that invokes another Gradle build. The use of this type is generally discouraged. There are some corner cases where the invoked build doesn't expose the same runtime behavior as from the command line or through the Tooling API leading to unexpected results.

Usually, there's a better way to model the requirement. The appropriate approach depends on the problem at hand. Here're some options:

- Model the build as multi-project build if the intention is to execute tasks from different modules as unified build.

- Use composite builds for projects that are physically separated but should occasionally be built as a single unit.

Avoid inter-project configuration

Gradle does not restrict build script authors from reaching into the domain model from one project into another one in a multi-project build. Strongly-coupled projects hurts build execution performance as well as readability and maintainability of code.

The following practices should be avoided:

- Explicitly depending on a task from another project via Task.dependsOn(java.lang.Object...).
- Setting property values or calling methods on domain objects from another project.
- Executing another portion of the build with GradleBuild.
- Declaring unnecessary project dependencies.

Externalize and encrypt your passwords

Most builds need to consume one or many passwords. The reasons for this need may vary. Some builds need a password for publishing artifacts to a secured binary repository, other builds need a password for downloading binary files. Passwords should always kept safe to prevent fraud. Under no circumstance should you add the password to the build script in plain text or declare it in `gradle.properties` file in the project's directory. Those files usually live in a version control repository and can be viewed by anyone that has access to it.

Passwords together with any other sensitive data should be kept external from the version controlled project files. Gradle exposes an API for providing credentials in ProviderFactory as well as Artifact Repositories that allows to supply credential values using Gradle properties when they are needed by the build. This way the credentials can be stored in the `gradle.properties` file that resides in the user's home directory or be injected to the build using command line arguments or environment variables.

If you store sensitive credentials in user home's `gradle.properties`, consider encrypting them. At the moment Gradle does not provide a built-in mechanism for encrypting, storing and accessing passwords. A good solution for solving this problem is the Gradle Credentials plugin.

Lazy Configuration

As a build grows in complexity, knowing when and where a particular value is configured can become difficult to reason about. Gradle provides several ways to manage this complexity using *lazy configuration*.

Lazy properties

Gradle provides lazy properties, which delay the calculation of a property's value until it's actually required. These provide three main benefits to build script and plugin authors:

1. Build authors can wire together Gradle models without worrying when a particular property's

value will be known. For example, you may want to set the input source files of a task based on the source directories property of an extension but the extension property value isn't known until the build script or some other plugin configures them.

2. Build authors can wire an output property of a task into an input property of some other task and Gradle automatically determines the task dependencies based on this connection. Property instances carry information about which task, if any, produces their value. Build authors do not need to worry about keeping task dependencies in sync with configuration changes.
3. Build authors can avoid resource intensive work during the configuration phase, which can have a large impact on build performance. For example, when a configuration value comes from parsing a file but is only used when functional tests are run, using a property instance to capture this means that the file is parsed only when the functional tests are run, but not when, for example, `clean` is run.

Gradle represents lazy properties with two interfaces:

- Provider represents a value that can only be queried and cannot be changed.
  - Properties with these types are read-only.
  - The method Provider.get() returns the current value of the property.
  - A `Provider` can be created from another `Provider` using Provider.map(Transformer).
  - Many other types extend `Provider` and can be used where-ever a `Provider` is required.
- Property represents a value that can be queried and also changed.
  - Properties with these types are configurable.
  - `Property` extends the `Provider` interface.
  - The method Property.set(T) specifies a value for the property, overwriting whatever value may have been present.
  - The method Property.set(Provider) specifies a `Provider` for the value for the property, overwriting whatever value may have been present. This allows you to wire together `Provider` and `Property` instances before the values are configured.
  - A `Property` can be created by the factory method ObjectFactory.property(Class).

Lazy properties are intended to be passed around and only queried when required. Usually, this will happen during the execution phase. For more information about the Gradle build phases, please see Build Lifecycle.

The following demonstrates a task with a configurable `greeting` property and a read-only `message` property that is derived from this:

Example 221. Using a read-only and configurable property

build.gradle

```
// A task that displays a greeting
abstract class Greeting extends DefaultTask {
    // A configurable greeting
    @Input
    abstract Property<String> getGreeting()

    // Read-only property calculated from the greeting
    @Internal
    final Provider<String> message = greeting.map { it + ' from Gradle' }

    @TaskAction
    void printMessage() {
        logger.quiet(message.get())
    }
}

tasks.register("greeting", Greeting) {
    // Configure the greeting
    greeting.set('Hi')
    greeting = 'Hi' // Alternative notation to calling Property.set() - only
available in Groovy DSL
}
```

build.gradle.kts

```
// A task that displays a greeting
abstract class Greeting : DefaultTask() {
    // Configurable by the user
    @get:Input
    abstract val greeting: Property<String>

    // Read-only property calculated from the greeting
    @Internal
    val message: Provider<String> = greeting.map { it + " from Gradle" }

    @TaskAction
    fun printMessage() {
        logger.quiet(message.get())
    }
}

tasks.register<Greeting>("greeting") {
    // Configure the greeting
    greeting.set("Hi")

}
```

Output of `gradle greeting`

```
$ gradle greeting

> Task :greeting
Hi from Gradle

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

The `Greeting` task has a property of type `Property<String>` to represent the configurable greeting and a property of type `Provider<String>` to represent the calculated, read-only, message. The message `Provider` is created from the greeting `Property` using the `map()` method, and so its value is kept up-to-date as the value of the greeting property changes.

NOTE

Note that Gradle Groovy DSL generates setter methods for each `Property`-typed property in a task implementation. These setter methods allow you to configure the property using the assignment (`=`) operator as a convenience.

In Kotlin DSL, the `set()` method on the property needs to be used instead of `=`.

Creating a Property or Provider instance

Neither `Provider` nor its subtypes such as `Property` are intended to be implemented by a build script or plugin author. Gradle provides factory methods to create instances of these types instead. See the Quick Reference for all of the types and factories available. In the previous example, we have seen 2 factory methods:

- ObjectFactory.property(Class) create a new `Property` instance. An instance of the ObjectFactory can be referenced from Project.getObjects() or by injecting `ObjectFactory` through a constructor or method.
- Provider.map(Transformer) creates a new `Provider` from an existing `Provider` or `Property` instance.

A `Provider` can also be created by the factory method ProviderFactory.provider(Callable). You should prefer using `map()` instead, as this has some useful benefits, which we will see later.

NOTE

There are no specific methods create a provider using a `groovy.lang.Closure`. When writing a plugin or build script with Groovy, you can use the `map(Transformer)` method with a closure and Groovy will take care of converting the closure to a `Transformer`. You can see this in action in the previous example.

Similarly, when writing a plugin or build script with Kotlin, the Kotlin compiler will take care of converting a Kotlin function into a `Transformer`.

Connecting properties together

An important feature of lazy properties is that they can be connected together so that changes to one property are automatically reflected in other properties. Here's an example where the property of a task is connected to a property of a project extension:

Example 222. Connecting properties together

build.gradle

```
// A project extension
interface MessageExtension {
    // A configurable greeting
    Property<String> getGreeting()
}

// A task that displays a greeting
abstract class Greeting extends DefaultTask {
    // Configurable by the user
    @Input
    abstract Property<String> getGreeting()

    // Read-only property calculated from the greeting
    @Internal
    final Provider<String> message = greeting.map { it + ' from Gradle' }

    @TaskAction
    void printMessage() {
        logger.quiet(message.get())
    }
}

// Create the project extension
project.extensions.create('messages', MessageExtension)

// Create the greeting task
tasks.register("greeting", Greeting) {
    // Attach the greeting from the project extension
    // Note that the values of the project extension have not been configured
yet
    greeting = messages.greeting
}

messages {
    // Configure the greeting on the extension
    // Note that there is no need to reconfigure the task's `greeting`
property. This is automatically updated as the extension property changes
    greeting = 'Hi'
}
```

build.gradle.kts

```
// A project extension
interface MessageExtension {
    // A configurable greeting
    abstract val greeting: Property<String>
}

// A task that displays a greeting
abstract class Greeting : DefaultTask() {
    // Configurable by the user
    @get:Input
    abstract val greeting: Property<String>

    // Read-only property calculated from the greeting
    @Internal
    val message: Provider<String> = greeting.map { it + " from Gradle" }

    @TaskAction
    fun printMessage() {
        logger.quiet(message.get())
    }
}

// Create the project extension
val messages = project.extensions.create<MessageExtension>("messages")

// Create the greeting task
tasks.register<Greeting>("greeting") {
    // Attach the greeting from the project extension
    // Note that the values of the project extension have not been configured
yet
    greeting.set(messages.greeting)
}

messages.apply {
    // Configure the greeting on the extension
    // Note that there is no need to reconfigure the task's `greeting`
property. This is automatically updated as the extension property changes
    greeting.set("Hi")
}
```

Output of `gradle greeting`

```
$ gradle greeting

> Task :greeting
Hi from Gradle

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

This example calls the Property.set(Provider) method to attach a `Provider` to a `Property` to supply the value of the property. In this case, the `Provider` happens to be a `Property` as well, but you can connect any `Provider` implementation, for example one created using `Provider.map()`

Working with files

In Working with Files, we introduced four collection types for `File`-like objects:

Table 7. Collection of files recap

Read-only Type	Configurable Type
FileCollection	ConfigurableFileCollection
FileTree	ConfigurableFileTree

All of these types are also considered lazy types.

In this section, we are going to introduce more strongly typed models types to represent elements of the file system: Directory and RegularFile. These types shouldn't be confused with the standard Java File type as they are used to tell Gradle, and other people, that you expect more specific values such as a directory or a non-directory, regular file.

Gradle provides two specialized `Property` subtypes for dealing with values of these types: RegularFileProperty and DirectoryProperty. ObjectFactory has methods to create these: ObjectFactory.fileProperty() and ObjectFactory.directoryProperty().

A `DirectoryProperty` can also be used to create a lazily evaluated `Provider` for a `Directory` and `RegularFile` via DirectoryProperty.dir(String) and DirectoryProperty.file(String) respectively. These methods create providers whose values are calculated relative to the location for the `DirectoryProperty` they were created from. The values returned from these providers will reflect changes to the `DirectoryProperty`.

Example 223. Using file and directory property

build.gradle

```
// A task that generates a source file and writes the result to an output
directory
abstract class GenerateSource extends DefaultTask {
    // The configuration file to use to generate the source file
    @InputFile
    abstract RegularFileProperty getConfigFile()

    // The directory to write source files to
    @OutputDirectory
    abstract DirectoryProperty getOutputDir()

    @TaskAction
    def compile() {
        def inFile = configFile.get().asFile
        logger.quiet("configuration file = $inFile")
        def dir = outputDir.get().asFile
        logger.quiet("output dir = $dir")
        def className = inFile.text.trim()
        def srcFile = new File(dir, "${className}.java")
        srcFile.text = "public class ${className} { ... }"
    }
}

// Create the source generation task
tasks.register('generate', GenerateSource) {
    // Configure the locations, relative to the project and build directories
    configFile = layout.projectDirectory.file('src/config.txt')
    outputDir = layout.buildDirectory.dir('generated-source')
}

// Change the build directory
// Don't need to reconfigure the task properties. These are automatically
updated as the build directory changes
layout.buildDirectory = layout.projectDirectory.dir('output')
```

build.gradle.kts

```
// A task that generates a source file and writes the result to an output
directory
abstract class GenerateSource : DefaultTask() {
    // The configuration file to use to generate the source file
    @get:InputFile
    abstract val configFile: RegularFileProperty

    // The directory to write source files to
    @get:OutputDirectory
    abstract val outputDir: DirectoryProperty

    @TaskAction
    fun compile() {
        val inFile = configFile.get().asFile
        logger.quiet("configuration file = $inFile")
        val dir = outputDir.get().asFile
        logger.quiet("output dir = $dir")
        val className = inFile.readText().trim()
        val srcFile = File(dir, "${className}.java")
        srcFile.writeText("public class ${className} { }")
    }
}

// Create the source generation task
tasks.register<GenerateSource>("generate") {
    // Configure the locations, relative to the project and build directories
    configFile.set(layout.projectDirectory.file("src/config.txt"))
    outputDir.set(layout.buildDirectory.dir("generated-source"))
}

// Change the build directory
// Don't need to reconfigure the task properties. These are automatically
updated as the build directory changes
layout.buildDirectory.set(layout.projectDirectory.dir("output"))
```

Output of `gradle generate`

```
$ gradle generate

> Task :generate
configuration file = /home/user/gradle/samples/src/config.txt
output dir = /home/user/gradle/samples/output/generated-source

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Output of `gradle generate`

```
$ gradle generate

> Task :generate
configuration file = /home/user/gradle/samples/kotlin/src/config.txt
output dir = /home/user/gradle/samples/kotlin/output/generated-source

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

This example creates providers that represent locations in the project and build directories through Project.getLayout() with ProjectLayout.getBuildDirectory() and ProjectLayout.getProjectDirectory().

To close the loop, note that a `DirectoryProperty`, or a simple `Directory`, can be turned into a `FileTree` that allows the files and directories contained in the directory to be queried with DirectoryProperty.getAsFileTree() or Directory.getAsFileTree(). Moreover, from a `DirectoryProperty`, or a `Directory`, you can also create `FileCollection` instances containing a set of the files contained in the directory with DirectoryProperty.files(Object...) or Directory.files(Object...).

Working with task inputs and outputs

Many builds have several tasks connected together, where one task consumes the outputs of another task as an input. To make this work, we would need to configure each task to know where to look for its inputs and place its outputs, make sure that the producing and consuming tasks are configured with the same location, and attach task dependencies between the tasks. This can be cumbersome and brittle if any of these values are configurable by a user or configured by multiple plugins, as task properties need to be configured in the correct order and locations and task dependencies kept in sync as values change.

The `Property` API makes this easier by keeping track of not just the value for a property, which we have seen already, but also the task that produces the value, so that you don't have to specify it as well. As an example consider the following plugin with a producer and consumer task which are wired together:

Example 224. Implicit task input file dependency

build.gradle

```
abstract class Producer extends DefaultTask {
    @OutputFile
    abstract RegularFileProperty getOutputFile()

    @TaskAction
    void produce() {
        String message = 'Hello, World!'
        def output = outputFile.get().asFile
        output.text = message
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

abstract class Consumer extends DefaultTask {
    @InputFile
    abstract RegularFileProperty getInputFile()

    @TaskAction
    void consume() {
        def input = inputFile.get().asFile
        def message = input.text
        logger.quiet("Read '${message}' from ${input}")
    }
}

def producer = tasks.register("producer", Producer)
def consumer = tasks.register("consumer", Consumer)

consumer.configure {
    // Connect the producer task output to the consumer task input
    // Don't need to add a task dependency to the consumer task. This is
automatically added
    inputFile = producer.flatMap { it.outputFile }
}

producer.configure {
    // Set values for the producer lazily
    // Don't need to update the consumer.inputFile property. This is
automatically updated as producer.outputFile changes
    outputFile = layout.buildDirectory.file('file.txt')
}

// Change the build directory.
// Don't need to update producer.outputFile and consumer.inputFile. These are
automatically updated as the build directory changes
layout.buildDirectory = layout.projectDirectory.dir('output')
```

build.gradle.kts

```
abstract class Producer : DefaultTask() {
    @get:OutputFile
    abstract val outputFile: RegularFileProperty

    @TaskAction
    fun produce() {
        val message = "Hello, World!"
        val output = outputFile.get().asFile
        output.writeText( message)
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

abstract class Consumer : DefaultTask() {
    @get:InputFile
    abstract val inputFile: RegularFileProperty

    @TaskAction
    fun consume() {
        val input = inputFile.get().asFile
        val message = input.readText()
        logger.quiet("Read '${message}' from ${input}")
    }
}

val producer = tasks.register<Producer>("producer")
val consumer = tasks.register<Consumer>("consumer")

consumer {
    // Connect the producer task output to the consumer task input
    // Don't need to add a task dependency to the consumer task. This is
automatically added
    inputFile.set(producer.flatMap { it.outputFile })
}

producer {
    // Set values for the producer lazily
    // Don't need to update the consumer.inputFile property. This is
automatically updated as producer.outputFile changes
    outputFile.set(layout.buildDirectory.file("file.txt"))
}

// Change the build directory.
// Don't need to update producer.outputFile and consumer.inputFile. These are
automatically updated as the build directory changes
layout.buildDirectory.set(layout.projectDirectory.dir("output"))
```

Output of `gradle consumer`

```
$ gradle consumer

> Task :producer
Wrote 'Hello, World!' to /home/user/gradle/samples/output/file.txt

> Task :consumer
Read 'Hello, World!' from /home/user/gradle/samples/output/file.txt

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Output of `gradle consumer`

```
$ gradle consumer

> Task :producer
Wrote 'Hello, World!' to /home/user/gradle/samples/kotlin/output/file.txt

> Task :consumer
Read 'Hello, World!' from /home/user/gradle/samples/kotlin/output/file.txt

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

In the example above, the task outputs and inputs are connected before any location is defined. The setters can be called at any time before the task is executed and the change will automatically affect all related input and output properties.

Another important thing to note in this example is the absence of any explicit task dependency. Task outputs represented using `Providers` keep track of which task produces their value, and using them as task inputs will implicitly add the correct task dependencies.

Implicit task dependencies also works for input properties that are not files.

Example 225. Implicit task input dependency

build.gradle

```
abstract class Producer extends DefaultTask {
    @OutputFile
    abstract RegularFileProperty getOutputFile()

    @TaskAction
    void produce() {
        String message = 'Hello, World!'
        def output = outputFile.get().asFile
        output.text = message
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

abstract class Consumer extends DefaultTask {
    @Input
    abstract Property<String> getMessage()

    @TaskAction
    void consume() {
        logger.quiet(message.get())
    }
}

def producer = tasks.register('producer', Producer) {
    // Set values for the producer lazily
    // Don't need to update the consumer.inputFile property. This is
automatically updated as producer.outputFile changes
    outputFile = layout.buildDirectory.file('file.txt')
}
tasks.register('consumer', Consumer) {
    // Connect the producer task output to the consumer task input
    // Don't need to add a task dependency to the consumer task. This is
automatically added
    message = producer.map { it.outputFile.get().asFile.text }
}
```

build.gradle.kts

```
abstract class Producer : DefaultTask() {
    @get:OutputFile
    abstract val outputFile: RegularFileProperty

    @TaskAction
    fun produce() {
        val message = "Hello, World!"
        val output = outputFile.get().asFile
        output.writeText( message)
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

abstract class Consumer : DefaultTask() {
    @get:Input
    abstract val message: Property<String>

    @TaskAction
    fun consume() {
        logger.quiet(message.get())
    }
}

val producer = tasks.register<Producer>("producer") {
    // Set values for the producer lazily
    // Don't need to update the consumer.inputFile property. This is
automatically updated as producer.outputFile changes
    outputFile.set(layout.buildDirectory.file("file.txt"))
}
tasks.register<Consumer>("consumer") {
    // Connect the producer task output to the consumer task input
    // Don't need to add a task dependency to the consumer task. This is
automatically added
    message.set(producer.map { it.outputFile.get().asFile.readText() })
}
```

Output of `gradle consumer`

```
$ gradle consumer

> Task :producer
Wrote 'Hello, World!' to /home/user/gradle/samples/build/file.txt

> Task :consumer
Hello, World!

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Output of `gradle consumer`

```
$ gradle consumer

> Task :producer
Wrote 'Hello, World!' to /home/user/gradle/samples/kotlin/build/file.txt

> Task :consumer
Hello, World!

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Working with collections

Gradle provides two lazy property types to help configure `Collection` properties. These work exactly like any other `Provider` and, just like file providers, they have additional modeling around them:

- For `List` values the interface is called ListProperty. You can create a new `ListProperty` using ObjectFactory.listProperty(Class) and specifying the element type.
- For `Set` values the interface is called SetProperty. You can create a new `SetProperty` using ObjectFactory.setProperty(Class) and specifying the element type.

This type of property allows you to overwrite the entire collection value with HasMultipleValues.set(Iterable) and HasMultipleValues.set(Provider) or add new elements through the various `add` methods:

- HasMultipleValues.add(T): Add a single element to the collection
- HasMultipleValues.add(Provider): Add a lazily calculated element to the collection
- HasMultipleValues.addAll(Provider): Add a lazily calculated collection of elements to the list

Just like every `Provider`, the collection is calculated when Provider.get() is called. The following example shows the ListProperty in action:

Example 226. List property

build.gradle

```
abstract class Producer extends DefaultTask {
    @OutputFile
    abstract RegularFileProperty getOutputFile()

    @TaskAction
    void produce() {
        String message = 'Hello, World!'
        def output = outputFile.get().asFile
        output.text = message
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

abstract class Consumer extends DefaultTask {
    @InputFiles
    abstract ListProperty<RegularFile> getInputFiles()

    @TaskAction
    void consume() {
        inputFiles.get().each { inputFile ->
            def input = inputFile.asFile
            def message = input.text
            logger.quiet("Read '${message}' from ${input}")
        }
    }
}

def producerOne = tasks.register('producerOne', Producer)
def producerTwo = tasks.register('producerTwo', Producer)
tasks.register('consumer', Consumer) {
    // Connect the producer task outputs to the consumer task input
    // Don't need to add task dependencies to the consumer task. These are
automatically added
    inputFiles.add(producerOne.get().outputFile)
    inputFiles.add(producerTwo.get().outputFile)
}

// Set values for the producer tasks lazily
// Don't need to update the consumer.inputFiles property. This is
automatically updated as producer.outputFile changes
producerOne.configure { outputFile = layout.buildDirectory.file('one.txt') }
producerTwo.configure { outputFile = layout.buildDirectory.file('two.txt') }

// Change the build directory.
// Don't need to update the task properties. These are automatically updated
as the build directory changes
layout.buildDirectory = layout.projectDirectory.dir('output')
```

build.gradle.kts

```
abstract class Producer : DefaultTask() {
    @get:OutputFile
    abstract val outputFile: RegularFileProperty

    @TaskAction
    fun produce() {
        val message = "Hello, World!"
        val output = outputFile.get().asFile
        output.writeText( message)
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

abstract class Consumer : DefaultTask() {
    @get:InputFiles
    abstract val inputFiles: ListProperty<RegularFile>

    @TaskAction
    fun consume() {
        inputFiles.get().forEach { inputFile ->
            val input = inputFile.asFile
            val message = input.readText()
            logger.quiet("Read '${message}' from ${input}")
        }
    }
}

val producerOne = tasks.register<Producer>("producerOne")
val producerTwo = tasks.register<Producer>("producerTwo")
tasks.register<Consumer>("consumer") {
    // Connect the producer task outputs to the consumer task input
    // Don't need to add task dependencies to the consumer task. These are
automatically added
    inputFiles.add(producerOne.get().outputFile)
    inputFiles.add(producerTwo.get().outputFile)
}

// Set values for the producer tasks lazily
// Don't need to update the consumer.inputFiles property. This is
automatically updated as producer.outputFile changes
producerOne { outputFile.set(layout.buildDirectory.file("one.txt")) }
producerTwo { outputFile.set(layout.buildDirectory.file("two.txt")) }

// Change the build directory.
// Don't need to update the task properties. These are automatically updated
as the build directory changes
layout.buildDirectory.set(layout.projectDirectory.dir("output"))
```

Output of `gradle consumer`

```
$ gradle consumer

> Task :producerOne
Wrote 'Hello, World!' to /home/user/gradle/samples/output/one.txt

> Task :producerTwo
Wrote 'Hello, World!' to /home/user/gradle/samples/output/two.txt

> Task :consumer
Read 'Hello, World!' from /home/user/gradle/samples/output/one.txt
Read 'Hello, World!' from /home/user/gradle/samples/output/two.txt

BUILD SUCCESSFUL in 0s
3 actionable tasks: 3 executed
```

Output of `gradle consumer`

```
$ gradle consumer

> Task :producerOne
Wrote 'Hello, World!' to /home/user/gradle/samples/kotlin/output/one.txt

> Task :producerTwo
Wrote 'Hello, World!' to /home/user/gradle/samples/kotlin/output/two.txt

> Task :consumer
Read 'Hello, World!' from /home/user/gradle/samples/kotlin/output/one.txt
Read 'Hello, World!' from /home/user/gradle/samples/kotlin/output/two.txt

BUILD SUCCESSFUL in 0s
3 actionable tasks: 3 executed
```

Working with maps

Gradle provides a lazy MapProperty type to allow `Map` values to be configured. You can create a `MapProperty` instance using ObjectFactory.mapProperty(Class, Class).

Similar to other property types, a `MapProperty` has a set() method that you can use to specify the value for the property. There are some additional methods to allow entries with lazy values to be added to the map.

Example 227. Map property

build.gradle

```
abstract class Generator extends DefaultTask {
    @Input
    abstract MapProperty<String, Integer> getProperties()

    @TaskAction
    void generate() {
        properties.get().each { key, value ->
            logger.quiet("${key} = ${value}")
        }
    }
}

// Some values to be configured later
def b = 0
def c = 0

tasks.register('generate', Generator) {
    properties.put("a", 1)
    // Values have not been configured yet
    properties.put("b", providers.provider { b })
    properties.putAll(providers.provider { [c: c, d: c + 1] })
}

// Configure the values. There is no need to reconfigure the task
b = 2
c = 3
```

build.gradle.kts

```
abstract class Generator: DefaultTask() {
    @get:Input
    abstract val properties: MapProperty<String, Int>

    @TaskAction
    fun generate() {
        properties.get().forEach { entry ->
            logger.quiet("${entry.key} = ${entry.value}")
        }
    }
}

// Some values to be configured later
var b = 0
var c = 0

tasks.register<Generator>("generate") {
    properties.put("a", 1)
    // Values have not been configured yet
    properties.put("b", providers.provider { b })
    properties.putAll(providers.provider { mapOf("c" to c, "d" to c + 1) })
}

// Configure the values. There is no need to reconfigure the task
b = 2
c = 3
```

Output of `gradle consumer`

```
$ gradle generate

> Task :generate
a = 1
b = 2
c = 3
d = 4

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Applying a convention to a property

Often you want to apply some *convention*, or default value, to a property to be used if no value has been configured for the property. You can use the `convention()` method for this. This method accepts either a value or a `Provider` and this will be used as the value until some other value is

configured.

Example 228. Property conventions

build.gradle

```
def property = objects.property(String)

// Set a convention
property.convention("convention 1")
println("value = " + property.get())

// Can replace the convention
property.convention("convention 2")
println("value = " + property.get())

property.set("value")

// Once a value is set, the convention is ignored
property.convention("ignored convention")
println("value = " + property.get())
```

build.gradle.kts

```
val property = objects.property(String::class)

// Set a convention
property.convention("convention 1")
println("value = " + property.get())

// Can replace the convention
property.convention("convention 2")
println("value = " + property.get())

property.set("value")

// Once a value is set, the convention is ignored
property.convention("ignored convention")
println("value = " + property.get())
```

Output of `gradle show`

```
$ gradle show

> Task :show
value = convention 1
value = convention 2
value = value

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Making a property unmodifiable

Most properties of a task or project are intended to be configured by plugins or build scripts and then the resulting value used to do something useful. For example, a property that specifies the output directory for a compilation task may start off with a value specified by a plugin, then a build script might change the value to some custom location, then this value is used by the task when it runs. However, once the task starts to run, we want to prevent any further change to the property. This way we avoid errors that result from different consumers, such as the task action or Gradle's up-to-date checks or build caching or other tasks, using different values for the property.

Lazy properties provide several methods that you can use to disallow changes to their value once the value has been configured. The finalizeValue() method calculates the *final* value for the property and prevents further changes to the property. When the value of the property comes from a `Provider`, the provider is queried for its current value and the result becomes the final value for the property. This final value replaces the provider and the property no longer tracks the value of the provider. Calling this method also makes a property instance unmodifiable and any further attempts to change the value of the property will fail. Gradle automatically makes the properties of a task final when the task starts execution.

The finalizeValueOnRead() method is similar, except that the property's final value is not calculated until the value of the property is queried. In other words, this method calculates the final value lazily as required, whereas `finalizeValue()` calculates the final value eagerly. This method can be used when the value may be expensive to calculate or may not have been configured yet, but you also want to ensure that all consumers of the property see the same value when they query the value.

Guidelines

This section will introduce guidelines to be successful with the Provider API. To see those guidelines in action, have a look at gradle-site-plugin, a Gradle plugin demonstrating established techniques and practices for plugin development.

- The Property and Provider types have all of the overloads you need to query or configure a value. For this reason, you should follow the following guidelines:
  - For configurable properties, expose the Property directly through a single getter.
  - For non-configurable properties, expose an Provider directly through a single getter.

- Avoid simplifying calls like `obj.getProperty().get()` and `obj.getProperty().set(T)` in your code by introducing additional getters and setters.
- When migrating your plugin to use providers, follow these guidelines:
  - If it's a new property, expose it as a Property or Provider using a single getter.
  - If it's incubating, change it to use a Property or Provider using a single getter.
  - If it's a stable property, add a new Property or Provider and deprecate the old one. You should wire the old getter/setters into the new property as appropriate.

Future development

Going forward, new properties will use the Provider API. The Groovy Gradle DSL adds convenience methods to make the use of Providers mostly transparent in build scripts. Existing tasks will have their existing "raw" properties replaced by Providers as needed and in a backwards compatible way. New tasks will be designed with the Provider API.

Provider Files API Reference

Use these types for *read-only* values:

Provider<RegularFile>

File on disk

Factories

- Provider.map(Transformer).
- Provider.flatMap(Transformer).
- DirectoryProperty.file(String)

Provider<Directory>

Directory on disk

Factories

- Provider.map(Transformer).
- Provider.flatMap(Transformer).
- DirectoryProperty.dir(String)

FileCollection

Unstructured collection of files

Factories

- Project.files(Object[])
- ProjectLayout.files(Object...)
- DirectoryProperty.files(Object...)

FileTree

Hierarchy of files

Factories

- Project.fileTree(Object) will produce a ConfigurableFileTree, or you can use Project.zipTree(Object) and Project.tarTree(Object)
- DirectoryProperty.getAsFileTree()

Property Files API Reference

Use these types for *mutable* values:

RegularFileProperty

File on disk

Factories

- ObjectFactory.fileProperty()

DirectoryProperty

Directory on disk

Factories

- ObjectFactory.directoryProperty()

ConfigurableFileCollection

Unstructured collection of files

Factories

- ObjectFactory.fileCollection()

ConfigurableFileTree

Hierarchy of files

Factories

- ObjectFactory.fileTree()

SourceDirectorySet

Hierarchy of source directories

Factories

- ObjectFactory.sourceDirectorySet(String, String)

Lazy Collections API Reference

Use these types for *mutable* values:

ListProperty<T>

a property whose value is `List<T>`

Factories

- ObjectFactory.listProperty(Class)

SetProperty<T>

a property whose value is `Set<T>`

Factories

- ObjectFactory.setProperty(Class)

Lazy Objects API Reference

Use these types for *read only* values:

Provider<T>

a property whose value is an instance of `T`

Factories

- Provider.map(Transformer).
- Provider.flatMap(Transformer).
- ProviderFactory.provider(Callable). Always prefer one of the other factory methods over this method.

Use these types for *mutable* values:

Property<T>

a property whose value is an instance of `T`

Factories

- ObjectFactory.property(Class)

Testing Build Logic with TestKit

The Gradle TestKit (a.k.a. just TestKit) is a library that aids in testing Gradle plugins and build logic generally. At this time, it is focused on *functional* testing. That is, testing build logic by exercising it as part of a programmatically executed build. Over time, the TestKit will likely expand to facilitate other kinds of tests.

Usage

To use the TestKit, include the following in your plugin's build:

Example 229. Declaring the TestKit dependency

build.gradle

```
dependencies {
    testImplementation gradleTestKit()
}
```

build.gradle.kts

```
dependencies {
    testImplementation(gradleTestKit())
}
```

The `gradleTestKit()` encompasses the classes of the TestKit, as well as the Gradle Tooling API client. It does not include a version of JUnit, TestNG, or any other test execution framework. Such a dependency must be explicitly declared.

Example 230. Declaring the JUnit dependency

build.gradle

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:5.7.1")
}

tasks.withType(Test).configureEach {
    useJUnitPlatform()
}
```

build.gradle.kts

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:5.7.1")
}

tasks.withType<Test>().configureEach {
    useJUnitPlatform()
}
```

Functional testing with the Gradle runner

The GradleRunner facilitates programmatically executing Gradle builds, and inspecting the result.

A contrived build can be created (e.g. programmatically, or from a template) that exercises the "logic under test". The build can then be executed, potentially in a variety of ways (e.g. different combinations of tasks and arguments). The correctness of the logic can then be verified by asserting the following, potentially in combination:

- The build's output;
- The build's logging (i.e. console output);
- The set of tasks executed by the build and their results (e.g. FAILED, UP-TO-DATE etc.).

After creating and configuring a runner instance, the build can be executed via the GradleRunner.build() or GradleRunner.buildAndFail() methods depending on the anticipated outcome.

The following demonstrates the usage of the Gradle runner in a Java JUnit test:

Example: Using GradleRunner with Java and JUnit

BuildLogicFunctionalTest.java

```
import org.gradle.testkit.runner.BuildResult;
import org.gradle.testkit.runner.GradleRunner;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.io.TempDir;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

import static org.gradle.testkit.runner.TaskOutcome.SUCCESS;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

public class BuildLogicFunctionalTest {

    @TempDir File testProjectDir;
    private File settingsFile;
    private File buildFile;

    @BeforeEach
    public void setup() {
        settingsFile = new File(testProjectDir, "settings.gradle");
        buildFile = new File(testProjectDir, "build.gradle");
    }
```

```
    @Test
    public void testHelloWorldTask() throws IOException {
        writeFile(settingsFile, "rootProject.name = 'hello-world'");
        String buildFileContent = "task helloWorld {" +
                                  "    doLast {" +
                                  "        println 'Hello world!'" +
                                  "    }" +
                                  "}";
        writeFile(buildFile, buildFileContent);

        BuildResult result = GradleRunner.create()
            .withProjectDir(testProjectDir)
            .withArguments("helloWorld")
            .build();

        assertTrue(result.getOutput().contains("Hello world!"));
        assertEquals(SUCCESS, result.task(":helloWorld").getOutcome());
    }

    private void writeFile(File destination, String content) throws IOException {
        BufferedWriter output = null;
        try {
            output = new BufferedWriter(new FileWriter(destination));
            output.write(content);
        } finally {
            if (output != null) {
                output.close();
            }
        }
    }
}
```

Any test execution framework can be used.

As Gradle build scripts can also be written in the Groovy programming language, it is often a productive choice to write Gradle functional tests in Groovy. Furthermore, it is recommended to use the (Groovy based) Spock test execution framework as it offers many compelling features over the use of JUnit.

The following demonstrates the usage of the Gradle runner in a Groovy Spock test:

Example: Using GradleRunner with Groovy and Spock

BuildLogicFunctionalTest.groovy

```
import org.gradle.testkit.runner.GradleRunner
import static org.gradle.testkit.runner.TaskOutcome.*
import spock.lang.TempDir
import spock.lang.Specification

class BuildLogicFunctionalTest extends Specification {
    @TempDir File testProjectDir
    File settingsFile
    File buildFile

    def setup() {
        settingsFile = new File(testProjectDir, 'settings.gradle')
        buildFile = new File(testProjectDir, 'build.gradle')
    }

    def "hello world task prints hello world"() {
        given:
        settingsFile << "rootProject.name = 'hello-world'"
        buildFile << """
            task helloWorld {
                doLast {
                    println 'Hello world!'
                }
            }
        """

        when:
        def result = GradleRunner.create()
            .withProjectDir(testProjectDir)
            .withArguments('helloWorld')
            .build()

        then:
        result.output.contains('Hello world!')
        result.task(":helloWorld").outcome == SUCCESS
    }
}
```

It is a common practice to implement any custom build logic (like plugins and task types) that is more complex in nature as external classes in a standalone project. The main driver behind this approach is bundle the compiled code into a JAR file, publish it to a binary repository and reuse it across various projects.

Getting the plugin-under-test into the test build

The GradleRunner uses the Tooling API to execute builds. An implication of this is that the builds are executed in a separate process (i.e. not the same process executing the tests). Therefore, the test build does not share the same classpath or classloaders as the test process and the code under test

is not implicitly available to the test build.

Starting with version 2.13, Gradle provides a conventional mechanism to inject the code under test into the test build.

Automatic injection with the Java Gradle Plugin Development plugin

The Java Gradle Plugin development plugin can be used to assist in the development of Gradle plugins. Starting with Gradle version 2.13, the plugin provides a direct integration with TestKit. When applied to a project, the plugin automatically adds the `gradleTestKit()` dependency to the `testApi` configuration. Furthermore, it automatically generates the classpath for the code under test and injects it via GradleRunner.withPluginClasspath() for any `GradleRunner` instance created by the user. It's important to note that the mechanism currently *only* works if the plugin under test is applied using the plugins DSL. If the target Gradle version is prior to 2.8, automatic plugin classpath injection is not performed.

The plugin uses the following conventions for applying the TestKit dependency and injecting the classpath:

- Source set containing code under test: `sourceSets.main`
- Source set used for injecting the plugin classpath: `sourceSets.test`

Any of these conventions can be reconfigured with the help of the class GradlePluginDevelopmentExtension.

The following Groovy-based sample demonstrates how to automatically inject the plugin classpath by using the standard conventions applied by the Java Gradle Plugin Development plugin.

Example 231. Using the Java Gradle Development plugin for generating the plugin metadata

build.gradle

```
plugins {
    id 'groovy'
    id 'java-gradle-plugin'
}

dependencies {
    testImplementation('org.spockframework:spock-core:2.1-groovy-3.0') {
        exclude group: 'org.codehaus.groovy'
    }
}
```

build.gradle.kts

```
plugins {
    groovy
    `java-gradle-plugin`
}

dependencies {
    testImplementation("org.spockframework:spock-core:2.1-groovy-3.0") {
        exclude(group = "org.codehaus.groovy")
    }
}
```

Example: Automatically injecting the code under test classes into test builds

src/test/groovy/org/gradle/sample/BuildLogicFunctionalTest.groovy

```
def "hello world task prints hello world"() {
    given:
    settingsFile << "rootProject.name = 'hello-world'"
    buildFile << """
        plugins {
            id 'org.gradle.sample.helloworld'
        }
    """

    when:
    def result = GradleRunner.create()
        .withProjectDir(testProjectDir)
        .withArguments('helloWorld')
        .withPluginClasspath()
        .build()

    then:
    result.output.contains('Hello world!')
    result.task(":helloWorld").outcome == SUCCESS
}
```

The following build script demonstrates how to reconfigure the conventions provided by the Java Gradle Plugin Development plugin for a project that uses a custom `Test` source set.

NOTE	A new configuration DSL for modeling the below `functionalTest` suite is available via the incubating JVM Test Suite plugin.

Example 232. Reconfiguring the classpath generation conventions of the Java Gradle Development plugin

build.gradle

```
plugins {
    id 'groovy'
    id 'java-gradle-plugin'
}

def functionalTest = sourceSets.create('functionalTest')
def functionalTestTask = tasks.register('functionalTest', Test) {
    group = 'verification'
    testClassesDirs = sourceSets.functionalTest.output.classesDirs
    classpath = sourceSets.functionalTest.runtimeClasspath
    useJUnitPlatform()
}

tasks.named("check") {
    dependsOn functionalTestTask
}

gradlePlugin {
    testSourceSets sourceSets.functionalTest
}

dependencies {
    functionalTestImplementation('org.spockframework:spock-core:2.1-groovy-
3.0') {
        exclude group: 'org.codehaus.groovy'
    }
}
```

build.gradle.kts

```
plugins {
    groovy
    `java-gradle-plugin`
}

val functionalTest = sourceSets.create("functionalTest")
val functionalTestTask = tasks.register<Test>("functionalTest") {
    group = "verification"
    testClassesDirs = functionalTest.output.classesDirs
    classpath = functionalTest.runtimeClasspath
    useJUnitPlatform()
}

tasks.check {
    dependsOn(functionalTestTask)
}

gradlePlugin {
    testSourceSets(functionalTest)
}

dependencies {
    "functionalTestImplementation"("org.spockframework:spock-core:2.1-groovy-
3.0") {
        exclude(group = "org.codehaus.groovy")
    }
}
```

Controlling the build environment

The runner executes the test builds in an isolated environment by specifying a dedicated "working directory" in a directory inside the JVM's temp directory (i.e. the location specified by the `java.io.tmpdir` system property, typically `/tmp`). Any configuration in the default Gradle user home directory (e.g. `~/.gradle/gradle.properties`) is not used for test execution. The TestKit does not expose a mechanism for fine grained control of all aspects of the environment (e.g., JDK). Future versions of the TestKit will provide improved configuration options.

The TestKit uses dedicated daemon processes that are automatically shut down after test execution.

The dedicated working directory is not deleted by the runner after the build. The TestKit provides two ways to specify a location that is regularly cleaned, such as the project's build folder:

- The `org.gradle.testkit.dir` system property;
- The GradleRunner.withTestKitDir(file testKitDir) method.

The Gradle version used to test

The Gradle runner requires a Gradle distribution in order to execute the build. The TestKit does not depend on all of Gradle's implementation.

By default, the runner will attempt to find a Gradle distribution based on where the `GradleRunner` class was loaded from. That is, it is expected that the class was loaded from a Gradle distribution, as is the case when using the `gradleTestKit()` dependency declaration.

When using the runner as part of tests *being executed by Gradle* (e.g. executing the `test` task of a plugin project), the same distribution used to execute the tests will be used by the runner. When using the runner as part of tests *being executed by an IDE*, the same distribution of Gradle that was used when importing the project will be used. This means that the plugin will effectively be tested with the same version of Gradle that it is being built with.

Alternatively, a different and specific version of Gradle to use can be specified by the any of the following `GradleRunner` methods:

- GradleRunner.withGradleVersion(java.lang.String)
- GradleRunner.withGradleInstallation(java.io.File)
- GradleRunner.withGradleDistribution(java.net.URI)

This can potentially be used to test build logic across Gradle versions. The following demonstrates a cross-version compatibility test written as Groovy Spock test:

Example: Specifying a Gradle version for test execution

BuildLogicFunctionalTest.groovy

```
import org.gradle.testkit.runner.GradleRunner
import static org.gradle.testkit.runner.TaskOutcome.*
import spock.lang.TempDir
import spock.lang.Specification

class BuildLogicFunctionalTest extends Specification {
    @TempDir File testProjectDir
    File settingsFile
    File buildFile

    def setup() {
        settingsFile = new File(testProjectDir, 'settings.gradle')
        buildFile = new File(testProjectDir, 'build.gradle')
    }

    def "can execute hello world task with Gradle version #gradleVersion"() {
        given:
        buildFile << """
            task helloWorld {
                doLast {
                    logger.quiet 'Hello world!'
                }
            }
        """
        settingsFile << ""

        when:
        def result = GradleRunner.create()
            .withGradleVersion(gradleVersion)
            .withProjectDir(testProjectDir)
            .withArguments('helloWorld')
            .build()

        then:
        result.output.contains('Hello world!')
        result.task(":helloWorld").outcome == SUCCESS

        where:
        gradleVersion << ['5.0', '6.0.1']
    }
}
```

Feature support when testing with different Gradle versions

It is possible to use the GradleRunner to execute builds with Gradle 1.0 and later. However, some runner features are not supported on earlier versions. In such cases, the runner will throw an exception when attempting to use the feature.

The following table lists the features that are sensitive to the Gradle version being used.

Table 8. Gradle version compatibility

Feature	Minimum Version	Description
Inspecting executed tasks	2.5	Inspecting the executed tasks, using BuildResult.getTasks() and similar methods.
Plugin classpath injection	2.8	Injecting the code under test viaGradleRunner.withPluginClasspath(java.lang.Iterable).
Inspecting build output in debug mode	2.9	Inspecting the build's text output when run in debug mode, using BuildResult.getOutput().
Automatic plugin classpath injection	2.13	Injecting the code under test automatically via GradleRunner.withPluginClasspath() by applying the Java Gradle Plugin Development plugin.
Setting environment variables to be used by the build.	3.5	The Gradle Tooling API only supports setting environment variables in later versions.

Debugging build logic

The runner uses the Tooling API to execute builds. An implication of this is that the builds are executed in a separate process (i.e. not the same process executing the tests). Therefore, executing your *tests* in debug mode does not allow you to debug your build logic as you may expect. Any breakpoints set in your IDE will be not be tripped by the code being exercised by the test build.

The TestKit provides two different ways to enable the debug mode:

- Setting "`org.gradle.testkit.debug`" system property to `true` for the JVM *using* the `GradleRunner` (i.e. not the build being executed with the runner);
- Calling the GradleRunner.withDebug(boolean) method.

The system property approach can be used when it is desirable to enable debugging support without making an adhoc change to the runner configuration. Most IDEs offer the capability to set JVM system properties for test execution, and such a feature can be used to set this system property.

Testing with the Build Cache

To enable the Build Cache in your tests, you can pass the `--build-cache` argument to GradleRunner or use one of the other methods described in Enable the build cache. You can then check for the task outcome TaskOutcome.FROM_CACHE when your plugin's custom task is cached. This outcome is only valid for Gradle 3.5 and newer.

Example: Testing cacheable tasks

BuildLogicFunctionalTest.groovy

```
def "cacheableTask is loaded from cache"() {
    given:
    buildFile << """
        plugins {
            id 'org.gradle.sample.helloworld'
        }
    """

    when:
    def result = runner()
        .withArguments( '--build-cache', 'cacheableTask')
        .build()

    then:
    result.task(":cacheableTask").outcome == SUCCESS

    when:
    new File(testProjectDir, 'build').deleteDir()
    result = runner()
        .withArguments( '--build-cache', 'cacheableTask')
        .build()

    then:
    result.task(":cacheableTask").outcome == FROM_CACHE
}
```

Note that TestKit re-uses a Gradle user home between tests (see GradleRunner.withTestKitDir(java.io.File)) which contains the default location for the local build cache. For testing with the build cache, the build cache directory should be cleaned between tests. The easiest way to accomplish this is to configure the local build cache to use a temporary directory.

Example: Clean build cache between tests

BuildLogicFunctionalTest.groovy

```
@TempDir File testProjectDir
File buildFile
File localBuildCacheDirectory

def setup() {
    localBuildCacheDirectory = new File(testProjectDir, 'local-cache')
    buildFile = new File(testProjectDir,'settings.gradle') << """
        buildCache {
            local {
                directory '${localBuildCacheDirectory.toURI()}'
            }
        }
    """
    buildFile = new File(testProjectDir,'build.gradle')
}
```

Using Ant from Gradle

Gradle provides excellent integration with Ant. You can use individual Ant tasks or entire Ant builds in your Gradle builds. In fact, you will find that it's far easier and more powerful using Ant tasks in a Gradle build script, than it is to use Ant's XML format. You could even use Gradle simply as a powerful Ant task scripting tool.

Ant can be divided into two layers. The first layer is the Ant language. It provides the syntax for the `build.xml` file, the handling of the targets, special constructs like macrodefs, and so on. In other words, everything except the Ant tasks and types. Gradle understands this language, and allows you to import your Ant `build.xml` directly into a Gradle project. You can then use the targets of your Ant build as if they were Gradle tasks.

The second layer of Ant is its wealth of Ant tasks and types, like `javac`, `copy` or `jar`. For this layer Gradle provides integration simply by relying on Groovy, and the fantastic `AntBuilder`.

Finally, since build scripts are Groovy scripts, you can always execute an Ant build as an external process. Your build script may contain statements like: `"ant clean compile".execute()`.[6]

You can use Gradle's Ant integration as a path for migrating your build from Ant to Gradle. For example, you could start by importing your existing Ant build. Then you could move your dependency declarations from the Ant script to your build file. Finally, you could move your tasks across to your build file, or replace them with some of Gradle's plugins. This process can be done in parts over time, and you can have a working Gradle build during the entire process.

Using Ant tasks and types in your build

In your build script, a property called `ant` is provided by Gradle. This is a reference to an AntBuilder instance. This `AntBuilder` is used to access Ant tasks, types and properties from your build script. There is a very simple mapping from Ant's `build.xml` format to Groovy, which is explained below.

You execute an Ant task by calling a method on the `AntBuilder` instance. You use the task name as the method name. For example, you execute the Ant `echo` task by calling the `ant.echo()` method. The attributes of the Ant task are passed as Map parameters to the method. Below is an example of the `echo` task. Notice that we can also mix Groovy code and the Ant task markup. This can be extremely powerful.

Example 233. Using an Ant task

build.gradle

```
tasks.register('hello') {
    doLast {
        String greeting = 'hello from Ant'
        ant.echo(message: greeting)
    }
}
```

build.gradle.kts

```
tasks.register("hello") {
    doLast {
        val greeting = "hello from Ant"
        ant.withGroovyBuilder {
            "echo"("message" to greeting)
        }
    }
}
```

Output of `gradle hello`

```
> gradle hello

> Task :hello
[ant:echo] hello from Ant

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

You pass nested text to an Ant task by passing it as a parameter of the task method call. In this example, we pass the message for the `echo` task as nested text:

Example 234. Passing nested text to an Ant task

build.gradle

```
tasks.register('hello') {
    doLast {
        ant.echo('hello from Ant')
    }
}
```

build.gradle.kts

```
tasks.register("hello") {
    doLast {
        ant.withGroovyBuilder {
            "echo"("message" to "hello from Ant")
        }
    }
}
```

Output of `gradle hello`

```
> gradle hello

> Task :hello
[ant:echo] hello from Ant

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

You pass nested elements to an Ant task inside a closure. Nested elements are defined in the same way as tasks, by calling a method with the same name as the element we want to define.

Example 235. Passing nested elements to an Ant task

build.gradle

```
tasks.register('zip') {
    doLast {
        ant.zip(destfile: 'archive.zip') {
            fileset(dir: 'src') {
                include(name: '**.xml')
                exclude(name: '**.java')
            }
        }
    }
}
```

build.gradle.kts

```
tasks.register("zip") {
    doLast {
        ant.withGroovyBuilder {
            "zip"("destfile" to "archive.zip") {
                "fileset"("dir" to "src") {
                    "include"("name" to "**.xml")
                    "exclude"("name" to "**.java")
                }
            }
        }
    }
}
```

You can access Ant types in the same way that you access tasks, using the name of the type as the method name. The method call returns the Ant data type, which you can then use directly in your build script. In the following example, we create an Ant `path` object, then iterate over the contents of it.

Example 236. Using an Ant type

build.gradle

```
tasks.register('list') {
    doLast {
        def path = ant.path {
            fileset(dir: 'libs', includes: '*.jar')
        }
        path.list().each {
            println it
        }
    }
}
```

build.gradle.kts

```
import org.apache.tools.ant.types.Path

tasks.register("list") {
    doLast {
        val path = ant.withGroovyBuilder {
            "path" {
                "fileset"("dir" to "libs", "includes" to "*.jar")
            }
        } as Path
        path.list().forEach {
            println(it)
        }
    }
}
```

More information about `AntBuilder` can be found in 'Groovy in Action' 8.4 or at the Groovy Wiki.

Using custom Ant tasks in your build

To make custom tasks available in your build, you can use the `taskdef` (usually easier) or `typedef` Ant task, just as you would in a `build.xml` file. You can then refer to the custom Ant task as you would a built-in Ant task.

Example 237. Using a custom Ant task

build.gradle

```
tasks.register('check') {
    doLast {
        ant.taskdef(resource: 'checkstyletask.properties') {
            classpath {
                fileset(dir: 'libs', includes: '*.jar')
            }
        }
        ant.checkstyle(config: 'checkstyle.xml') {
            fileset(dir: 'src')
        }
    }
}
```

build.gradle.kts

```
tasks.register("check") {
    doLast {
        ant.withGroovyBuilder {
            "taskdef"("resource" to "checkstyletask.properties") {
                "classpath" {
                    "fileset"("dir" to "libs", "includes" to "*.jar")
                }
            }
            "checkstyle"("config" to "checkstyle.xml") {
                "fileset"("dir" to "src")
            }
        }
    }
}
```

You can use Gradle's dependency management to assemble the classpath to use for the custom tasks. To do this, you need to define a custom configuration for the classpath, then add some dependencies to the configuration. This is described in more detail in Declaring Dependencies.

Example 238. Declaring the classpath for a custom Ant task

build.gradle

```
configurations {
    pmd
}

dependencies {
    pmd group: 'pmd', name: 'pmd', version: '4.2.5'
}
```

build.gradle.kts

```
val pmd = configurations.create("pmd")

dependencies {
    pmd(group = "pmd", name = "pmd", version = "4.2.5")
}
```

To use the classpath configuration, use the `asPath` property of the custom configuration.

Example 239. Using a custom Ant task and dependency management together

build.gradle

```
tasks.register('check') {
    doLast {
        ant.taskdef(name: 'pmd',
                    classname: 'net.sourceforge.pmd.ant.PMDTask',
                    classpath: configurations.pmd.asPath)
        ant.pmd(shortFilenames: 'true',
                failonruleviolation: 'true',
                rulesetfiles: file('pmd-rules.xml').toURI().toString()) {
            formatter(type: 'text', toConsole: 'true')
            fileset(dir: 'src')
        }
    }
}
```

build.gradle.kts

```
tasks.register("check") {
    doLast {
        ant.withGroovyBuilder {
            "taskdef"("name" to "pmd",
                      "classname" to "net.sourceforge.pmd.ant.PMDTask",
                      "classpath" to pmd.asPath)
            "pmd"("shortFilenames" to true,
                  "failonruleviolation" to true,
                  "rulesetfiles" to file("pmd-rules.xml").toURI().toString())
{
                "formatter"("type" to "text", "toConsole" to "true")
                "fileset"("dir" to "src")
            }
        }
    }
}
```

Importing an Ant build

You can use the `ant.importBuild()` method to import an Ant build into your Gradle project. When you import an Ant build, each Ant target is treated as a Gradle task. This means you can manipulate and execute the Ant targets in exactly the same way as Gradle tasks.

Example 240. Importing an Ant build

build.gradle

```
ant.importBuild 'build.xml'
```

build.gradle.kts

```
ant.importBuild("build.xml")
```

build.xml

```
<project>
    <target name="hello">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

Output of `gradle hello`

```
> gradle hello

> Task :hello
[ant:echo] Hello, from Ant

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

You can add a task which depends on an Ant target:

Example 241. Task that depends on Ant target

build.gradle

```
ant.importBuild 'build.xml'

tasks.register('intro') {
    dependsOn("hello")
    doLast {
        println 'Hello, from Gradle'
    }
}
```

build.gradle.kts

```
ant.importBuild("build.xml")

tasks.register("intro") {
    dependsOn("hello")
    doLast {
        println("Hello, from Gradle")
    }
}
```

Output of `gradle intro`

```
> gradle intro

> Task :hello
[ant:echo] Hello, from Ant

> Task :intro
Hello, from Gradle

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Or, you can add behaviour to an Ant target:

Example 242. Adding behaviour to an Ant target

build.gradle

```
ant.importBuild 'build.xml'

hello {
    doLast {
        println 'Hello, from Gradle'
    }
}
```

build.gradle.kts

```
ant.importBuild("build.xml")

tasks.named("hello") {
    doLast {
        println("Hello, from Gradle")
    }
}
```

Output of `gradle hello`

```
> gradle hello

> Task :hello
[ant:echo] Hello, from Ant
Hello, from Gradle

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

It is also possible for an Ant target to depend on a Gradle task:

Example 243. Ant target that depends on Gradle task

build.gradle

```
ant.importBuild 'build.xml'

tasks.register('intro') {
    doLast {
        println 'Hello, from Gradle'
    }
}
```

build.gradle.kts

```
ant.importBuild("build.xml")

tasks.register("intro") {
    doLast {
        println("Hello, from Gradle")
    }
}
```

build.xml

```
<project>
    <target name="hello" depends="intro">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

Output of `gradle hello`

```
> gradle hello

> Task :intro
Hello, from Gradle

> Task :hello
[ant:echo] Hello, from Ant

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Sometimes it may be necessary to “rename” the task generated for an Ant target to avoid a naming collision with existing Gradle tasks. To do this, use the AntBuilder.importBuild(java.lang.Object, org.gradle.api.Transformer) method.

Example 244. Renaming imported Ant targets

build.gradle

```gradle
ant.importBuild('build.xml') { antTargetName ->
    'a-' + antTargetName
}
```

build.gradle.kts

```gradle-kotlin-dsl
ant.importBuild("build.xml") { antTargetName ->
    "a-" + antTargetName
}
```

build.xml

```
<project>
    <target name="hello">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

Output of `gradle a-hello`

```
> gradle a-hello

> Task :a-hello
[ant:echo] Hello, from Ant

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Note that while the second argument to this method should be a Transformer, when programming in Groovy we can simply use a closure instead of an anonymous inner class (or similar) due to Groovy's support for automatically coercing closures to single-abstract-method types.

Ant properties and references

There are several ways to set an Ant property, so that the property can be used by Ant tasks. You can set the property directly on the `AntBuilder` instance. The Ant properties are also available as a Map which you can change. You can also use the Ant `property` task. Below are some examples of how to do this.

Example 245. Setting an Ant property

build.gradle

```
ant.buildDir = buildDir
ant.properties.buildDir = buildDir
ant.properties['buildDir'] = buildDir
ant.property(name: 'buildDir', location: buildDir)
```

build.gradle.kts

```
ant.setProperty("buildDir", buildDir)
ant.properties.set("buildDir", buildDir)
ant.properties["buildDir"] = buildDir
ant.withGroovyBuilder {
    "property"("name" to "buildDir", "location" to "buildDir")
}
```

Many Ant tasks set properties when they execute. There are several ways to get the value of these properties. You can get the property directly from the `AntBuilder` instance. The Ant properties are also available as a Map. Below are some examples.

Example 246. Getting an Ant property

build.xml

```
<property name="antProp" value="a property defined in an Ant build"/>
```

build.gradle

```
println ant.antProp
println ant.properties.antProp
println ant.properties['antProp']
```

build.gradle.kts

```
println(ant.getProperty("antProp"))
println(ant.properties.get("antProp"))
println(ant.properties["antProp"])
```

There are several ways to set an Ant reference:

Example 247. Setting an Ant reference

build.gradle

```
ant.path(id: 'classpath', location: 'libs')
ant.references.classpath = ant.path(location: 'libs')
ant.references['classpath'] = ant.path(location: 'libs')
```

build.gradle.kts

```
ant.withGroovyBuilder { "path"("id" to "classpath", "location" to "libs") }
ant.references.set("classpath", ant.withGroovyBuilder { "path"("location" to
"libs") })
ant.references["classpath"] = ant.withGroovyBuilder { "path"("location" to
"libs") }
```

build.xml

```
<path refid="classpath"/>
```

There are several ways to get an Ant reference:

Example 248. Getting an Ant reference

build.xml

```
<path id="antPath" location="libs"/>
```

build.gradle

```
println ant.references.antPath
println ant.references['antPath']
```

build.gradle.kts

```
println(ant.references.get("antPath"))
println(ant.references["antPath"])
```

Ant logging

Gradle maps Ant message priorities to Gradle log levels so that messages logged from Ant appear in the Gradle output. By default, these are mapped as follows:

Table 9. Ant message priority mapping

Ant Message Priority	Gradle Log Level
VERBOSE	`DEBUG`
DEBUG	`DEBUG`
INFO	`INFO`

Ant Message Priority	Gradle Log Level
WARN	`WARN`
ERROR	`ERROR`

Fine tuning Ant logging

The default mapping of Ant message priority to Gradle log level can sometimes be problematic. For example, there is no message priority that maps directly to the `LIFECYCLE` log level, which is the default for Gradle. Many Ant tasks log messages at the *INFO* priority, which means to expose those messages from Gradle, a build would have to be run with the log level set to `INFO`, potentially logging much more output than is desired.

Conversely, if an Ant task logs messages at too high of a level, to suppress those messages would require the build to be run at a higher log level, such as `QUIET`. However, this could result in other, desirable output being suppressed.

To help with this, Gradle allows the user to fine tune the Ant logging and control the mapping of message priority to Gradle log level. This is done by setting the priority that should map to the default Gradle `LIFECYCLE` log level using the AntBuilder.setLifecycleLogLevel(java.lang.String) method. When this value is set, any Ant message logged at the configured priority or above will be logged at least at `LIFECYCLE`. Any Ant message logged below this priority will be logged at most at `INFO`.

For example, the following changes the mapping such that Ant *INFO* priority messages are exposed at the `LIFECYCLE` log level.

Example 249. Fine tuning Ant logging

build.gradle

```
ant.lifecycleLogLevel = "INFO"

tasks.register('hello') {
    doLast {
        ant.echo(level: "info", message: "hello from info priority!")
    }
}
```

build.gradle.kts

```
ant.lifecycleLogLevel = AntBuilder.AntMessagePriority.INFO

tasks.register("hello") {
    doLast {
        ant.withGroovyBuilder {
            "echo"("level" to "info", "message" to "hello from info
priority!")
        }
    }
}
```

Output of `gradle hello`

```
> gradle hello

> Task :hello
[ant:echo] hello from info priority!

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

On the other hand, if the `lifecycleLogLevel` was set to *ERROR*, Ant messages logged at the *WARN* priority would no longer be logged at the `WARN` log level. They would now be logged at the `INFO` level and would be suppressed by default.

API

The Ant integration is provided by AntBuilder.

[1] There are command line switches to change this behavior. See Command-Line Interface

[2] There are command line switches to change this behavior. See Command-Line Interface

[3] You might be wondering why there is neither an import for the `StopExecutionException` nor do we access it via its fully qualified name. The reason is, that Gradle adds a set of default imports to your script (see Default imports).

[4] Any language element except for statement labels.

[5] Gradle supports partial multi-project builds (see Executing Multi-Project Builds).

[6] In Groovy you can execute Strings. To learn more about executing external processes with Groovy have a look in 'Groovy in Action' 9.3.2 or at the Groovy wiki

Dependency Management

Learning the Basics

Dependency management in Gradle

What is dependency management?

Software projects rarely work in isolation. In most cases, a project relies on reusable functionality in the form of libraries or is broken up into individual components to compose a modularized system. Dependency management is a technique for declaring, resolving and using dependencies required by the project in an automated fashion.

NOTE	For a general overview on the terms used throughout the user guide, refer to Dependency Management Terminology.

Dependency management in Gradle

Gradle has built-in support for dependency management and lives up to the task of fulfilling typical scenarios encountered in modern software projects. We'll explore the main concepts with the help of an example project. The illustration below should give you an rough overview on all the moving parts.

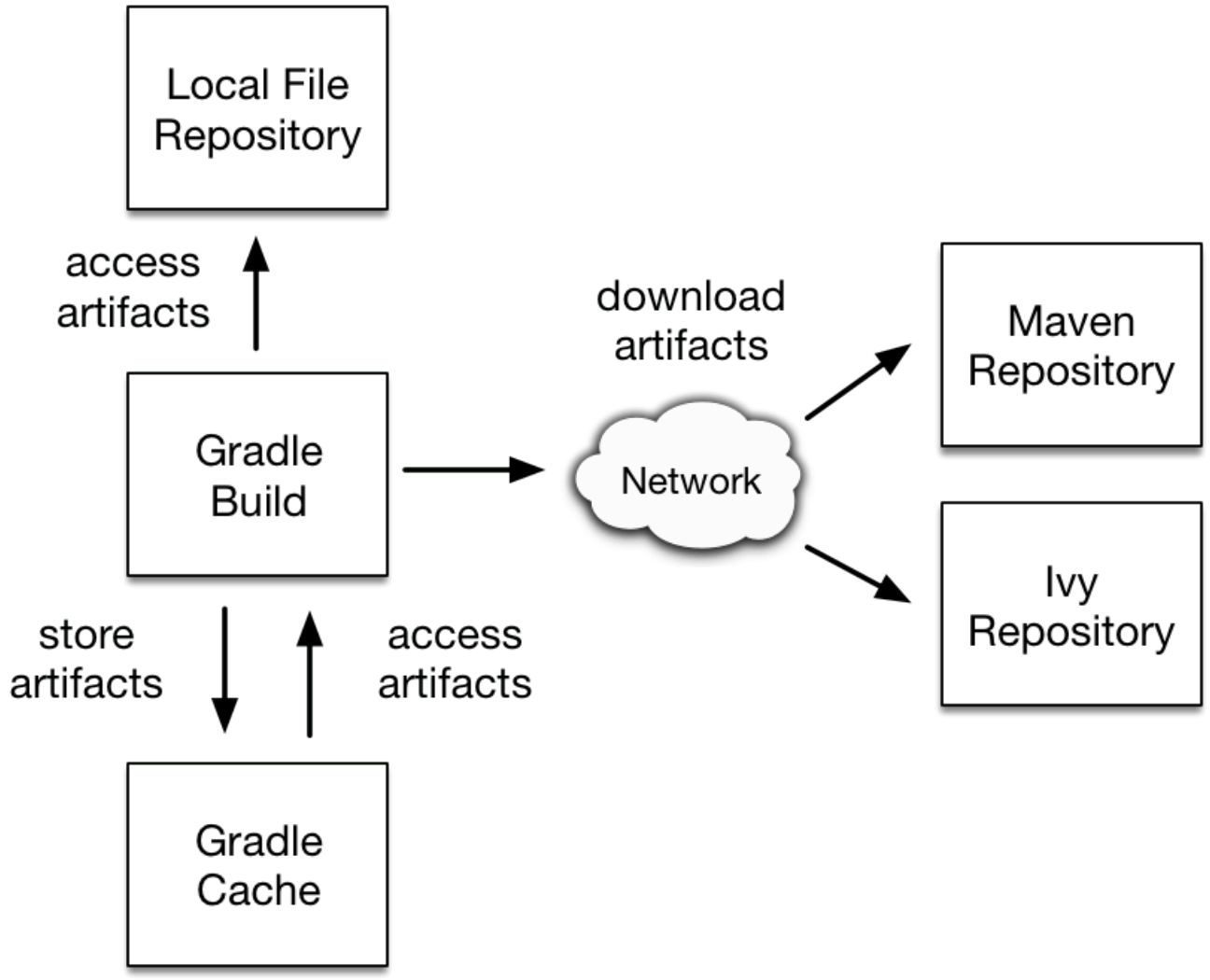


Figure 12. Dependency management big picture

The example project builds Java source code. Some of the Java source files import classes from Google Guava, a open-source library providing a wealth of utility functionality. In addition to Guava, the project needs the JUnit libraries for compiling and executing test code.

Guava and JUnit represent the *dependencies* of this project. A build script developer can declare dependencies for different scopes e.g. just for compilation of source code or for executing tests. In Gradle, the scope of a dependency is called a *configuration*. For a full overview, see the reference material on dependency types.

Often times dependencies come in the form of modules. You'll need to tell Gradle where to find those modules so they can be consumed by the build. The location for storing modules is called a *repository*. By declaring repositories for a build, Gradle will know how to find and retrieve modules. Repositories can come in different forms: as local directory or a remote repository. The reference on repository types provides a broad coverage on this topic.

At runtime, Gradle will locate the declared dependencies if needed for operating a specific task. The dependencies might need to be downloaded from a remote repository, retrieved from a local directory or requires another project to be built in a multi-project setting. This process is called *dependency resolution*. You can find a detailed discussion in How Gradle downloads dependencies.

Once resolved, the resolution mechanism stores the underlying files of a dependency in a local

cache, also referred to as the *dependency cache*. Future builds reuse the files stored in the cache to avoid unnecessary network calls.

Modules can provide additional metadata. Metadata is the data that describes the module in more detail e.g. the coordinates for finding it in a repository, information about the project, or its authors. As part of the metadata, a module can define that other modules are needed for it to work properly. For example, the JUnit 5 platform module also requires the platform commons module. Gradle automatically resolves those additional modules, so called *transitive dependencies*. If needed, you can customize the behavior the handling of transitive dependencies to your project's requirements.

Projects with tens or hundreds of declared dependencies can easily suffer from dependency hell. Gradle provides sufficient tooling to visualize, navigate and analyze the dependency graph of a project either with the help of a build scan or built-in tasks. Learn more in Viewing and debugging dependencies.

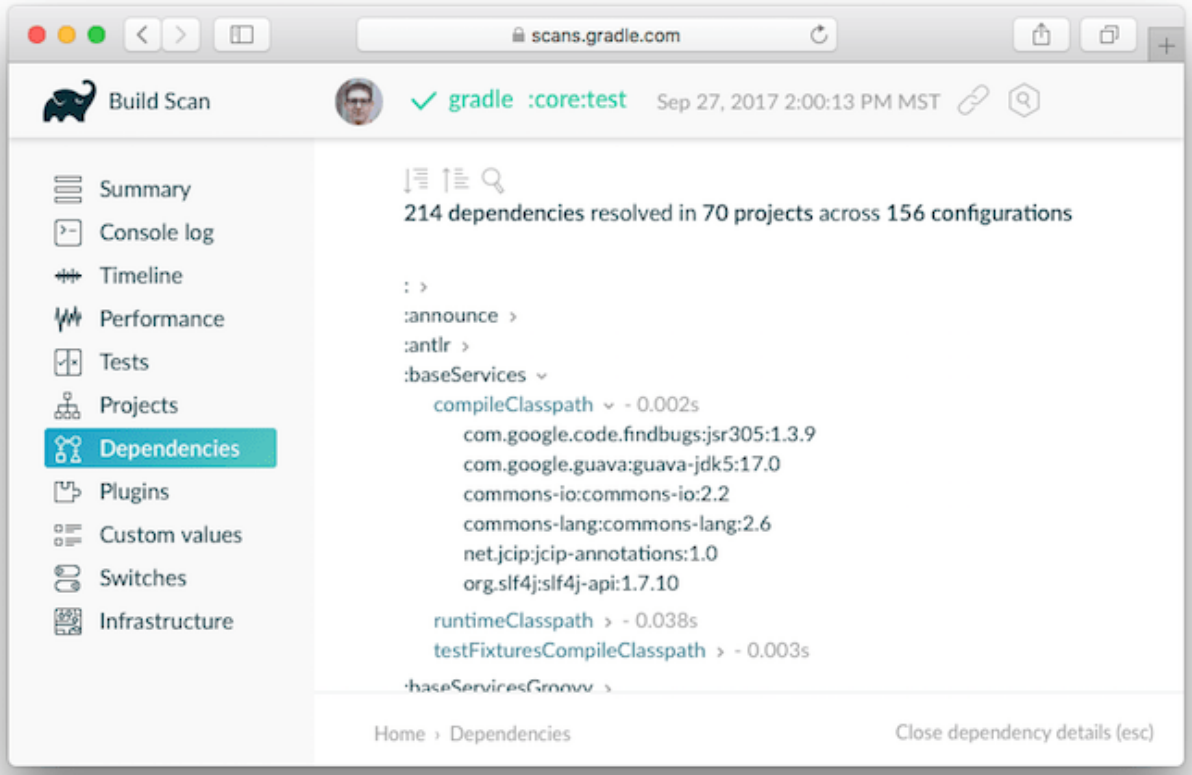


Figure 13. Build scan dependencies report

Declaring repositories

Gradle can resolve dependencies from one or many repositories based on Maven, Ivy or flat directory formats. Check out the full reference on all types of repositories for more information.

Declaring a publicly-available repository

Organizations building software may want to leverage public binary repositories to download and consume open source dependencies. Popular public repositories include Maven Central and the Google Android repository. Gradle provides built-in shorthand notations for these widely-used repositories.

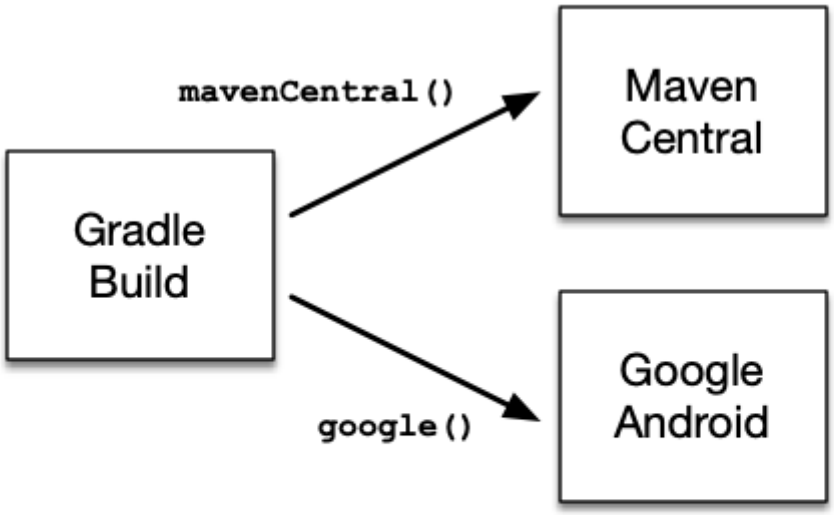


Figure 14. Declaring a repository with the help of shorthand notations

Under the covers Gradle resolves dependencies from the respective URL of the public repository defined by the shorthand notation. All shorthand notations are available via the RepositoryHandler API. Alternatively, you can spell out the URL of the repository for more fine-grained control.

Maven Central repository

Maven Central is a popular repository hosting open source libraries for consumption by Java projects.

To declare the Maven Central repository for your build add this to your script:

Example 250. Adding central Maven repository

build.gradle

```
repositories {
    mavenCentral()
}
```

build.gradle.kts

```
repositories {
    mavenCentral()
}
```

Google Maven repository

The Google repository hosts Android-specific artifacts including the Android SDK. For usage examples, see the relevant Android documentation.

To declare the Google Maven repository add this to your build script:

Example 251. Adding Google Maven repository

build.gradle

```
repositories {
    google()
}
```

build.gradle.kts

```
repositories {
    google()
}
```

Declaring a custom repository by URL

Most enterprise projects set up a binary repository available only within an intranet. In-house repositories enable teams to publish internal binaries, setup user management and security measure and ensure uptime and availability. Specifying a custom URL is also helpful if you want to declare a less popular, but publicly-available repository.

Repositories with custom URLs can be specified as Maven or Ivy repositories by calling the corresponding methods available on the RepositoryHandler API. Gradle supports other protocols than `http` or `https` as part of the custom URL e.g. `file`, `sftp` or `s3`. For a full coverage see the section on supported repository types.

You can also define your own repository layout by using `ivy { }` repositories as they are very flexible in terms of how modules are organised in a repository.

Declaring multiple repositories

You can define more than one repository for resolving dependencies. Declaring multiple repositories is helpful if some dependencies are only available in one repository but not the other. You can mix any type of repository described in the reference section.

This example demonstrates how to declare various named and custom URL repositories for a project:

Example 252. Declaring multiple repositories

build.gradle

```
repositories {
    mavenCentral()
    maven {
        url "https://repo.spring.io/release"
    }
    maven {
        url "https://repository.jboss.org/maven2"
    }
}
```

build.gradle.kts

```
repositories {
    mavenCentral()
    maven {
        url = uri("https://repo.spring.io/release")
    }
    maven {
        url = uri("https://repository.jboss.org/maven2")
    }
}
```

NOTE The order of declaration determines how Gradle will check for dependencies at runtime. If Gradle finds a module descriptor in a particular repository, it will attempt to download all of the artifacts for that module from *the same repository*. You can learn more about the inner workings of dependency downloads.

Strict limitation to declared repositories

Maven POM metadata can reference additional repositories. These will be *ignored* by Gradle, which will only use the repositories declared in the build itself.

NOTE This is a reproducibility safe-guard but also a security protection. Without it, an updated version of a dependency could pull artifacts from anywhere into your build.

Supported repository types

Gradle supports a wide range of sources for dependencies, both in terms of format and in terms of connectivity. You may resolve dependencies from:

- Different formats
  - a Maven compatible artifact repository (e.g: Maven Central)
  - an Ivy compatible artifact repository (including custom layouts)
  - local (flat) directories
- with different connectivity
  - authenticated repositories
  - a wide variety of remote protocols such as HTTPS, SFTP, AWS S3 and Google Cloud Storage

Flat directory repository

Some projects might prefer to store dependencies on a shared drive or as part of the project source code instead of a binary repository product. If you want to use a (flat) filesystem directory as a repository, simply type:

Example 253. Flat repository resolver

build.gradle

```
repositories {
    flatDir {
        dirs 'lib'
    }
    flatDir {
        dirs 'lib1', 'lib2'
    }
}
```

build.gradle.kts

```
repositories {
    flatDir {
        dirs("lib")
    }
    flatDir {
        dirs("lib1", "lib2")
    }
}
```

This adds repositories which look into one or more directories for finding dependencies.

This type of repository does not support any meta-data formats like Ivy XML or Maven POM files. Instead, Gradle will dynamically generate a module descriptor (without any dependency information) based on the presence of artifacts.

NOTE

As Gradle prefers to use modules whose descriptor has been created from real meta-data rather than being generated, flat directory repositories cannot be used to override artifacts with real meta-data from other repositories declared in the build.

For example, if Gradle finds only `jmxri-1.2.1.jar` in a flat directory repository, but `jmxri-1.2.1.pom` in another repository that supports meta-data, it will use the second repository to provide the module.

For the use case of overriding remote artifacts with local ones consider using an Ivy or Maven repository instead whose URL points to a local directory.

If you only work with flat directory repositories you don't need to set all attributes of a dependency.

Local repositories

The following sections describe repositories format, Maven or Ivy. These can be declared as local repositories, using a local filesystem path to access them.

The difference with the flat directory repository is that they do respect a format and contain metadata.

When such a repository is configured, Gradle totally bypasses its dependency cache for it as there can be no guarantee that content may not change between executions. Because of that limitation, they can have a performance impact.

They also make build reproducibility much harder to achieve and their use should be limited to tinkering or prototyping.

Maven repositories

Many organizations host dependencies in an in-house Maven repository only accessible within the company's network. Gradle can declare Maven repositories by URL.

For adding a custom Maven repository you can do:

Example 254. Adding custom Maven repository

build.gradle

```
repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("http://repo.mycompany.com/maven2")
    }
}
```

Setting up composite Maven repositories

Sometimes a repository will have the POMs published to one location, and the JARs and other artifacts published at another location. To define such a repository, you can do:

Example 255. Adding additional Maven repositories for JAR files

build.gradle

```
repositories {
    maven {
        // Look for POMs and artifacts, such as JARs, here
        url "http://repo2.mycompany.com/maven2"
        // Look for artifacts here if not found at the above location
        artifactUrls "http://repo.mycompany.com/jars"
        artifactUrls "http://repo.mycompany.com/jars2"
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        // Look for POMs and artifacts, such as JARs, here
        url = uri("http://repo2.mycompany.com/maven2")
        // Look for artifacts here if not found at the above location
        artifactUrls("http://repo.mycompany.com/jars")
        artifactUrls("http://repo.mycompany.com/jars2")
    }
}
```

Gradle will look at the base `url` location for the POM and the JAR. If the JAR can't be found there, the extra `artifactUrls` are used to look for JARs.

Accessing authenticated Maven repositories

You can specify credentials for Maven repositories secured by different type of authentication.

See Supported repository transport protocols for authentication options.

Local Maven repository

Gradle can consume dependencies available in the local Maven repository. Declaring this repository is beneficial for teams that publish to the local Maven repository with one project and consume the artifacts by Gradle in another project.

NOTE	Gradle stores resolved dependencies in its own cache. A build does not need to declare the local Maven repository even if you resolve dependencies from a Maven-based, remote repository.

WARNING	Before adding Maven local as a repository, you should make sure this is really required.

To declare the local Maven cache as a repository add this to your build script:

Example 256. Adding the local Maven cache as a repository

build.gradle

```
repositories {
    mavenLocal()
}
```

build.gradle.kts

```
repositories {
    mavenLocal()
}
```

Gradle uses the same logic as Maven to identify the location of your local Maven cache. If a local repository location is defined in a `settings.xml`, this location will be used. The `settings.xml` in `USER_HOME/.m2` takes precedence over the `settings.xml` in `M2_HOME/conf`. If no `settings.xml` is available, Gradle uses the default location `USER_HOME/.m2/repository`.

The case for mavenLocal()

As a general advice, you should avoid adding `mavenLocal()` as a repository. There are different issues with using `mavenLocal()` that you should be aware of:

- Maven uses it as a cache, not a repository, meaning it can contain partial modules.
  - For example, if Maven never downloaded the source or javadoc files for a given module, Gradle will not find them either since it searches for files in a single repository once a module has been found.
- As a local repository, Gradle does not trust its content, because:
  - Origin of artifacts cannot be tracked, which is a correctness and security problem
  - Artifacts can be easily overwritten, which is a security, correctness and reproducibility problem
- To mitigate the fact that metadata and/or artifacts can be changed, Gradle does not perform any caching for local repositories
  - As a consequence, your builds are slower
  - Given that order of repositories is important, adding `mavenLocal()` *first* means that all your

builds are going to be slower

There are a few cases where you might have to use `mavenLocal()`:

- For interoperability with Maven
  - For example, project A is built with Maven, project B is built with Gradle, and you need to share the artifacts during development
  - It is *always* preferable to use an internal full featured repository instead
  - In case this is not possible, you should limit this to *local builds only*
- For interoperability with Gradle itself
  - In a multi-repository world, you want to check that changes to project A work with project B
  - It is preferable to use composite builds for this use case
  - If for some reason neither composite builds nor full featured repository are possible, then `mavenLocal()` is a last resort option

After all these warnings, if you end up using `mavenLocal()`, consider combining it with a repository filter. This will make sure it only provides what is expected and nothing else.

Ivy repositories

Organizations might decide to host dependencies in an in-house Ivy repository. Gradle can declare Ivy repositories by URL.

Defining an Ivy repository with a standard layout

To declare an Ivy repository using the standard layout no additional customization is needed. You just declare the URL.

Example 257. Ivy repository

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
    }
}
```

build.gradle.kts

```
repositories {
    ivy {
        url = uri("http://repo.mycompany.com/repo")
    }
}
```

Defining a named layout for an Ivy repository

You can specify that your repository conforms to the Ivy or Maven default layout by using a named layout.

Example 258. Ivy repository with named layout

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        layout "maven"
    }
}
```

build.gradle.kts

```
repositories {
    ivy {
        url = uri("http://repo.mycompany.com/repo")
        layout("maven")
    }
}
```

Valid named layout values are `'gradle'` (the default), `'maven'` and `'ivy'`. See IvyArtifactRepository.layout(java.lang.String) in the API documentation for details of these named layouts.

Defining custom pattern layout for an Ivy repository

To define an Ivy repository with a non-standard layout, you can define a *pattern* layout for the repository:

Example 259. Ivy repository with pattern layout

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        patternLayout {
            artifact "[module]/[revision]/[type]/[artifact].[ext]"
        }
    }
}
```

build.gradle.kts

```
repositories {
    ivy {
        url = uri("http://repo.mycompany.com/repo")
        patternLayout {
            artifact("[module]/[revision]/[type]/[artifact].[ext]")
        }
    }
}
```

To define an Ivy repository which fetches Ivy files and artifacts from different locations, you can define separate patterns to use to locate the Ivy files and artifacts:

Each `artifact` or `ivy` specified for a repository adds an *additional* pattern to use. The patterns are used in the order that they are defined.

Example 260. Ivy repository with multiple custom patterns

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        patternLayout {
            artifact "3rd-party-
artifacts/[organisation]/[module]/[revision]/[artifact]-[revision].[ext]"
            artifact "company-
artifacts/[organisation]/[module]/[revision]/[artifact]-[revision].[ext]"
            ivy "ivy-files/[organisation]/[module]/[revision]/ivy.xml"
        }
    }
}
```

build.gradle.kts

```
repositories {
    ivy {
        url = uri("http://repo.mycompany.com/repo")
        patternLayout {
            artifact("3rd-party-
artifacts/[organisation]/[module]/[revision]/[artifact]-[revision].[ext]")
            artifact("company-
artifacts/[organisation]/[module]/[revision]/[artifact]-[revision].[ext]")
            ivy("ivy-files/[organisation]/[module]/[revision]/ivy.xml")
        }
    }
}
```

Optionally, a repository with pattern layout can have its `'organisation'` part laid out in Maven style, with forward slashes replacing dots as separators. For example, the organisation `my.company` would then be represented as `my/company`.

Example 261. Ivy repository with Maven compatible layout

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        patternLayout {
            artifact "[organisation]/[module]/[revision]/[artifact]-
[revision].[ext]"
            m2compatible = true
        }
    }
}
```

build.gradle.kts

```
repositories {
    ivy {
        url = uri("http://repo.mycompany.com/repo")
        patternLayout {
            artifact("[organisation]/[module]/[revision]/[artifact]-
[revision].[ext]")
            setM2compatible(true)
        }
    }
}
```

Accessing authenticated Ivy repositories

You can specify credentials for Ivy repositories secured by basic authentication.

Example 262. Ivy repository with authentication

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com"
        credentials {
            username "user"
            password "password"
        }
    }
}
```

build.gradle.kts

```
repositories {
    ivy {
        url = uri("http://repo.mycompany.com")
        credentials {
            username = "user"
            password = "password"
        }
    }
}
```

See Supported repository transport protocols for authentication options.

Repository content filtering

Gradle exposes an API to declare what a repository may or may not contain. There are different use cases for it:

- performance, when you know a dependency will never be found in a specific repository
- security, by avoiding leaking what dependencies are used in a private project
- reliability, when some repositories contain corrupted metadata or artifacts

It's even more important when considering that the declared order of repositories matter.

Declaring a repository filter

Example 263. Declaring repository contents

build.gradle

```
repositories {
    maven {
        url "https://repo.mycompany.com/maven2"
        content {
            // this repository *only* contains artifacts with group
"my.company"
            includeGroup "my.company"
        }
    }
    mavenCentral {
        content {
            // this repository contains everything BUT artifacts with group
starting with "my.company"
            excludeGroupByRegex "my\\.company.*"
        }
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("https://repo.mycompany.com/maven2")
        content {
            // this repository *only* contains artifacts with group
"my.company"
            includeGroup("my.company")
        }
    }
    mavenCentral {
        content {
            // this repository contains everything BUT artifacts with group
starting with "my.company"
            excludeGroupByRegex("my\\.company.*")
        }
    }
}
```

By default, repositories include everything and exclude nothing:

- If you declare an include, then it excludes everything *but* what is included.

- If you declare an exclude, then it includes everything *but* what is excluded.
- If you declare both includes and excludes, then it includes only what is explicitly included and not excluded.

It is possible to filter either by explicit *group*, *module* or *version*, either strictly or using regular expressions. When using a strict version, it is possible to use a version range, using the format supported by Gradle. In addition, there are filtering options by resolution context: configuration name or even configuration attributes. See RepositoryContentDescriptor for details.

Declaring content exclusively found in one repository

Filters declared using the repository-level content filter are not exclusive. This means that declaring that a repository *includes* an artifact doesn't mean that the other repositories can't have it either: you must declare what every repository contains in extension.

Alternatively, Gradle provides an API which lets you declare that a repository *exclusively includes* an artifact. If you do so:

- an artifact declared in a repository *can't* be found in any other
- exclusive repository content must be declared in extension (just like for repository-level content)

Example 264. Declaring exclusive repository contents

build.gradle

```
repositories {
    // This repository will _not_ be searched for artifacts in my.company
    // despite being declared first
    mavenCentral()
    exclusiveContent {
        forRepository {
            maven {
                url "https://repo.mycompany.com/maven2"
            }
        }
        filter {
            // this repository *only* contains artifacts with group
"my.company"
            includeGroup "my.company"
        }
    }
}
```

build.gradle.kts

```
repositories {
    // This repository will _not_ be searched for artifacts in my.company
    // despite being declared first
    mavenCentral()
    exclusiveContent {
        forRepository {
            maven {
                url = uri("https://repo.mycompany.com/maven2")
            }
        }
        filter {
            // this repository *only* contains artifacts with group
"my.company"
            includeGroup("my.company")
        }
    }
}
```

It is possible to filter either by explicit *group*, *module* or *version*, either strictly or using regular expressions. See InclusiveRepositoryContentDescriptor for details.

NOTE

If you leverage exclusive content filtering in the `pluginManagement` section of the `settings.gradle(.kts)`, it becomes illegal to add more repositories through the project `buildscript.repositories`. In that case, the build configuration will fail.

Your options are either to declare all repositories in settings or to use non-exclusive content filtering.

Maven repository filtering

For Maven repositories, it's often the case that a repository would either contain releases or snapshots. Gradle lets you declare what kind of artifacts are found in a repository using this DSL:

Example 265. Splitting snapshots and releases

build.gradle

```
repositories {
    maven {
        url "https://repo.mycompany.com/releases"
        mavenContent {
            releasesOnly()
        }
    }
    maven {
        url "https://repo.mycompany.com/snapshots"
        mavenContent {
            snapshotsOnly()
        }
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("https://repo.mycompany.com/releases")
        mavenContent {
            releasesOnly()
        }
    }
    maven {
        url = uri("https://repo.mycompany.com/snapshots")
        mavenContent {
            snapshotsOnly()
        }
    }
}
```

Supported metadata sources

When searching for a module in a repository, Gradle, by default, checks for supported metadata file formats in that repository. In a Maven repository, Gradle looks for a `.pom` file, in an ivy repository it looks for an `ivy.xml` file and in a flat directory repository it looks directly for `.jar` files as it does not expect any metadata. Starting with 5.0, Gradle also looks for `.module` (Gradle module metadata) files.

However, if you define a customized repository you might want to configure this behavior. For example, you can define a Maven repository without `.pom` files but only jars. To do so, you can

configure *metadata sources* for any repository.

Example 266. Maven repository that supports artifacts without metadata

build.gradle

```
repositories {
    maven {
        url "http://repo.mycompany.com/repo"
        metadataSources {
            mavenPom()
            artifact()
        }
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("http://repo.mycompany.com/repo")
        metadataSources {
            mavenPom()
            artifact()
        }
    }
}
```

You can specify multiple sources to tell Gradle to keep looking if a file was not found. In that case, the order of checking for sources is predefined.

The following metadata sources are supported:

Table 10. Supported metadata sources

Metadata source	Description	Order	Maven	Ivy / flat dir
`gradleMetadata()`	Look for Gradle `.module` files	1st	yes	yes
`mavenPom()`	Look for Maven `.pom` files	2nd	yes	yes
`ivyDescriptor()`	Look for `ivy.xml` files	2nd	no	yes
`artifact()`	Look directly for artifact	3rd	yes	yes

NOTE The defaults for Ivy and Maven repositories change with Gradle 6.0. Before 6.0, `artifact()` was included in the defaults. Leading to some inefficiency when modules are missing completely. To restore this behavior, for example, for Maven central you can use `mavenCentral { metadataSources { mavenPom(); artifact() } }`. In a similar way, you can opt into the new behavior in older Gradle versions using `mavenCentral { metadataSources { mavenPom() } }`

Since Gradle 5.3, when parsing a metadata file, be it Ivy or Maven, Gradle will look for a marker indicating that a matching Gradle Module Metadata files exists. If it is found, it will be used instead of the Ivy or Maven file.

Starting with Gradle 5.6, you can disable this behavior by adding `ignoreGradleMetadataRedirection()` to the metadataSources declaration.

Example 267. Maven repository that does not use gradle metadata redirection

build.gradle

```
repositories {
    maven {
        url "http://repo.mycompany.com/repo"
        metadataSources {
            mavenPom()
            artifact()
            ignoreGradleMetadataRedirection()
        }
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("http://repo.mycompany.com/repo")
        metadataSources {
            mavenPom()
            artifact()
            ignoreGradleMetadataRedirection()
        }
    }
}
```

Plugin repositories vs. build repositories

Gradle will use repositories at two different phases during your build.

The first phase is when configuring your build and loading the plugins it applied. To do that Gradle will use a special set of repositories.

The second phase is during dependency resolution. At this point Gradle will use the repositories declared in your project, as shown in the previous sections.

Plugin repositories

By default Gradle will use the Gradle plugin portal to look for plugins.

However, for different reasons, there are plugins available in other, public or not, repositories. When a build requires one of these plugins, additional repositories need to be specified so that Gradle knows where to search.

As the way to declare the repositories and what they are expected to contain depends on the way the plugin is applied, it is best to refer to Custom Plugin Repositories.

Centralizing repositories declaration

Instead of declaring repositories in every subproject of your build or via an `allprojects` block, Gradle offers a way to declare them in a central place for all project.

NOTE	Central declaration of repositories is an incubating feature

Repositories used by convention by every subproject can be declared in the `settings.gradle(.kts)` file:

Example 268. Declaring a Maven repository in settings

settings.gradle

```
dependencyResolutionManagement {
    repositories {
        mavenCentral()
    }
}
```

settings.gradle.kts

```
dependencyResolutionManagement {
    repositories {
        mavenCentral()
    }
}
```

The `dependencyResolutionManagement` repositories block accepts the same notations as in a project, which includes Maven or Ivy repositories, with or without credentials, etc.

By default, repositories declared by a project will **override** whatever is declared in settings. You can change this behavior to make sure that you always use the settings repositories:

Example 269. Preferring settings repositories

settings.gradle

```
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.PREFER_SETTINGS)
}
```

settings.gradle.kts

```
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.PREFER_SETTINGS)
}
```

If, for some reason, a project or a plugin declares a repository in a project, Gradle would warn you. You can however make it *fail the build* if you want to enforce that only settings repositories are used:

Example 270. Enforcing settings repositories

settings.gradle

```
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
}
```

settings.gradle.kts

```
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
}
```

Eventually, the default is equivalent to setting `PREFER_PROJECT`:

Example 271. Preferring project repositories

settings.gradle

```
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.PREFER_PROJECT)
}
```

settings.gradle.kts

```
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.PREFER_PROJECT)
}
```

Supported repository transport protocols

Maven and Ivy repositories support the use of various transport protocols. At the moment the following protocols are supported:

Table 11. Repository transport protocols

Type	Credential types	Link
`file`	none	
`http`	username/password	Documentation
`https`	username/password	Documentation
`sftp`	username/password	Documentation
`s3`	access key/secret key/session token or Environment variables	Documentation
`gcs`	default application credentials sourced from well known files, Environment variables etc.	Documentation

NOTE Username and password should never be checked in plain text into version control as part of your build file. You can store the credentials in a local `gradle.properties` file and use one of the open source Gradle plugins for encrypting and consuming credentials e.g. the credentials plugin.

The transport protocol is part of the URL definition for a repository. The following build script demonstrates how to create HTTP-based Maven and Ivy repositories:

Example 272. Declaring a Maven and Ivy repository

build.gradle

```
repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
    }

    ivy {
        url "http://repo.mycompany.com/repo"
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("http://repo.mycompany.com/maven2")
    }

    ivy {
        url = uri("http://repo.mycompany.com/repo")
    }
}
```

The following example shows how to declare SFTP repositories:

Example 273. Using the SFTP protocol for a repository

build.gradle

```
repositories {
    maven {
        url "sftp://repo.mycompany.com:22/maven2"
        credentials {
            username "user"
            password "password"
        }
    }

    ivy {
        url "sftp://repo.mycompany.com:22/repo"
        credentials {
            username "user"
            password "password"
        }
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("sftp://repo.mycompany.com:22/maven2")
        credentials {
            username = "user"
            password = "password"
        }
    }

    ivy {
        url = uri("sftp://repo.mycompany.com:22/repo")
        credentials {
            username = "user"
            password = "password"
        }
    }
}
```

For details on HTTP related authentication, see the section HTTP(S) authentication schemes configuration.

When using an AWS S3 backed repository you need to authenticate using AwsCredentials,

providing access-key and a private-key. The following example shows how to declare a S3 backed repository and providing AWS credentials:

Example 274. Declaring an S3 backed Maven and Ivy repository

build.gradle

```
repositories {
    maven {
        url "s3://myCompanyBucket/maven2"
        credentials(AwsCredentials) {
            accessKey "someKey"
            secretKey "someSecret"
            // optional
            sessionToken "someSTSToken"
        }
    }

    ivy {
        url "s3://myCompanyBucket/ivyrepo"
        credentials(AwsCredentials) {
            accessKey "someKey"
            secretKey "someSecret"
            // optional
            sessionToken "someSTSToken"
        }
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("s3://myCompanyBucket/maven2")
        credentials(AwsCredentials::class) {
            accessKey = "someKey"
            secretKey = "someSecret"
            // optional
            sessionToken = "someSTSToken"
        }
    }

    ivy {
        url = uri("s3://myCompanyBucket/ivyrepo")
        credentials(AwsCredentials::class) {
            accessKey = "someKey"
            secretKey = "someSecret"
            // optional
            sessionToken = "someSTSToken"
        }
    }
}
```

You can also delegate all credentials to the AWS sdk by using the AwsImAuthentication. The following example shows how:

Example 275. Declaring an S3 backed Maven and Ivy repository using IAM

build.gradle

```
repositories {
    maven {
        url "s3://myCompanyBucket/maven2"
        authentication {
           awsIm(AwsImAuthentication) // load from EC2 role or env var
        }
    }

    ivy {
        url "s3://myCompanyBucket/ivyrepo"
        authentication {
           awsIm(AwsImAuthentication)
        }
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("s3://myCompanyBucket/maven2")
        authentication {
            create<AwsImAuthentication>("awsIm") // load from EC2 role or env
var
        }
    }

    ivy {
        url = uri("s3://myCompanyBucket/ivyrepo")
        authentication {
            create<AwsImAuthentication>("awsIm")
        }
    }
}
```

For details on AWS S3 related authentication, see the section AWS S3 repositories configuration.

When using a Google Cloud Storage backed repository default application credentials will be used with no further configuration required:

Example 276. Declaring a Google Cloud Storage backed Maven and Ivy repository using default application credentials

build.gradle

```
repositories {
    maven {
        url "gcs://myCompanyBucket/maven2"
    }

    ivy {
        url "gcs://myCompanyBucket/ivyrepo"
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("gcs://myCompanyBucket/maven2")
    }

    ivy {
        url = uri("gcs://myCompanyBucket/ivyrepo")
    }
}
```

For details on Google GCS related authentication, see the section Google Cloud Storage repositories configuration.

HTTP(S) authentication schemes configuration

When configuring a repository using HTTP or HTTPS transport protocols, multiple authentication schemes are available. By default, Gradle will attempt to use all schemes that are supported by the Apache HttpClient library, documented here. In some cases, it may be preferable to explicitly specify which authentication schemes should be used when exchanging credentials with a remote server. When explicitly declared, only those schemes are used when authenticating to a remote repository.

You can specify credentials for Maven repositories secured by basic authentication using PasswordCredentials.

Example 277. Accessing password-protected Maven repository

build.gradle

```
repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
        credentials {
            username "user"
            password "password"
        }
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("http://repo.mycompany.com/maven2")
        credentials {
            username = "user"
            password = "password"
        }
    }
}
```

The following example show how to configure a repository to use only DigestAuthentication:

Example 278. Configure repository to use only digest authentication

build.gradle

```
repositories {
    maven {
        url 'https://repo.mycompany.com/maven2'
        credentials {
            username "user"
            password "password"
        }
        authentication {
            digest(DigestAuthentication)
        }
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("https://repo.mycompany.com/maven2")
        credentials {
            username = "user"
            password = "password"
        }
        authentication {
            create<DigestAuthentication>("digest")
        }
    }
}
```

Currently supported authentication schemes are:

BasicAuthentication

Basic access authentication over HTTP. When using this scheme, credentials are sent preemptively.

DigestAuthentication

Digest access authentication over HTTP.

HttpHeaderAuthentication

Authentication based on any custom HTTP header, e.g. private tokens, OAuth tokens, etc.

Using preemptive authentication

Gradle's default behavior is to only submit credentials when a server responds with an authentication challenge in the form of an HTTP 401 response. In some cases, the server will respond with a different code (ex. for repositories hosted on GitHub a 404 is returned) causing dependency resolution to fail. To get around this behavior, credentials may be sent to the server preemptively. To enable preemptive authentication simply configure your repository to explicitly use the BasicAuthentication scheme:

Example 279. Configure repository to use preemptive authentication

build.gradle

```
repositories {
    maven {
        url 'https://repo.mycompany.com/maven2'
        credentials {
            username "user"
            password "password"
        }
        authentication {
            basic(BasicAuthentication)
        }
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("https://repo.mycompany.com/maven2")
        credentials {
            username = "user"
            password = "password"
        }
        authentication {
            create<BasicAuthentication>("basic")
        }
    }
}
```

Using HTTP header authentication

You can specify any HTTP header for secured Maven repositories requiring token, OAuth2 or other HTTP header based authentication using HttpHeaderCredentials with HttpHeaderAuthentication.

Example 280. Accessing header-protected Maven repository

build.gradle

```
repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
        credentials(HttpHeaderCredentials) {
            name = "Private-Token"
            value = "TOKEN"
        }
        authentication {
            header(HttpHeaderAuthentication)
        }
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        url = uri("http://repo.mycompany.com/maven2")
        credentials(HttpHeaderCredentials::class) {
            name = "Private-Token"
            value = "TOKEN"
        }
        authentication {
            create<HttpHeaderAuthentication>("header")
        }
    }
}
```

AWS S3 repositories configuration

S3 configuration properties

The following system properties can be used to configure the interactions with s3 repositories:

`org.gradle.s3.endpoint`

Used to override the AWS S3 endpoint when using a non AWS, S3 API compatible, storage service.

`org.gradle.s3.maxErrorRetry`

Specifies the maximum number of times to retry a request in the event that the S3 server responds with a HTTP 5xx status code. When not specified a default value of 3 is used.

S3 URL formats

S3 URL's are 'virtual-hosted-style' and must be in the following format

```
s3://<bucketName>[.<regionSpecificEndpoint>]/<s3Key>
```

e.g. `s3://myBucket.s3.eu-central-1.amazonaws.com/maven/release`

- `myBucket` is the AWS S3 bucket name.
- `s3.eu-central-1.amazonaws.com` is the *optional* region specific endpoint.
- `/maven/release` is the AWS S3 key (unique identifier for an object within a bucket)

S3 proxy settings

A proxy for S3 can be configured using the following system properties:

- `https.proxyHost`
- `https.proxyPort`
- `https.proxyUser`
- `https.proxyPassword`
- `http.nonProxyHosts` (NOTE: this is not a typo.)

If the `org.gradle.s3.endpoint` property has been specified with a HTTP (not HTTPS) URI the following system proxy settings can be used:

- `http.proxyHost`
- `http.proxyPort`
- `http.proxyUser`
- `http.proxyPassword`
- `http.nonProxyHosts`

AWS S3 V4 Signatures (AWS4-HMAC-SHA256)

Some of the AWS S3 regions (eu-central-1 - Frankfurt) require that all HTTP requests are signed in accordance with AWS's signature version 4. It is recommended to specify S3 URL's containing the region specific endpoint when using buckets that require V4 signatures. e.g.

```
s3://somebucket.s3.eu-central-1.amazonaws.com/maven/release
```

NOTE

When a region-specific endpoint is not specified for buckets requiring V4 Signatures, Gradle will use the default AWS region (us-east-1) and the following warning will appear on the console:

> Attempting to re-send the request to .... with AWS V4 authentication. To avoid this warning in the future, use region-specific endpoint to access buckets located in regions that require V4 signing.

Failing to specify the region-specific endpoint for buckets requiring V4 signatures means:

- 3 round-trips to AWS, as opposed to one, for every file upload and download.
- Depending on location - increased network latencies and slower builds.
- Increased likelihood of transmission failures.

AWS S3 Cross Account Access

Some organizations may have multiple AWS accounts, e.g. one for each team. The AWS account of the bucket owner is often different from the artifact publisher and consumers. The bucket owner needs to be able to grant the consumers access otherwise the artifacts will only be usable by the publisher's account. This is done by adding the `bucket-owner-full-control` Canned ACL to the uploaded objects. Gradle will do this in every upload. Make sure the publisher has the required IAM permission, `PutObjectAcl` (and `PutObjectVersionAcl` if bucket versioning is enabled), either directly or via an assumed IAM Role (depending on your case). You can read more at AWS S3 Access Permissions.

Google Cloud Storage repositories configuration

GCS configuration properties

The following system properties can be used to configure the interactions with Google Cloud Storage repositories:

`org.gradle.gcs.endpoint`

Used to override the Google Cloud Storage endpoint when using a non-Google Cloud Platform, Google Cloud Storage API compatible, storage service.

`org.gradle.gcs.servicePath`

Used to override the Google Cloud Storage root service path which the Google Cloud Storage client builds requests from, defaults to `/`.

GCS URL formats

Google Cloud Storage URL's are 'virtual-hosted-style' and must be in the following format `gcs://<bucketName>/<objectKey>`

e.g. `gcs://myBucket/maven/release`

- `myBucket` is the Google Cloud Storage bucket name.
- `/maven/release` is the Google Cloud Storage key (unique identifier for an object within a bucket)

Handling credentials

Repository credentials should never be part of your build script but rather be kept external. Gradle provides an API in artifact repositories that allows you to declare only the type of required credentials. Credential values are looked up from the Gradle Properties during the build that requires them.

For example, given repository configuration:

Example 281. Externalized repository credentials

build.gradle

```
repositories {
    maven {
        name = 'mySecureRepository'
        credentials(PasswordCredentials)
        // url = uri(<<some repository url>>)
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        name = "mySecureRepository"
        credentials(PasswordCredentials::class)
        // url = uri(<<some repository url>>)
    }
}
```

The username and password will be looked up from `mySecureRepositoryUsername` and `mySecureRepositoryPassword` properties.

Note that the configuration property prefix - the identity - is determined from the repository name. Credentials can then be provided in any of supported ways for Gradle Properties - `gradle.properties` file, command line arguments, environment variables or a combination of those options.

Also, note that credentials will only be required if the invoked build requires them. If for example a project is configured to publish artifacts to a secured repository, but the build does not invoke publishing task, Gradle will not require publishing credentials to be present. On the other hand, if

the build needs to execute a task that requires credentials at some point, Gradle will check for credential presence first thing and will not start running any of the tasks if it knows that the build will fail at a later point because of missing credentials.

Here is a downloadable sample that demonstrates the concept in more detail.

Lookup is only supported for credentials listed in the Table 12.

Table 12. Credentials that support value lookup and their corresponding properties

Type	Argument	Base property name	Required ?
`PasswordCredentials`	`username`	`Username`	required
	`password`	`Password`	required
`AwsCredentials`	`accessKey`	`AccessKey`	required
	`secretKey`	`SecretKey`	required
	`sessionToken`	`SessionToken`	optional
`HttpHeaderCredentials`	`name`	`AuthHeaderName`	required
	`value`	`AuthHeaderValue`	required

Declaring dependencies

Before looking at dependency declarations themselves, the concept of *dependency configuration* needs to be defined.

What are dependency configurations

Every dependency declared for a Gradle project applies to a specific scope. For example some dependencies should be used for compiling source code whereas others only need to be available at runtime. Gradle represents the scope of a dependency with the help of a Configuration. Every configuration can be identified by a unique name.

Many Gradle plugins add pre-defined configurations to your project. The Java plugin, for example, adds configurations to represent the various classpaths it needs for source code compilation, executing tests and the like. See the Java plugin chapter for an example.

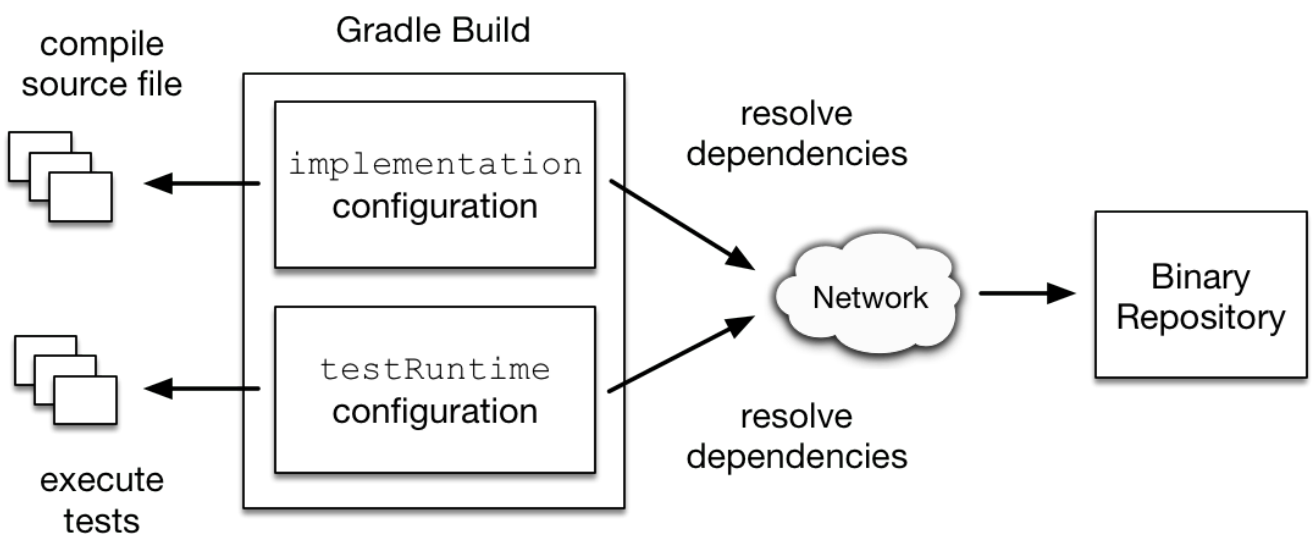


Figure 15. Configurations use declared dependencies for specific purposes

For more examples on the usage of configurations to navigate, inspect and post-process metadata and artifacts of assigned dependencies, have a look at the resolution result APIs.

Configuration inheritance and composition

A configuration can extend other configurations to form an inheritance hierarchy. Child configurations inherit the whole set of dependencies declared for any of its superconfigurations.

Configuration inheritance is heavily used by Gradle core plugins like the Java plugin. For example the `testImplementation` configuration extends the `implementation` configuration. The configuration hierarchy has a practical purpose: compiling tests requires the dependencies of the source code under test on top of the dependencies needed write the test class. A Java project that uses JUnit to write and execute test code also needs Guava if its classes are imported in the production source code.

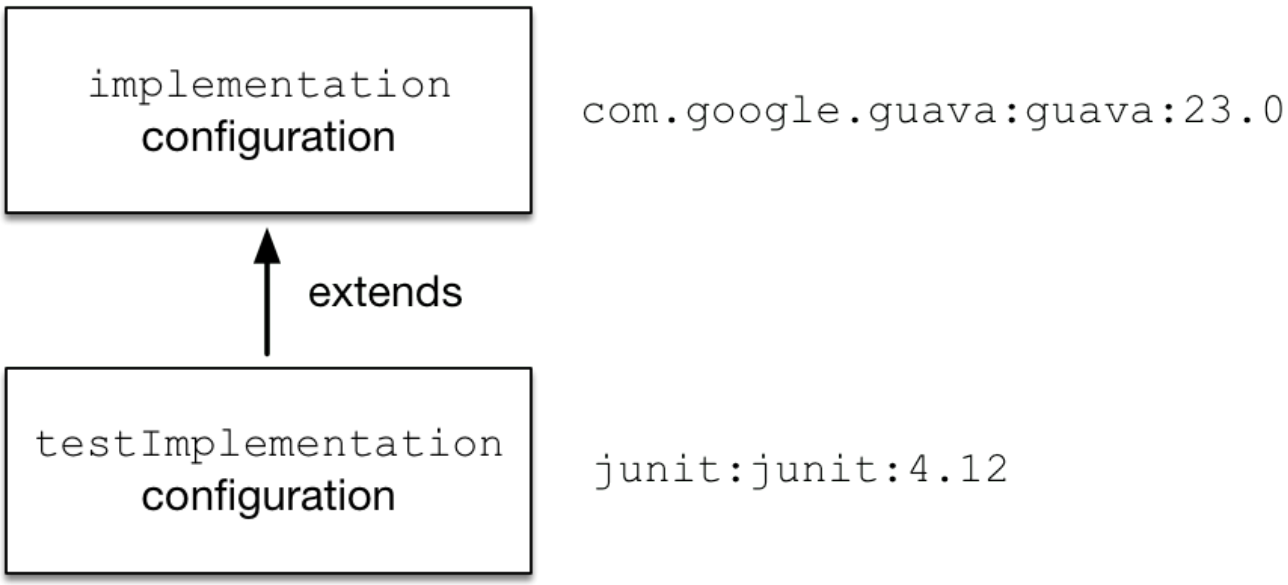


Figure 16. Configuration inheritance provided by the Java plugin

Under the covers the `testImplementation` and `implementation` configurations form an inheritance

hierarchy by calling the method Configuration.extendsFrom(org.gradle.api.artifacts.Configuration[]). A configuration can extend any other configuration irrespective of its definition in the build script or a plugin.

Let's say you wanted to write a suite of smoke tests. Each smoke test makes a HTTP call to verify a web service endpoint. As the underlying test framework the project already uses JUnit. You can define a new configuration named `smokeTest` that extends from the `testImplementation` configuration to reuse the existing test framework dependency.

Example 282. Extending a configuration from another configuration

build.gradle

```
configurations {
    smokeTest.extendsFrom testImplementation
}

dependencies {
    testImplementation 'junit:junit:4.13'
    smokeTest 'org.apache.httpcomponents:httpclient:4.5.5'
}
```

build.gradle.kts

```
val smokeTest by configurations.creating {
    extendsFrom(configurations.testImplementation.get())
}

dependencies {
    testImplementation("junit:junit:4.13")
    smokeTest("org.apache.httpcomponents:httpclient:4.5.5")
}
```

Resolvable and consumable configurations

Configurations are a fundamental part of dependency resolution in Gradle. In the context of dependency resolution, it is useful to distinguish between a *consumer* and a *producer*. Along these lines, configurations have at least 3 different roles:

1. to declare dependencies
2. as a *consumer*, to resolve a set of dependencies to files
3. as a *producer*, to expose artifacts and their dependencies for consumption by other projects (such *consumable* configurations usually represent the variants the producer offers to its consumers)

For example, to express that an application `app` *depends on* library `lib`, *at least* one configuration is required:

Example 283. Configurations are used to declare dependencies

build.gradle

```
configurations {
    // declare a "configuration" named "someConfiguration"
    someConfiguration
}
dependencies {
    // add a project dependency to the "someConfiguration" configuration
    someConfiguration project(":lib")
}
```

build.gradle.kts

```
// declare a "configuration" named "someConfiguration"
val someConfiguration by configurations.creating

dependencies {
    // add a project dependency to the "someConfiguration" configuration
    someConfiguration(project(":lib"))
}
```

Configurations can inherit dependencies from other configurations by extending from them. Now, notice that the code above doesn't tell us anything about the intended *consumer* of this configuration. In particular, it doesn't tell us how the configuration is meant to be *used*. Let's say that `lib` is a Java library: it might expose different things, such as its API, implementation, or test fixtures. It might be necessary to change how we resolve the dependencies of `app` depending upon the task we're performing (compiling against the API of `lib`, executing the application, compiling tests, etc.). To address this problem, you'll often find companion configurations, which are meant to unambiguously declare the usage:

Example 284. Configurations representing concrete dependency graphs

build.gradle

```
configurations {
    // declare a configuration that is going to resolve the compile classpath
of the application
    compileClasspath.extendsFrom(someConfiguration)

    // declare a configuration that is going to resolve the runtime classpath
of the application
    runtimeClasspath.extendsFrom(someConfiguration)
}
```

build.gradle.kts

```
configurations {
    // declare a configuration that is going to resolve the compile classpath
of the application
    compileClasspath.extendsFrom(someConfiguration)

    // declare a configuration that is going to resolve the runtime classpath
of the application
    runtimeClasspath.extendsFrom(someConfiguration)
}
```

At this point, we have 3 different configurations with different roles:

- `someConfiguration` declares the dependencies of my application. It's just a bucket that can hold a list of dependencies.
- `compileClasspath` and `runtimeClasspath` are configurations *meant to be resolved*: when resolved they should contain the compile classpath, and the runtime classpath of the application respectively.

This distinction is represented by the `canBeResolved` flag in the `Configuration` type. A configuration that *can be resolved* is a configuration for which we can compute a dependency graph, because it contains all the necessary information for resolution to happen. That is to say we're going to compute a dependency graph, resolve the components in the graph, and eventually get artifacts. A configuration which has `canBeResolved` set to `false` is not meant to be resolved. Such a configuration is there *only to declare dependencies*. The reason is that depending on the usage (compile classpath, runtime classpath), it *can* resolve to different graphs. It is an error to try to resolve a configuration which has `canBeResolved` set to `false`. To some extent, this is similar to an *abstract class* (`canBeResolved`=false) which is not supposed to be instantiated, and a concrete class extending the abstract class (`canBeResolved`=true). A resolvable configuration will extend at least one non-

resolvable configuration (and may extend more than one).

On the other end, at the library project side (the *producer*), we also use configurations to represent what can be consumed. For example, the library may expose an API or a runtime, and we would attach artifacts to either one, the other, or both. Typically, to compile against `lib`, we need the API of `lib`, but we don't need its runtime dependencies. So the `lib` project will expose an `apiElements` configuration, which is aimed at consumers looking for its API. Such a configuration is consumable, but is not meant to be resolved. This is expressed via the *canBeConsumed* flag of a `Configuration`:

Example 285. Setting up configurations

build.gradle

```
configurations {
    // A configuration meant for consumers that need the API of this
component
    exposedApi {
        // This configuration is an "outgoing" configuration, it's not meant
to be resolved
        canBeResolved = false
        // As an outgoing configuration, explain that consumers may want to
consume it
        canBeConsumed = true
    }
    // A configuration meant for consumers that need the implementation of
this component
    exposedRuntime {
        canBeResolved = false
        canBeConsumed = true
    }
}
```

build.gradle.kts

```
configurations {
    // A configuration meant for consumers that need the API of this
component
    create("exposedApi") {
        // This configuration is an "outgoing" configuration, it's not meant
to be resolved
        isCanBeResolved = false
        // As an outgoing configuration, explain that consumers may want to
consume it
        isCanBeConsumed = true
    }
    // A configuration meant for consumers that need the implementation of
this component
    create("exposedRuntime") {
        isCanBeResolved = false
        isCanBeConsumed = true
    }
}
```

In short, a configuration's role is determined by the `canBeResolved` and `canBeConsumed` flag

combinations:

Table 13. Configuration roles

Configuration role	can be resolved	can be consumed
Bucket of dependencies	false	false
Resolve for certain usage	true	false
Exposed to consumers	false	true
Legacy, don't use	true	true

For backwards compatibility, both flags have a default value of `true`, but as a plugin author, you should always determine the right values for those flags, or you might accidentally introduce resolution errors.

Choosing the right configuration for dependencies

The choice of the configuration where you declare a dependency is important. However there is no fixed rule into which configuration a dependency must go. It mostly depends on the way the configurations are organised, which is most often a property of the applied plugin(s).

For example, in the `java` plugin, the created configuration are documented and should serve as the basis for determining where to declare a dependency, based on its role for your code.

As a recommendation, plugins should clearly document the way their configurations are linked together and should strive as much as possible to isolate their roles.

Defining custom configurations

You can define configurations yourself, so-called *custom configurations*. A custom configuration is useful for separating the scope of dependencies needed for a dedicated purpose.

Let's say you wanted to declare a dependency on the Jasper Ant task for the purpose of pre-compiling JSP files that should *not* end up in the classpath for compiling your source code. It's fairly simple to achieve that goal by introducing a custom configuration and using it in a task.

Example 286. Declaring and using a custom configuration

build.gradle

```
configurations {
    jasper
}

repositories {
    mavenCentral()
}

dependencies {
    jasper 'org.apache.tomcat.embed:tomcat-embed-jasper:9.0.2'
}

tasks.register('preCompileJsps') {
    doLast {
        ant.taskdef(classname: 'org.apache.jasper.JspC',
                    name: 'jasper',
                    classpath: configurations.jasper.asPath)
        ant.jasper(validateXml: false,
                   uriroot: file('src/main/webapp'),
                   outputDir: file("$buildDir/compiled-jsps"))
    }
}
```

build.gradle.kts

```
val jasper by configurations.creating

repositories {
    mavenCentral()
}

dependencies {
    jasper("org.apache.tomcat.embed:tomcat-embed-jasper:9.0.2")
}

tasks.register("preCompileJsps") {
    doLast {
        ant.withGroovyBuilder {
            "taskdef"("classname" to "org.apache.jasper.JspC",
                      "name" to "jasper",
                      "classpath" to jasper.asPath)
            "jasper"("validateXml" to false,
                     "uriroot" to file("src/main/webapp"),
                     "outputDir" to file("$buildDir/compiled-jsps"))
        }
    }
}
```

You can manage project configurations with a `configurations` object. Configurations have a name and can extend each other. To learn more about this API have a look at ConfigurationContainer.

Different kinds of dependencies

Module dependencies

Module dependencies are the most common dependencies. They refer to a module in a repository.

Example 287. Module dependencies

build.gradle

```
dependencies {
    runtimeOnly group: 'org.springframework', name: 'spring-core', version:
'2.5'
    runtimeOnly 'org.springframework:spring-core:2.5',
            'org.springframework:spring-aop:2.5'
    runtimeOnly(
        [group: 'org.springframework', name: 'spring-core', version: '2.5'],
        [group: 'org.springframework', name: 'spring-aop', version: '2.5']
    )
    runtimeOnly('org.hibernate:hibernate:3.0.5') {
        transitive = true
    }
    runtimeOnly group: 'org.hibernate', name: 'hibernate', version: '3.0.5',
transitive: true
    runtimeOnly(group: 'org.hibernate', name: 'hibernate', version: '3.0.5')
{
        transitive = true
    }
}
```

build.gradle.kts

```
dependencies {
    runtimeOnly(group = "org.springframework", name = "spring-core", version
= "2.5")
    runtimeOnly("org.springframework:spring-aop:2.5")
    runtimeOnly("org.hibernate:hibernate:3.0.5") {
        isTransitive = true
    }
    runtimeOnly(group = "org.hibernate", name = "hibernate", version =
"3.0.5") {
        isTransitive = true
    }
}
```

See the DependencyHandler class in the API documentation for more examples and a complete reference.

Gradle provides different notations for module dependencies. There is a string notation and a map notation. A module dependency has an API which allows further configuration. Have a look at ExternalModuleDependency to learn all about the API. This API provides properties and

configuration methods. Via the string notation you can define a subset of the properties. With the map notation you can define all properties. To have access to the complete API, either with the map or with the string notation, you can assign a single dependency to a configuration together with a closure.

NOTE

If you declare a module dependency, Gradle looks for a module metadata file (`.module`, `.pom` or `ivy.xml`) in the repositories. If such a module metadata file exists, it is parsed and the artifacts of this module (e.g. `hibernate-3.0.5.jar`) as well as its dependencies (e.g. `cglib`) are downloaded. If no such module metadata file exists, as of Gradle 6.0, you need to configure metadata sources definitions to look for an artifact file called `hibernate-3.0.5.jar` directly.

NOTE

In Maven, a module can have one and only one artifact.

In Gradle and Ivy, a module can have multiple artifacts. Each artifact can have a different set of dependencies.

File dependencies

Projects sometimes do not rely on a binary repository product e.g. JFrog Artifactory or Sonatype Nexus for hosting and resolving external dependencies. It's common practice to host those dependencies on a shared drive or check them into version control alongside the project source code. Those dependencies are referred to as *file dependencies*, the reason being that they represent a file without any metadata (like information about transitive dependencies, the origin or its author) attached to them.

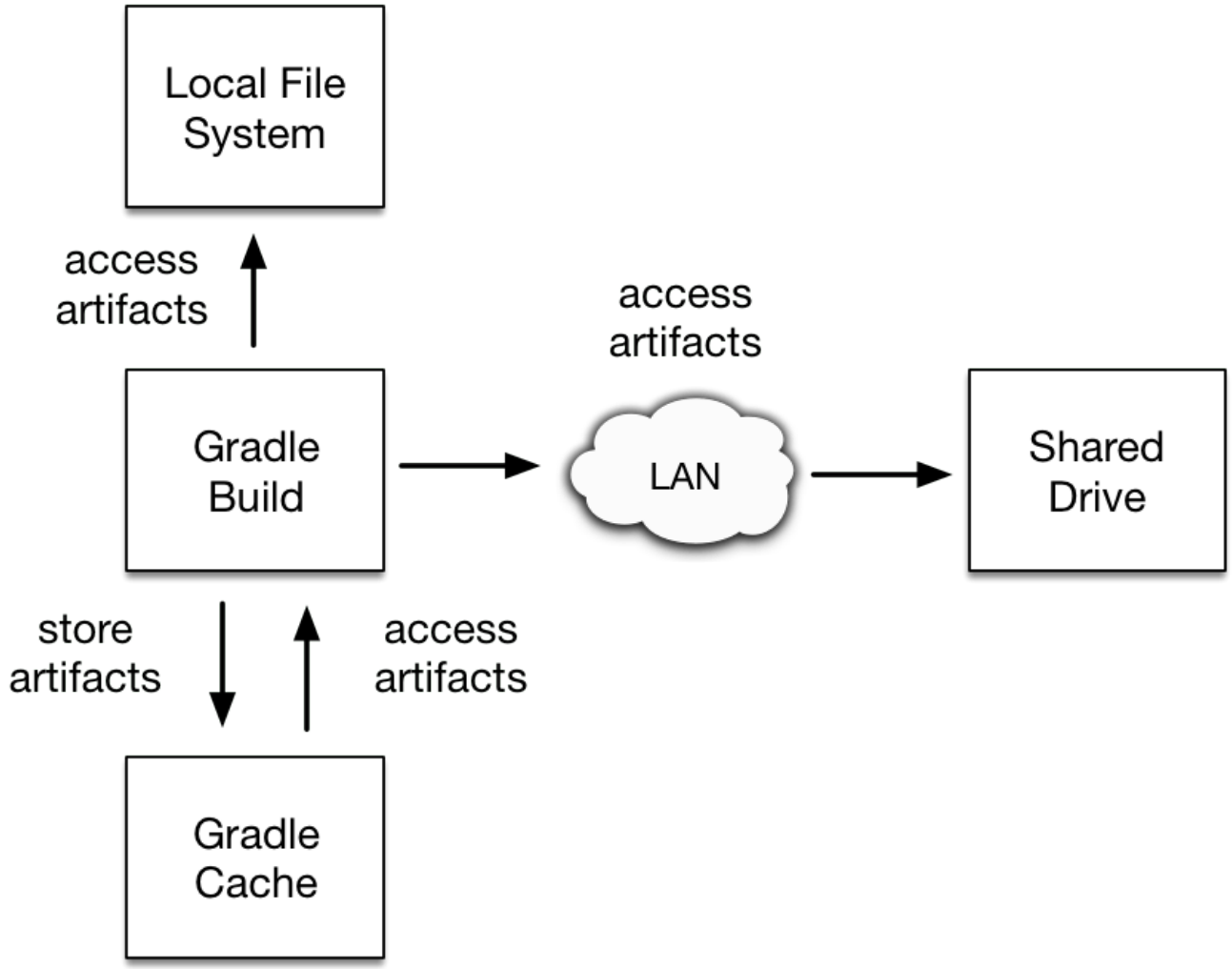


Figure 17. Resolving file dependencies from the local file system and a shared drive

The following example resolves file dependencies from the directories `ant`, `libs` and `tools`.

Example 288. Declaring multiple file dependencies

build.gradle

```
configurations {
    antContrib
    externalLibs
    deploymentTools
}

dependencies {
    antContrib files('ant/antcontrib.jar')
    externalLibs files('libs/commons-lang.jar', 'libs/log4j.jar')
    deploymentTools(fileTree('tools') { include '*.exe' })
}
```

build.gradle.kts

```
configurations {
    create("antContrib")
    create("externalLibs")
    create("deploymentTools")
}

dependencies {
    "antContrib"(files("ant/antcontrib.jar"))
    "externalLibs"(files("libs/commons-lang.jar", "libs/log4j.jar"))
    "deploymentTools"(fileTree("tools") { include("*.exe") })
}
```

As you can see in the code example, every dependency has to define its exact location in the file system. The most prominent methods for creating a file reference are Project.files(java.lang.Object...), ProjectLayout.files(java.lang.Object...) and Project.fileTree(java.lang.Object) Alternatively, you can also define the source directory of one or many file dependencies in the form of a flat directory repository.

NOTE The order of the files in a `FileTree` is not stable, even on a single computer. It means that dependency configuration seeded with such a construct may produce a resolution result which has a different ordering, possibly impacting the cacheability of tasks using the result as an input. Using the simpler `files` instead is recommended where possible.

File dependencies allow you to directly add a set of files to a configuration, without first adding them to a repository. This can be useful if you cannot, or do not want to, place certain files in a repository. Or if you do not want to use any repositories at all for storing your dependencies.

To add some files as a dependency for a configuration, you simply pass a file collection as a dependency:

Example 289. File dependencies

build.gradle

```
dependencies {
    runtimeOnly files('libs/a.jar', 'libs/b.jar')
    runtimeOnly fileTree('libs') { include '*.jar' }
}
```

build.gradle.kts

```
dependencies {
    runtimeOnly(files("libs/a.jar", "libs/b.jar"))
    runtimeOnly(fileTree("libs") { include("*.jar") })
}
```

File dependencies are not included in the published dependency descriptor for your project. However, file dependencies are included in transitive project dependencies within the same build. This means they cannot be used outside the current build, but they can be used within the same build.

NOTE The order of the files in a `FileTree` is not stable, even on a single computer. It means that dependency configuration seeded with such a construct may produce a resolution result which has a different ordering, possibly impacting the cacheability of tasks using the result as an input. Using the simpler `files` instead is recommended where possible.

You can declare which tasks produce the files for a file dependency. You might do this when, for example, the files are generated by the build.

Example 290. Generated file dependencies

build.gradle

```
dependencies {
    implementation files(layout.buildDirectory.dir('classes')) {
        builtBy 'compile'
    }
}

tasks.register('compile') {
    doLast {
        println 'compiling classes'
    }
}

tasks.register('list') {
    dependsOn configurations.compileClasspath
    doLast {
        println "classpath = ${configurations.compileClasspath.collect { File
file -> file.name }}"
    }
}
```

build.gradle.kts

```
dependencies {
    implementation(files(layout.buildDirectory.dir("classes")) {
        builtBy("compile")
    })
}

tasks.register("compile") {
    doLast {
        println("compiling classes")
    }
}

tasks.register("list") {
    dependsOn(configurations["compileClasspath"])
    doLast {
        println("classpath = ${configurations["compileClasspath"].map { file:
File -> file.name }}")
    }
}
```

```
$ gradle -q list
compiling classes
classpath = [classes]
```

Versioning of file dependencies

It is recommended to clearly express the intention and a concrete version for file dependencies. File dependencies are not considered by Gradle's version conflict resolution. Therefore, it is extremely important to assign a version to the file name to indicate the distinct set of changes shipped with it. For example `commons-beanutils-1.3.jar` lets you track the changes of the library by the release notes.

As a result, the dependencies of the project are easier to maintain and organize. It is much easier to uncover potential API incompatibilities by the assigned version.

Project dependencies

Software projects often break up software components into modules to improve maintainability and prevent strong coupling. Modules can define dependencies between each other to reuse code within the same project.

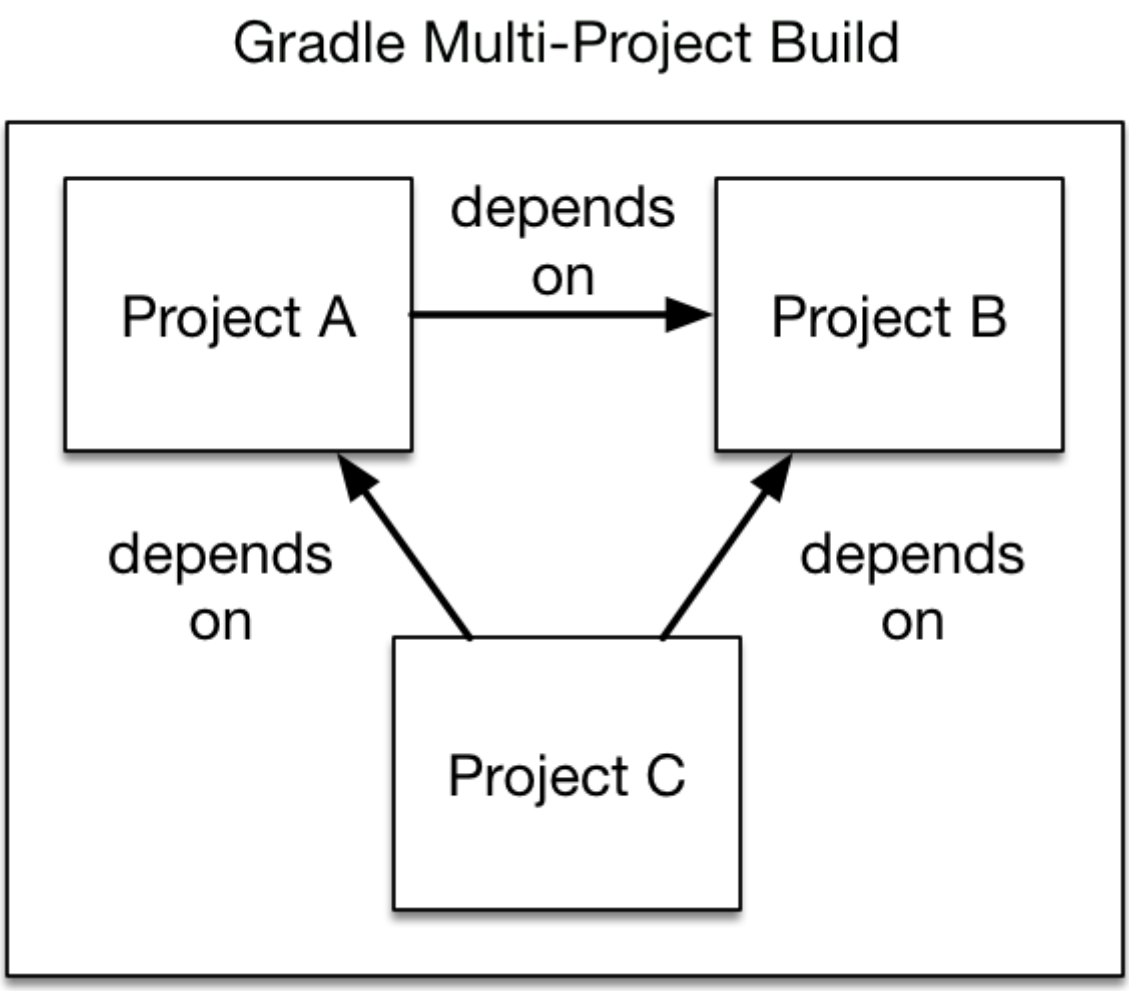


Figure 18. Dependencies between projects

Gradle can model dependencies between modules. Those dependencies are called *project dependencies* because each module is represented by a Gradle project.

Example 291. Project dependencies

build.gradle

```
dependencies {
    implementation project(':shared')
}
```

build.gradle.kts

```
dependencies {
    implementation(project(":shared"))
}
```

At runtime, the build automatically ensures that project dependencies are built in the correct order and added to the classpath for compilation. The chapter Authoring Multi-Project Builds discusses how to set up and configure multi-project builds in more detail.

For more information see the API documentation for ProjectDependency.

The following example declares the dependencies on the `utils` and `api` project from the `web-service` project. The method Project.project(java.lang.String) creates a reference to a specific subproject by path.

Example 292. Declaring project dependencies

web-service/build.gradle

```
dependencies {
    implementation project(':utils')
    implementation project(':api')
}
```

web-service/build.gradle.kts

```
dependencies {
    implementation(project(":utils"))
    implementation(project(":api"))
}
```

Type-safe project dependencies

WARNING

Type-safe project accessors are an incubating feature which must be enabled explicitly. Implementation may change at any time. To add support for type-safe project accessors, add this to your `settings.gradle(.kts)` file:

```
enableFeaturePreview("TYPESAFE_PROJECT_ACCESSORS")
```

One issue with the `project(":some:path")` notation is that you have to remember the path to every project you want to depend on. In addition, changing a project path requires you to change all places where the project dependency is used, but it is easy to miss one or more occurrences (because you have to rely on search and replace).

Since Gradle 7, Gradle offers an experimental type-safe API for project dependencies. The same example as above can now be rewritten as:

Example 293. Declaring project dependencies using the type-safe API

web-service/build.gradle

```
dependencies {
    implementation projects.utils
    implementation projects.api
}
```

web-service/build.gradle.kts

```
dependencies {
    implementation(projects.utils)
    implementation(projects.api)
}
```

The type-safe API has the advantage of providing IDE completion so you don't need to figure out the actual names of the projects.

If you add or remove a project that uses the Kotlin DSL, build script compilation fails if you forget to update a dependency.

The project accessors are mapped from the *project path*. For example, if a project path is `:commons:utils:some:lib` then the project accessor will be `projects.commons.utils.some.lib` (which is the short-hand notation for `projects.getCommons().getUtils().getSome().getLib()`).

A project name with kebab case (`some-lib`) or snake case (`some_lib`) will be converted to camel case in accessors: `projects.someLib`.

Local forks of module dependencies

A module dependency can be substituted by a dependency to a local fork of the sources of that module, if the module itself is built with Gradle. This can be done by utilising composite builds. This allows you, for example, to fix an issue in a library you use in an application by using, and building, a locally patched version instead of the published binary version. The details of this are described in the section on composite builds.

Gradle distribution-specific dependencies

Gradle API dependency

You can declare a dependency on the API of the current version of Gradle by using the DependencyHandler.gradleApi() method. This is useful when you are developing custom Gradle tasks or plugins.

Example 294. Gradle API dependencies

build.gradle

```
dependencies {
    implementation gradleApi()
}
```

build.gradle.kts

```
dependencies {
    implementation(gradleApi())
}
```

Gradle TestKit dependency

You can declare a dependency on the TestKit API of the current version of Gradle by using the DependencyHandler.gradleTestKit() method. This is useful for writing and executing functional tests for Gradle plugins and build scripts.

Example 295. Gradle TestKit dependencies

build.gradle

```
dependencies {
    testImplementation gradleTestKit()
}
```

build.gradle.kts

```
dependencies {
    testImplementation(gradleTestKit())
}
```

The TestKit chapter explains the use of TestKit by example.

Local Groovy dependency

You can declare a dependency on the Groovy that is distributed with Gradle by using the DependencyHandler.localGroovy() method. This is useful when you are developing custom Gradle

tasks or plugins in Groovy.

Example 296. Gradle's Groovy dependencies

build.gradle

```
dependencies {
    implementation localGroovy()
}
```

build.gradle.kts

```
dependencies {
    implementation(localGroovy())
}
```

Documenting dependencies

When you declare a dependency or a dependency constraint, you can provide a custom reason for the declaration. This makes the dependency declarations in your build script and the dependency insight report easier to interpret.

Example 297. Giving a reason for choosing a certain module version in a dependency declaration

build.gradle

```
plugins {
    id 'java-library'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation('org.ow2.asm:asm:7.1') {
        because 'we require a JDK 9 compatible bytecode generator'
    }
}
```

build.gradle.kts

```
plugins {
    `java-library`
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.ow2.asm:asm:7.1") {
        because("we require a JDK 9 compatible bytecode generator")
    }
}
```

Example: Using the dependency insight report with custom reasons

Output of `gradle -q dependencyInsight --dependency asm`

```
> gradle -q dependencyInsight --dependency asm
org.ow2.asm:asm:7.1
  Variant compile:
    | Attribute Name                 | Provided | Requested    |
    |--------------------------------|----------|--------------|
    | org.gradle.status              | release  |              |
    | org.gradle.category            | library  | library      |
    | org.gradle.libraryelements     | jar      | classes      |
    | org.gradle.usage               | java-api | java-api     |
    | org.gradle.dependency.bundling |          | external     |
    | org.gradle.jvm.environment     |          | standard-jvm |
    | org.gradle.jvm.version         |          | 11           |
   Selection reasons:
      - Was requested: we require a JDK 9 compatible bytecode generator

org.ow2.asm:asm:7.1
\--- compileClasspath

A web-based, searchable dependency report is available by adding the --scan option.
```

Resolving specific artifacts from a module dependency

Whenever Gradle tries to resolve a module from a Maven or Ivy repository, it looks for a metadata file and the default artifact file, a JAR. The build fails if none of these artifact files can be resolved. Under certain conditions, you might want to tweak the way Gradle resolves artifacts for a dependency.

- The dependency only provides a non-standard artifact without any metadata e.g. a ZIP file.
- The module metadata declares more than one artifact e.g. as part of an Ivy dependency descriptor.
- You only want to download a specific artifact without any of the transitive dependencies declared in the metadata.

Gradle is a polyglot build tool and not limited to just resolving Java libraries. Let's assume you wanted to build a web application using JavaScript as the client technology. Most projects check in external JavaScript libraries into version control. An external JavaScript library is no different than a reusable Java library so why not download it from a repository instead?

Google Hosted Libraries is a distribution platform for popular, open-source JavaScript libraries. With the help of the artifact-only notation you can download a JavaScript library file e.g. JQuery. The `@` character separates the dependency's coordinates from the artifact's file extension.

Example 298. Resolving a JavaScript artifact for a declared dependency

build.gradle

```
repositories {
    ivy {
        url 'https://ajax.googleapis.com/ajax/libs'
        patternLayout {
            artifact '[organization]/[revision]/[module].[ext]'
        }
        metadataSources {
            artifact()
        }
    }
}

configurations {
    js
}

dependencies {
    js 'jquery:jquery:3.2.1@js'
}
```

build.gradle.kts

```
repositories {
    ivy {
        url = uri("https://ajax.googleapis.com/ajax/libs")
        patternLayout {
            artifact("[organization]/[revision]/[module].[ext]")
        }
        metadataSources {
            artifact()
        }
    }
}

configurations {
    create("js")
}

dependencies {
    "js"("jquery:jquery:3.2.1@js")
}
```

Some modules ship different "flavors" of the same artifact or they publish multiple artifacts that belong to a specific module version but have a different purpose. It’s common for a Java library to publish the artifact with the compiled class files, another one with just the source code in it and a third one containing the Javadocs.

In JavaScript, a library may exist as uncompressed or minified artifact. In Gradle, a specific artifact identifier is called *classifier*, a term generally used in Maven and Ivy dependency management.

Let’s say we wanted to download the minified artifact of the JQuery library instead of the uncompressed file. You can provide the classifier `min` as part of the dependency declaration.

Example 299. Resolving a JavaScript artifact with classifier for a declared dependency

build.gradle

```
repositories {
    ivy {
        url 'https://ajax.googleapis.com/ajax/libs'
        patternLayout {
            artifact '
[organization]/[revision]/[module](.[classifier]).[ext]'
        }
        metadataSources {
            artifact()
        }
    }
}

configurations {
    js
}

dependencies {
    js 'jquery:jquery:3.2.1:min@js'
}
```

build.gradle.kts

```
repositories {
    ivy {
        url = uri("https://ajax.googleapis.com/ajax/libs")
        patternLayout {

artifact("[organization]/[revision]/[module](.[classifier]).[ext]")
        }
        metadataSources {
            artifact()
        }
    }
}

configurations {
    create("js")
}

dependencies {
    "js"("jquery:jquery:3.2.1:min@js")
}
```

Supported Metadata formats

External module dependencies require module metadata (so that, typically, Gradle can figure out the transitive dependencies of a module). To do so, Gradle supports different metadata formats.

You can also tweak which format will be looked up in the repository definition.

Gradle Module Metadata files

Gradle Module Metadata has been specifically designed to support all features of Gradle's dependency management model and is hence the preferred format. You can find its specification here.

POM files

Gradle natively supports Maven POM files. It's worth noting that by default Gradle will first look for a POM file, but if this file contains a special marker, Gradle will use Gradle Module Metadata instead.

Ivy files

Similarly, Gradle supports Apache Ivy metadata files. Again, Gradle will first look for an `ivy.xml` file, but if this file contains a special marker, Gradle will use Gradle Module Metadata instead.

Understanding the difference between libraries and applications

Producers vs consumers

A key concept in dependency management with Gradle is the difference between consumers and producers.

When you *build* a library, you are effectively on the *producer* side: you are producing *artifacts* which are going to be *consumed* by someone else, the *consumer*.

A lot of problems with traditional build systems is that they don't make the difference between a producer and a consumer.

A *consumer* needs to be understood in the large sense:

- a project that depends on another project is a *consumer*
- a *task* that depends on an artifact is a finer grained consumer

In dependency management, a lot of the decisions we make depend on the type of project we are building, that is to say, what kind of *consumer* we are.

Producer variants

A producer may want to generate different artifacts for different kinds of consumers: for the same source code, different *binaries* are produced. Or, a project may produce artifacts which are for consumption by other projects (same repository) but not for external use.

A typical example in the Java world is the Guava library which is published in different versions: one for Java projects, and one for Android projects.

However, it's the consumer responsibility to tell what version to use, and it's the dependency management engine responsibility to ensure *consistency of the graph* (for example making sure that you don't end up with both Java and Android versions of Guava on your classpath). This is where the variant model of Gradle comes into play.

In Gradle, *producer variants* are exposed via consumable configurations.

Strong encapsulation

In order for a producer to compile a library, it needs all its *implementation dependencies* on the compile classpath. There are dependencies which are only required *as an implementation detail* of the library and there are libraries which are effectively part of the API.

However, a library *depending* on this produced library only needs to "see" the public API of your library and therefore the dependencies of this API. It's a subset of the compile classpath of the producer: this is strong encapsulation of dependencies.

The consequence is that a dependency which is assigned to the `implementation` configuration of a library *does not end up on the compile classpath of the consumer*. On the other hand, a dependency which is assigned to the `api` configuration of a library *would end up on the compile classpath of the consumer*. At *runtime*, however, all dependencies are required. Gradle makes the difference

between different kinds of consumer even within a single project: the Java compile task, for example, is a different consumer than the Java exec task.

More details on the segregation of API and runtime dependencies in the Java world can be found here.

Being respectful of consumers

Whenever, as a developer, you decide to include a dependency, you must understand that there are *consequences for your consumers*. For example, if you add a dependency to your project, it becomes a *transitive dependency* of your consumers, and therefore may participate in conflict resolution if the consumer needs a different version.

A lot of the problems Gradle handles are about fixing the mismatch between the expectations of a consumer and a producer.

However, some projects are easier than others:

- if you are at the end of the consumption chain, that is to say you build an *application*, then there are effectively *no consumer* of your project (apart from final customers): adding exclusions will have no other consequence than fixing your problem.
- however if you are a library, adding exclusions may prevent consumers from working properly, because they would exercise a path of the code that you don't

Always keep in mind that the solution you choose to fix a problem can "leak" to your consumers. This documentation aims at guiding you to find the right solution to the right problem, and more importantly, make decisions which help the resolution engine to take the right decisions in case of conflicts.

Viewing and debugging dependencies

Gradle provides sufficient tooling to navigate large dependency graphs and mitigate situations that can lead to dependency hell. Users can choose to render the full graph of dependencies as well as identify the selection reason and origin for a dependency. The origin of a dependency can be a declared dependency in the build script or a transitive dependency in graph plus their corresponding configuration. Gradle offers both capabilities through visual representation via build scans and as command line tooling.

Build scans

NOTE	If you do not know what build scans are, be sure to check them out!

A build scan can visualize dependencies as a navigable, searchable tree. Additional context information can be rendered by clicking on a specific dependency in the graph.

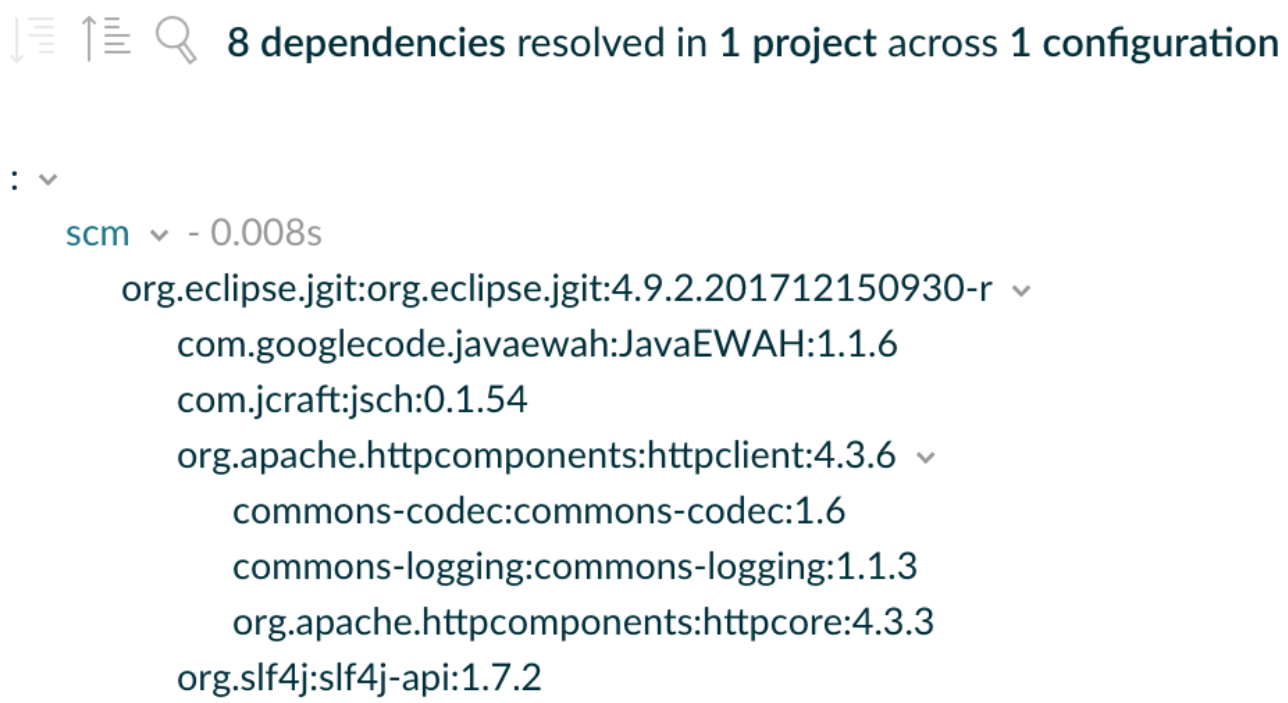


Figure 19. Dependency tree in a build scan

Listing dependencies in a project

Gradle can visualize the whole dependency tree for every configuration available in the project.

Rendering the dependency tree is particularly useful if you'd like to identify which dependencies have been resolved at runtime. It also provides you with information about any dependency conflict resolution that occurred in the process and clearly indicates the selected version. The dependency report always contains declared and transitive dependencies.

NOTE	The `dependencies` task selector will only execute the `dependencies` task on a *single* project. If you run the task on the root project, it will show dependencies of the root project and not of any subproject. Be sure to always target the right project when running `dependencies`.

Let's say you'd want to create tasks for your project that use the JGit library to execute SCM operations e.g. to model a release process. You can declare dependencies for any external tooling with the help of a custom configuration so that it doesn't pollute other contexts like the compilation classpath for your production source code.

Every Gradle project provides the task `dependencies` to render the so-called *dependency report* from the command line. By default the dependency report renders dependencies for all configurations. To focus on the information about one configuration, provide the optional parameter `--configuration`.

For example, to show dependencies that would be on the test runtime classpath in a Java project, run:

```
gradle -q dependencies --configuration testRuntimeClasspath
```

TIP Just like with project and task names, you can use abbreviated names to select a configuration. For example, you can specify `tRC` instead of `testRuntimeClasspath` if the pattern matches to a single configuration.

TIP To see a list of all the pre-defined configurations added by the `java` plugin, see the documentation for the Java Plugin.

Example 300. Declaring the JGit dependency with a custom configuration

build.gradle

```
repositories {
    mavenCentral()
}

configurations {
    scm
}

dependencies {
    scm 'org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r'
}
```

build.gradle.kts

```
repositories {
    mavenCentral()
}

configurations {
    create("scm")
}

dependencies {
    "scm"("org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r")
}
```

Example: Rendering the dependency report for a custom configuration

Output of `gradle -q dependencies --configuration scm`

```
> gradle -q dependencies --configuration scm

------------------------------------------------------------
Root project 'dependencies-report'
------------------------------------------------------------

scm
\--- org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r
     +--- com.jcraft:jsch:0.1.54
     +--- com.googlecode.javaewah:JavaEWAH:1.1.6
     +--- org.apache.httpcomponents:httpclient:4.3.6
     |    +--- org.apache.httpcomponents:httpcore:4.3.3
     |    +--- commons-logging:commons-logging:1.1.3
     |    \--- commons-codec:commons-codec:1.6
     \--- org.slf4j:slf4j-api:1.7.2

A web-based, searchable dependency report is available by adding the --scan option.
```

The dependencies report provides detailed information about the dependencies available in the graph. Any dependency that could not be resolved is marked with `FAILED` in red color. Dependencies with the same coordinates that can occur multiple times in the graph are omitted and indicated by an asterisk. Dependencies that had to undergo conflict resolution render the requested and selected version separated by a right arrow character.

Identifying which dependency version was selected and why

Large software projects inevitably deal with an increased number of dependencies either through direct or transitive dependencies. The dependencies report provides you with the raw list of dependencies but does not explain *why* they have been selected or *which* dependency is responsible for pulling them into the graph.

Let's have a look at a concrete example. A project may request two different versions of the same dependency either as direct or transitive dependency. Gradle applies version conflict resolution to ensure that only one version of the dependency exists in the dependency graph. In this example the conflicting dependency is represented by `commons-codec:commons-codec`.

Example 301. Declaring the JGit dependency and a conflicting dependency

build.gradle

```
repositories {
    mavenCentral()
}

configurations {
    scm
}

dependencies {
    scm 'org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r'
    scm 'commons-codec:commons-codec:1.7'
}
```

build.gradle.kts

```
repositories {
    mavenCentral()
}

configurations {
    create("scm")
}

dependencies {
    "scm"("org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r")
    "scm"("commons-codec:commons-codec:1.7")
}
```

The dependency tree in a build scan renders the selection reason (conflict resolution) as well as the origin of a dependency if you click on a dependency and select the "Required By" tab.

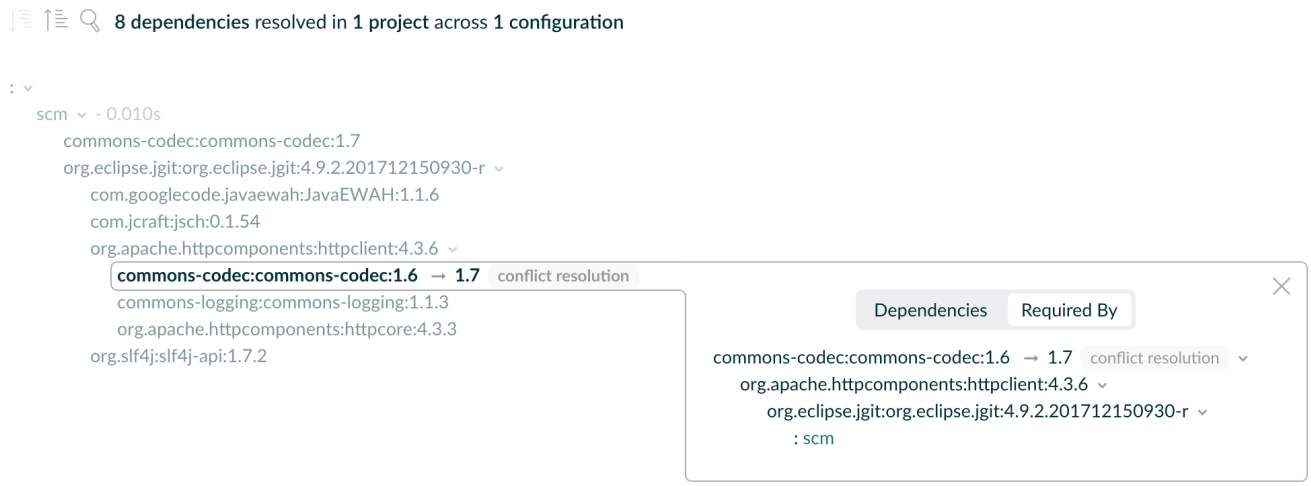


Figure 20. Dependency insight capabilities in a build scan

Every Gradle project provides the task `dependencyInsight` to render the so-called *dependency insight report* from the command line. Given a dependency in the dependency graph you can identify the selection reason and track down the origin of the dependency selection. You can think of the dependency insight report as the inverse representation of the dependency report for a given dependency.

The task takes the following parameters:

`--dependency <dependency>` **(mandatory)**

Indicates which dependency to focus on. It can be a complete `group:name`, or part of it. If multiple dependencies match, they are all printed in the report.

`--configuration <name>` **(mandatory)**

Indicates which configuration to resolve for showing the dependency information (*camelCase* also supported like in `dependencies` task). Note that the Java plugin wires a convention with the value `compileClasspath`, making the parameter optional.

`--single-path` **(optional)**

Indicates to render only a single path to the dependency. This might be useful to trim down the output in large graphs.

NOTE	The `dependencyInsight` task selector will only execute the `dependencyInsight` task on a *single* project. If you run the task on the root project, it will show the dependency information of the root project and not of any subproject. Be sure to always target the right project when running `dependencyInsight`.

Example: Using the dependency insight report for a given dependency

Output of `gradle -q dependencyInsight --dependency commons-codec --configuration scm`

```
> gradle -q dependencyInsight --dependency commons-codec --configuration scm
commons-codec:commons-codec:1.7
  Variant default:
    | Attribute Name    | Provided | Requested |
    |-------------------|----------|-----------|
    | org.gradle.status | release  |           |
   Selection reasons:
      - By conflict resolution: between versions 1.7 and 1.6

commons-codec:commons-codec:1.7
\--- scm

commons-codec:commons-codec:1.6 -> 1.7
\--- org.apache.httpcomponents:httpclient:4.3.6
     \--- org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r
          \--- scm

A web-based, searchable dependency report is available by adding the --scan option.
```

As indicated above, omitting the `--configuration` parameter in a project that is not a Java project will lead to an error:

```
> Dependency insight report cannot be generated because the input configuration was
not specified.
  It can be specified from the command line, e.g: ':dependencyInsight --configuration
someConf --dependency someDep'
```

For more information about configurations, see the documentation on declaring dependencies, which describes what dependency configurations are.

Understanding selection reasons

The "Selection reasons" part of the dependency insight report will list the different reasons as to why a dependency was selected. Have a look at the table below to understand the meaning of the different terms used:

Table 14. Selections reasons terminology

Reason	Meaning
(Absent)	This means that no other reason than having a reference, direct or transitive, was present
Was requested : <text>	The dependency appears in the graph, and the inclusion came with a `because` text.
Was requested : didn't match versions <versions>	The dependency appears in the graph, with a dynamic version, which did not include the listed versions. This can also be followed by a `because` text.

Reason	Meaning
Was requested : reject version <versions>	The dependency appears in the graph, with a rich version containing one or more `reject`. This can also be followed by a `because` text.
By conflict resolution : between versions <version>	The dependency appeared multiple times in the graph, with different version requests. This resulted in conflict resolution to select the most appropriate version.
By constraint	A dependency constraint participated in the version selection. This can also be followed by a `because` text.
By ancestor	There is a rich version with a `strictly` in the graph which enforces the version of this dependency.
Selected by rule	A dependency resolution rule overruled the default selection process. This can also be followed by a `because` text.
Rejection : <version> by rule because <text>	A `ComponentSelection.reject` rejected the given version of the dependency
Rejection: version <version>: <attributes information>	The dependency has a dynamic version, and some versions did not match the requested attributes.
Forced	The build enforces the version of the dependency.

Note that if multiple selection reasons exist in the graph, they will all be listed.

Resolving version conflicts

If the selected version does not match your expectation, Gradle offers a series of tools to help you control transitive dependencies.

Resolving variant selection errors

Sometimes a selection error will happen at the variant selection level. Have a look at the dedicated section to understand these errors and how to resolve them.

Resolving unsafe configuration resolution errors

Resolving a configuration can have side effects on Gradle's project model, so Gradle needs manage access to each project's configurations. There are a number of ways a configuration might be resolved unsafely. Gradle will produce a deprecation warning for each unsafe access. Each of these are bad practices and can cause strange and indeterminate errors.

If your build has an unsafe access deprecation warning, it needs to be fixed.

For example:

- A task from one project directly resolves a configuration in another project in the task's action.
- A task specifies a configuration from another project as an input file collection.
- A build script for one project resolves a configuration in another project during evaluation.

- Project configurations are resolved in the settings file.

In most cases, this issue can be resolved by creating a cross-project dependency on the other project. See the documentation for sharing outputs between projects for more information.

If you find a use case that can't be resolved using these techniques, please let us know by filing a GitHub Issue adhering to our issue guidelines.

Understanding dependency resolution

This chapter covers the way dependency resolution works *inside* Gradle. After covering how you can declare repositories and dependencies, it makes sense to explain how these declarations come together during dependency resolution.

Dependency resolution is a process that consists of two phases, which are repeated until the dependency graph is complete:

- When a new dependency is added to the graph, perform conflict resolution to determine which version should be added to the graph.
- When a specific dependency, that is a module with a version, is identified as part of the graph, retrieve its metadata so that its dependencies can be added in turn.

The following section will describe what Gradle identifies as conflicts and how it can resolve them automatically. After that, the retrieval of metadata will be covered, explaining how Gradle can follow dependency links.

How Gradle handles conflicts?

When doing dependency resolution, Gradle handles two types of conflicts:

Version conflicts

That is when two or more dependencies require a given dependency but with different versions.

Implementation conflicts

That is when the dependency graph contains multiple modules that provide the same implementation, or capability in Gradle terminology.

The following sections will explain in detail how Gradle attempts to resolve these conflicts.

The dependency resolution process is highly customizable to meet enterprise requirements. For more information, see the chapter on Controlling transitive dependencies.

Version conflict resolution

A version conflict occurs when two components:

- Depend on the same module, let's say `com.google.guava:guava`
- But on different versions, let's say `20.0` and `25.1-android`
  - Our project itself depends on `com.google.guava:guava:20.0`

- Our project also depends on `com.google.inject:guice:4.2.2` which itself depends on `com.google.guava:guava:25.1-android`

Resolution strategy

Given the conflict above, there exist multiple ways to handle it, either by selecting a version or failing the resolution. Different tools that handle dependency management have different ways of handling these type of conflicts.

NOTE

Apache Maven uses a nearest first strategy.

Maven will take the *shortest* path to a dependency and use that version. In case there are multiple paths of the same length, the first one wins.

This means that in the example above, the version of `guava` will be `20.0` because the direct dependency is *closer* than the `guice` dependency.

The main drawback of this method is that it is ordering dependent. Keeping order in a very large graph can be a challenge. For example, what if the new version of a dependency ends up having its own dependency declarations in a different order than the previous version?

With Maven, this could have unwanted impact on resolved versions.

NOTE

Apache Ivy is a very flexible dependency management tool. It offers the possibility to customize dependency resolution, including conflict resolution.

This flexibility comes with the price of making it hard to reason about.

Gradle will consider *all* requested versions, wherever they appear in the dependency graph. Out of these versions, it will select the *highest* one. More information on version ordering here.

As you have seen, Gradle supports a concept of rich version declaration, so what is the highest version depends on the way versions were declared:

- If no ranges are involved, then the highest version that is not rejected will be selected.
  - If a version declared as `strictly` is lower than that version, selection will fail.
- If ranges are involved:
  - If there is a non range version that falls within the specified ranges or is higher than their upper bound, it will be selected.
  - If there are only ranges, the selection will depend on the intersection of ranges:
    - If all the ranges intersect, then the highest *existing* version of the intersection will be selected.
    - If there is no clear intersection between all the ranges, the highest *existing* version will be selected from the highest range. If there is no version available for the highest range, the resolution will fail.
  - If a version declared as `strictly` is lower than that version, selection will fail.

Note that in the case where ranges come into play, Gradle requires metadata to determine which versions do exist for the considered range. This causes an intermediate lookup for metadata, as described in How Gradle retrieves dependency metadata?.

Qualifiers

There is a caveat to comparing versions when it comes to selecting the *highest* one. All the rules of version ordering still apply, but the conflict resolver has a bias towards versions without qualifiers.

The "qualifier" of a version, if it exists, is the tail end of the version string, starting at the first non-dot separator found in it. The other (first) part of the version string is called the "base form" of the version. Here are some examples to illustrate:

Original version	Base version	Qualifier
1.2.3	1.2.3	<none>
1.2-3	1.2	3
1_alpha	1	alpha
abc	abc	<none>
1.2b3	1.2	b3
abc.1+3	abc.1	3
b1-2-3.3	b	2-3.3

As you can see separators are any of the `.`, `-`, `_`, `+` characters, plus the empty string when a numeric and a non-numeric part of the version are next to each-other.

When resolving the conflict between competing versions, the following logic applies: * first the versions with the highest base version are selected, the rest are discarded * if there are still multiple competing versions left, then one is picked with a preference for not having a qualifier or having release status.

Implementation conflict resolution

Gradle uses variants and capabilities to identify what a module *provides*.

This is a unique feature that deserves its own chapter to understand what it means and enables.

A conflict occurs the moment two modules either:

- Attempt to select incompatible variants,
- Declare the same capability

Learn more about handling these type of conflicts in Selecting between candidates.

How Gradle retrieves dependency metadata?

Gradle requires metadata about the modules included in your dependency graph. That information is required for two main points:

- Determine the existing versions of a module when the declared version is dynamic.
- Determine the dependencies of the module for a given version.

Discovering versions

Faced with a dynamic version, Gradle needs to identify the concrete matching versions:

- Each repository is inspected, Gradle does not stop on the first one returning some metadata. When multiple are defined, they are inspected *in the order they were added*.
- For Maven repositories, Gradle will use the `maven-metadata.xml` which provides information about the available versions.
- For Ivy repositories, Gradle will resort to directory listing.

This process results in a list of candidate versions that are then matched to the dynamic version expressed. At this point, version conflict resolution is resumed.

Note that Gradle caches the version information, more information can be found in the section Controlling dynamic version caching.

Obtaining module metadata

Given a required dependency, with a version, Gradle attempts to resolve the dependency by searching for the module the dependency points at.

- Each repository is inspected in order.
  - Depending on the type of repository, Gradle looks for metadata files describing the module (`.module`, `.pom` or `ivy.xml` file) or directly for artifact files.
  - Modules that have a module metadata file (`.module`, `.pom` or `ivy.xml` file) are preferred over modules that have an artifact file only.
  - Once a repository returns a *metadata* result, following repositories are ignored.
- Metadata for the dependency is retrieved and parsed, if found
  - If the module metadata is a POM file that has a parent POM declared, Gradle will recursively attempt to resolve each of the parent modules for the POM.
- All of the artifacts for the module are then requested from the *same repository* that was chosen in the process above.
- All of that data, including the repository source and potential misses are then stored in the The Dependency Cache.

NOTE	The penultimate point above is what can make the integration with Maven Local problematic. As it is a cache for Maven, it will sometimes miss some artifacts of a given module. If Gradle is sourcing such a module from Maven Local, it will consider the missing artifacts to be missing altogether.

Repository disabling

When Gradle fails to retrieve information from a repository, it will disable it for the duration of the

build and fail all dependency resolution.

That last point is important for reproducibility. If the build was allowed to continue, ignoring the faulty repository, subsequent builds could have a different result once the repository is back online.

HTTP Retries

Gradle will make several attempts to connect to a given repository before disabling it. If connection fails, Gradle will retry on certain errors which have a chance of being transient, increasing the amount of time waiting between each retry.

Blacklisting happens when the repository cannot be contacted, either because of a permanent error or because the maximum retries was reached.

The Dependency Cache

Gradle contains a highly sophisticated dependency caching mechanism, which seeks to minimise the number of remote requests made in dependency resolution, while striving to guarantee that the results of dependency resolution are correct and reproducible.

The Gradle dependency cache consists of two storage types located under `GRADLE_USER_HOME/caches`:

- A file-based store of downloaded artifacts, including binaries like jars as well as raw downloaded meta-data like POM files and Ivy files. The storage path for a downloaded artifact includes the SHA1 checksum, meaning that 2 artifacts with the same name but different content can easily be cached.
- A binary store of resolved module metadata, including the results of resolving dynamic versions, module descriptors, and artifacts.

The Gradle cache does not allow the local cache to hide problems and create other mysterious and difficult to debug behavior. Gradle enables reliable and reproducible enterprise builds with a focus on bandwidth and storage efficiency.

Separate metadata cache

Gradle keeps a record of various aspects of dependency resolution in binary format in the metadata cache. The information stored in the metadata cache includes:

- The result of resolving a dynamic version (e.g. `1.+`) to a concrete version (e.g. `1.2`).
- The resolved module metadata for a particular module, including module artifacts and module dependencies.
- The resolved artifact metadata for a particular artifact, including a pointer to the downloaded artifact file.
- The *absence* of a particular module or artifact in a particular repository, eliminating repeated attempts to access a resource that does not exist.

Every entry in the metadata cache includes a record of the repository that provided the information as well as a timestamp that can be used for cache expiry.

Repository caches are independent

As described above, for each repository there is a separate metadata cache. A repository is identified by its URL, type and layout. If a module or artifact has not been previously resolved from *this repository*, Gradle will attempt to resolve the module against the repository. This will always involve a remote lookup on the repository, however in many cases no download will be required.

Dependency resolution will fail if the required artifacts are not available in any repository specified by the build, even if the local cache has a copy of this artifact which was retrieved from a different repository. Repository independence allows builds to be isolated from each other in an advanced way that no build tool has done before. This is a key feature to create builds that are reliable and reproducible in any environment.

Artifact reuse

Before downloading an artifact, Gradle tries to determine the checksum of the required artifact by downloading the sha file associated with that artifact. If the checksum can be retrieved, an artifact is not downloaded if an artifact already exists with the same id and checksum. If the checksum cannot be retrieved from the remote server, the artifact will be downloaded (and ignored if it matches an existing artifact).

As well as considering artifacts downloaded from a different repository, Gradle will also attempt to reuse artifacts found in the local Maven Repository. If a candidate artifact has been downloaded by Maven, Gradle will use this artifact if it can be verified to match the checksum declared by the remote server.

Checksum based storage

It is possible for different repositories to provide a different binary artifact in response to the same artifact identifier. This is often the case with Maven SNAPSHOT artifacts, but can also be true for any artifact which is republished without changing its identifier. By caching artifacts based on their SHA1 checksum, Gradle is able to maintain multiple versions of the same artifact. This means that when resolving against one repository Gradle will never overwrite the cached artifact file from a different repository. This is done without requiring a separate artifact file store per repository.

Cache Locking

The Gradle dependency cache uses file-based locking to ensure that it can safely be used by multiple Gradle processes concurrently. The lock is held whenever the binary metadata store is being read or written, but is released for slow operations such as downloading remote artifacts.

This concurrent access is only supported if the different Gradle processes can communicate together. This is usually *not the case* for containerized builds.

Cache Cleanup

Gradle keeps track of which artifacts in the dependency cache are accessed. Using this information, the cache is periodically (at most every 24 hours) scanned for artifacts that have not been used for more than 30 days. Obsolete artifacts are then deleted to ensure the cache does not grow indefinitely.

Dealing with ephemeral builds

It's a common practice to run builds in ephemeral containers. A container is typically spawned to only execute a single build before it is destroyed. This can become a practical problem when a build depends on a lot of dependencies which each container has to re-download. To help with this scenario, Gradle provides a couple of options:

- copying the dependency cache into each container
- sharing a read-only dependency cache between multiple containers

Copying and reusing the cache

The dependency cache, both the file and metadata parts, are fully encoded using relative paths. This means that it is perfectly possible to copy a cache around and see Gradle benefit from it.

The path that can be copied is `$GRADLE_HOME/caches/modules-<version>`. The only constraint is placing it using the same structure at the destination, where the value of `GRADLE_HOME` can be different.

Do not copy the `*.lock` or `gc.properties` files if they exist.

Note that creating the cache and consuming it should be done using compatible Gradle version, as shown in the table below. Otherwise, the build might still require some interactions with remote repositories to complete missing information, which might be available in a different version. If multiple incompatible Gradle versions are in play, all should be used when seeding the cache.

Table 15. Dependency cache compatibility

Module cache version	File cache version	Metadata cache version	Gradle version(s)
`modules-2`	`files-2.1`	`metadata-2.95`	Gradle 6.1 to Gradle 6.3
`modules-2`	`files-2.1`	`metadata-2.96`	Gradle 6.4 to Gradle 6.7
`modules-2`	`files-2.1`	`metadata-2.97`	Gradle 6.8 to Gradle 7.4
`modules-2`	`files-2.1`	`metadata-2.99`	Gradle 7.5 to Gradle 7.6.1
`modules-2`	`files-2.1`	`metadata-2.101`	Gradle 7.6.2 only

Sharing the dependency cache with other Gradle instances

Instead of copying the dependency cache into each container, it's possible to mount a shared, read-only directory that will act as a dependency cache for all containers. This cache, unlike the classical dependency cache, is accessed without locking, making it possible for multiple builds to read from the cache concurrently. It's important that the read-only cache is not written to when other builds may be reading from it.

When using the shared read-only cache, Gradle looks for dependencies (artifacts or metadata) in both the writable cache in the local Gradle user home directory and the shared read-only cache. If a dependency is present in the read-only cache, it will not be downloaded. If a dependency is missing from the read-only cache, it will be downloaded and added to the writable cache. In practice, this means that the writable cache will only contain dependencies that are unavailable in the read-only

cache.

The read-only cache should be sourced from a Gradle dependency cache that already contains some of the required dependencies. The cache can be incomplete; however, an empty shared cache will only add overhead.

NOTE	The shared read-only dependency cache is an incubating feature.

The first step in using a shared dependency cache is to create one by copying of an existing *local* cache. For this you need to follow the instructions above.

Then set the `GRADLE_RO_DEP_CACHE` environment variable to point to the directory containing the cache:

```
$GRADLE_RO_DEP_CACHE
   |-- modules-2 : the read-only dependency cache, should be mounted with read-only
privileges

$GRADLE_HOME
   |-- caches
         |-- modules-2 : the container specific dependency cache, should be writable
         |-- ...
   |-- ...
```

In a CI environment, it's a good idea to have one build which "seeds" a Gradle dependency cache, which is then *copied* to a different directory. This directory can then be used as the read-only cache for other builds. You shouldn't use an existing Gradle installation cache as the read-only cache, because this directory may contain locks and may be modified by the seeding build.

Accessing the resolution result programmatically

While most users only need access to a "flat list" of files, there are cases where it can be interesting to reason on a *graph* and get more information about the resolution result:

- for tooling integration, where a model of the dependency graph is required
- for tasks generating a visual representation (image, `.dot` file, ...) of a dependency graph
- for tasks providing diagnostics (similar to the `dependencyInsight` task)
- for tasks which need to perform dependency resolution at execution time (e.g, download files on demand)

For those use cases, Gradle provides lazy, thread-safe APIs, accessible by calling the Configuration.getIncoming() method:

- the ResolutionResult API gives access to a resolved dependency graph, whether the resolution was successful or not.
- the artifacts API provides a simple access to the resolved artifacts, untransformed, but with lazy download of artifacts (they would only be downloaded on demand).

- the artifact view API provides an advanced, filtered view of artifacts, possibly transformed.

NOTE	See the documentation on using dependency resolution results for more details on how to consume the results in a task.

Verifying dependencies

Working with external dependencies and plugins published on third-party repositories puts your build at risk. In particular, you need to be aware of what binaries are brought in transitively and if they are legit. To mitigate the security risks and avoid integrating compromised dependencies in your project, Gradle supports *dependency verification*.

IMPORTANT	Dependency verification is, by nature, an inconvenient feature to use. It means that whenever you're going to update a dependency, builds are likely to fail. It means that merging branches are going to be harder because each branch can have different dependencies. It means that you will be tempted to switch it off. So why should you bother? Dependency verification is about **trust** in what you get and what you ship. Without dependency verification it's easy for an attacker to compromise your supply chain. There are many real world examples of tools compromised by adding a malicious dependency. Dependency verification is meant to protect yourself from those attacks, by forcing you to ensure that the artifacts you include in your build are the ones that you expect. It is not meant, however, to prevent you from including *vulnerable* dependencies. Finding the right balance between security and convenience is hard but Gradle will try to let you choose the "right level" for you.

Dependency verification consists of two different and complementary operations:

- *checksum verification*, which allows asserting the integrity of a dependency
- *signature verification*, which allows asserting the provenance of a dependency

Gradle supports both checksum and signature verification out of the box but performs no dependency verification by default. This section will guide you into configuring dependency verification properly for your needs.

This feature can be used for:

- detecting compromised dependencies
- detecting compromised plugins
- detecting tampered dependencies in the local dependency caches

Enabling dependency verification

The verification metadata file

NOTE	Currently the only source of dependency verification metadata is this XML configuration file. Future versions of Gradle may include other sources (for example via external services).

Dependency verification is automatically enabled once the configuration file for dependency verification is discovered. This configuration file is located at `$PROJECT_ROOT/gradle/verification-metadata.xml`. This file minimally consists of the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata>
   <configuration>
      <verify-metadata>true</verify-metadata>
      <verify-signatures>false</verify-signatures>
    </configuration>
</verification-metadata>
```

Doing so, Gradle will verify all artifacts using checksums, but will not verify signatures. Gradle will verify any artifact downloaded using its dependency management engine, which includes, but is not limited to:

- artifact files (e.g jar files, zips, ...) used during a build
- metadata artifacts (POM files, Ivy descriptors, Gradle Module Metadata)
- plugins (both project and settings plugins)
- artifacts resolved using the advanced dependency resolution APIs

Gradle will *not* verify changing dependencies (in particular `SNAPSHOT` dependencies) nor locally produced artifacts (typically jars produced during the build itself) as by nature their checksums and signatures would always change.

With such a minimal configuration file, a project using *any* external dependency or plugin would immediately start failing because it doesn't contain any checksum to verify.

Scope of the dependency verification

A dependency verification configuration is *global*: a single file is used to configure verification of the whole build. In particular, the same file is used for both the (sub)projects and `buildSrc`.

If an included build is used:

- the configuration file of the *current* build is used for verification
- so if the included build itself uses verification, its configuration is ignored in favor of the current one
- which means that including a build works similarly to upgrading a dependency: it may require

you to update your current verification metadata

An easy way to get started is therefore to generate the minimal configuration for an existing build.

Configuring the console output

By default, if dependency verification fails, Gradle will generate a small summary about the verification failure as well as an HTML report containing the full information about the failures. If your environment prevents you from reading this HTML report file (for example if you run a build on CI and that it's not easy to fetch the remote artifacts), Gradle provides a way to opt-in a verbose console report. For this, you need to add this Gradle property to your `gradle.properties` file:

```
org.gradle.dependency.verification.console=verbose
```

Bootstrapping dependency verification

It's worth mentioning that while Gradle can generate a dependency verification file for you, you should always check whatever Gradle generated for you because your build may *already* contain compromised dependencies without you knowing about it. Please refer to the appropriate checksum verification or signature verification section for more information.

If you plan on using signature verification, please also read the corresponding section of the docs.

Bootstrapping can either be used to create a file from the beginning, or also to *update* an existing file with new information. Therefore, it's recommended to always use the same parameters once you started bootstrapping.

The dependency verification file can be generated with the following CLI instructions:

```
gradle --write-verification-metadata sha256 help
```

The `write-verification-metadata` flag requires the list of checksums that you want to generate or `pgp` for signatures.

Executing this command line will cause Gradle to:

- resolve all resolvable configurations, which includes:
  - configurations from the root project
  - configurations from all subprojects
  - configurations from `buildSrc`
  - included builds configurations
  - configurations used by plugins
- download all artifacts discovered during resolution
- compute the requested checksums and possibly verify signatures depending on what you asked
- At the end of the build, generate the configuration file which will contain the inferred

verification metadata

As a consequence, the `verification-metadata.xml` file will be used in subsequent builds to verify dependencies.

WARNING

There are dependencies that Gradle *cannot* discover this way. In particular, you will notice that the CLI above uses the `help` task. If you don't specify any task, Gradle will automatically run the default task and generate a configuration file at the end of the build too.

The difference is that Gradle *may* discover more dependencies and artifacts depending on the tasks you execute. As a matter of fact, Gradle cannot automatically discover *detached configurations*, which are basically dependency graphs resolved as an internal implementation detail of the execution of a task: they are not, in particular, declared as an input of the task because they effectively depend on the configuration of the task at execution time.

A good way to start is just to use the simplest task, `help`, which will discover as much as possible, and if subsequent builds fail with a verification error, you can re-execute generation with the appropriate tasks to "discover" more dependencies.

Gradle won't verify either checksums or signatures of plugins which use their own HTTP clients. Only plugins which use the infrastructure provided by Gradle for performing requests will see their requests verified.

Using generation for incremental updates

The verification file generated by Gradle has a strict ordering for all its content. It also uses the information from the existing state to limit changes to the strict minimum.

This means that generation is actually a convenient tool for *updating* a verification file:

- Checksum entries generated by Gradle will have a clear `origin` that starts with "Generated by Gradle", which is a good indicator that an entry needs to be reviewed,
- Entries added by hand will immediately be accounted for, and appear at the right location after writing the file,
- The header comments of the file will be preserved, i.e. comments before the root XML node. This allows you to have a license header or instructions on which tasks and which parameters to use for generating that file.

With the above benefits, it is really easy to account for new dependencies or dependency versions by simply generating the file again and reviewing the changes.

Using dry mode

By default, bootstrapping is incremental, which means that if you run it multiple times, information is *added* to the file and in particular you can rely on your VCS to check the diffs. There are

situations where you would just want to *see* what the generated verification metadata file would look like without actually changing the existing one or overwriting it.

For this purpose, you can just add `--dry-run`:

```
gradle --write-verification-metadata sha256 help --dry-run
```

Then instead of generating the `verification-metadata.xml` file, a *new file* will be generated, called `verification-metadata.dryrun.xml`.

NOTE	Because `--dry-run` doesn't execute tasks, this would be much faster, but it will miss any resolution happening at task execution time.

Disabling metadata verification

By default, Gradle will not only verify artifacts (jars, ...) but also the metadata associated with those artifacts (typically POM files). Verifying this ensures the maximum level of security: metadata files typically tell what transitive dependencies will be included, so a compromised metadata file may cause the introduction of undesired dependencies in the graph. However, because all artifacts are verified, such artifacts would in general easily be discovered by you, because they would cause a checksum verification failure (checksums would be *missing* from verification metadata). Because metadata verification can significantly increase the size of your configuration file, you may therefore want to disable verification of metadata. If you understand the risks of doing so, set the `<verify-metadata>` flag to `false` in the configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata>
   <configuration>
      <verify-metadata>false</verify-metadata>
      <verify-signatures>false</verify-signatures>
    </configuration>
    <!-- the rest of this file doesn't need to declare anything about metadata files
-->
</verification-metadata>
```

Verifying dependency checksums

Checksum verification allows you to ensure the integrity of an artifact. This is the simplest thing that Gradle can do for you to make sure that the artifacts you use are un-tampered.

Gradle supports MD5, SHA1, SHA-256 and SHA-512 checksums. However, only SHA-256 and SHA-512 checksums are considered secure nowadays.

Adding the checksum for an artifact

External components are identified by GAV coordinates, then each of the artifacts by their file names. To declare the checksums of an artifact, you need to add the corresponding section in the verification metadata file. For example, to declare the checksum for Apache PDFBox. The GAV

coordinates are:

- group `org.apache.pdfbox`
- name `pdfbox`
- version `2.0.17`

Using this dependency will trigger the download of 2 different files:

- `pdfbox-2.0.17.jar` which is the main artifact
- `pdfbox-2.0.17.pom` which is the metadata file associated with this artifact

As as consequence, you need to declare the checksums for both of them (unless you disabled metadata verification):

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata>
   <configuration>
      <verify-metadata>true</verify-metadata>
      <verify-signatures>false</verify-signatures>
   </configuration>
   <components>
      <component group="org.apache.pdfbox" name="pdfbox" version="2.0.17">
         <artifact name="pdfbox-2.0.17.jar">
            <sha512 value=
"7e11e54a21c395d461e59552e88b0de0ebaf1bf9d9bcacadf17b240d9bbc29bf6beb8e36896c186fe405d
287f5d517b02c89381aa0fcc5e0aa5814e44f0ab331" origin="PDFBox Official site
(https://pdfbox.apache.org/download.cgi)"/>
         </artifact>
         <artifact name="pdfbox-2.0.17.pom">
            <sha512 value=
"82de436b38faf6121d8d2e71dda06e79296fc0f7bc7aba0766728c8d306fd1b0684b5379c18808ca724bf
91707277eba81eb4fe19518e99e8f2a56459b79742f" origin="Generated by Gradle"/>
         </artifact>
      </component>
   </components>
</verification-metadata>
```

Where to get checksums from?

In general, checksums are published alongside artifacts on public repositories. However, if a dependency is compromised in a repository, it's likely its checksum will be too, so it's a good practice to get the checksum from a different place, usually the website of the library itself.

In fact, it's a good security practice to publish the checksums of artifacts on a *different server* than the server where the artifacts themselves are hosted: it's harder to compromise a library both on the repository *and* the official website.

In the example above, the checksum was published on the website for the JAR, but not the POM file. This is why it's usually easier to let Gradle generate the checksums and verify by reviewing the

generated file carefully.

In this example, not only could we check that the checksum was correct, but we could also find it on the official website, which is why we changed the value of the of `origin` attribute on the `sha512` element from `Generated by Gradle` to `PDFBox Official site`. Changing the `origin` gives users a sense of how trustworthy your build it.

Interestingly, using `pdfbox` will require *much more* than those 2 artifacts, because it will also bring in transitive dependencies. If the dependency verification file only included the checksums for the main artifacts you used, the build would fail with an error like this one:

```
Execution failed for task ':compileJava'.
> Dependency verification failed for configuration ':compileClasspath':
    - On artifact commons-logging-1.2.jar (commons-logging:commons-logging:1.2) in
repository 'MavenRepo': checksum is missing from verification metadata.
    - On artifact commons-logging-1.2.pom (commons-logging:commons-logging:1.2) in
repository 'MavenRepo': checksum is missing from verification metadata.
```

What this indicates is that your build requires `commons-logging` when executing `compileJava`, however the verification file doesn't contain enough information for Gradle to verify the integrity of the dependencies, meaning you need to add the required information to the verification metadata file.

See troubleshooting dependency verification for more insights on what to do in this situation.

What checksums are verified?

If a dependency verification metadata file declares more than one checksum for a dependency, Gradle will verify *all of them* and fail if *any of them fails*. For example, the following configuration would check both the `md5` and `sha1` checksums:

```
<component group="org.apache.pdfbox" name="pdfbox" version="2.0.17">
   <artifact name="pdfbox-2.0.17.jar">
      <md5 value="c713a8e252d0add65e9282b151adf6b4" origin="official site"/>
      <sha1 value="b5c8dff799bd967c70ccae75e6972327ae640d35" origin="official site"/>
   </artifact>
</component>
```

There are multiple reasons why you'd like to do so:

1. an official site doesn't publish *secure* checksums (SHA-256, SHA-512) but publishes multiple insecure ones (MD5, SHA1). While it's easy to fake a MD5 checksum and hard but possible to fake a SHA1 checksum, it's harder to fake both of them for the same artifact.
2. you might want to add generated checksums to the list above
3. when *updating* dependency verification file with more secure checksums, you don't want to accidentally erase checksums

Verifying dependency signatures

In addition to checksums, Gradle supports verification of signatures. Signatures are used to assess the *provenance* of a dependency (it tells who signed the artifacts, which usually corresponds to who produced it).

As enabling signature verification usually means a higher level of security, you might want to replace checksum verification with signature verification.

WARNING	Signatures *can* also be used to assess the integrity of a dependency similarly to checksums. Signatures are signatures of the *hash* of artifacts, not artifacts themselves. This means that if the signature is done on an *unsafe hash* (even SHA1), then you're not correctly assessing the *integrity* of a file. For this reason, if you care about both, you need to add both signatures *and* checksums to your verification metadata.

However:

- Gradle only supports verification of signatures published on remote repositories as ASCII-armored PGP files
- Not all artifacts are published with signatures
- A good signature doesn't mean that the signatory was legit

As a consequence, signature verification will often be used alongside checksum verification.

NOTE	*About expired keys* It's very common to find artifacts which are signed with an expired key. This is not a problem for *verification*: key expiry is mostly used to avoid signing with a stolen key. If an artifact was signed before expiry, it's still valid.

Enabling signature verification

Because verifying signatures is more expensive (both I/O and CPU wise) and harder to check manually, it's not enabled by default.

Enabling it requires you to change the configuration option in the `verification-metadata.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata>
   <configuration>
      <verify-signatures>true</verify-signatures>
   </configuration>
</verification-metadata>
```

Understanding signature verification

Once signature verification is enabled, for each artifact, Gradle will:

- try to download the corresponding `.asc` file
- if it's present
  - automatically download the keys required to perform verification of the signature
  - verify the artifact using the downloaded public keys
  - if signature verification passes, perform additional requested checksum verification
- if it's absent, fallback to checksum verification

That is to say that Gradle's verification mechanism is much stronger if signature verification is enabled than just with checksum verification. In particular:

- if an artifact is signed with multiple keys, all of them must pass validation or the build will fail
- if an artifact passes verification, any additional checksum configured for the artifact *will also be checked*

However, it's not because an artifact passes signature verification that you can trust it: you need to *trust the keys*.

In practice, it means you need to list the keys that you trust for each artifact, which is done by adding a `pgp` entry instead of a `sha1` for example:

```
<component group="com.github.javaparser" name="javaparser-core" version="3.6.11">
   <artifact name="javaparser-core-3.6.11.jar">
      <pgp value="8756c4f765c9ac3cb6b85d62379ce192d401ab61"/>
   </artifact>
</component>
```

WARNING

For the `pgp` and `trusted-key` elements, Gradle *requires* full fingerprint IDs (e.g. `b801e2f8ef035068ec1139cc29579f18fa8fd93b` instead of a long ID `29579f18fa8fd93b`) . This minimizes the chance of a collision attack.

At the time, V4 key fingerprints are of 160-bit (40 characters) length. We accept longer keys to be future-proof in case a longer key fingerprint is introduced.

In `ignore-key` elements, either fingerprints or long (64-bit) IDs can be used. A shorter ID can only result in a bigger range of exclusion, therefore, it's safe to use.

This effectively means that you trust `com.github.javaparser:javaparser-core:3.6.11` if it's signed with the key `8756c4f765c9ac3cb6b85d62379ce192d401ab61`.

Without this, the build would fail with this error:

```
> Dependency verification failed for configuration ':compileClasspath':
    - On artifact javaparser-core-3.6.11.jar (com.github.javaparser:javaparser-
core:3.6.11) in repository 'MavenRepo': Artifact was signed with key
'8756c4f765c9ac3cb6b85d62379ce192d401ab61' (Bintray (by JFrog) <****>) and passed
verification but the key isn't in your trusted keys list.
```

NOTE The key IDs that Gradle shows in error messages are the key IDs found in the signature file it tries to verify. It doesn't mean that it's necessarily the keys that you should trust. In particular, if the signature is correct but done by a malicious entity, Gradle wouldn't tell you.

Trusting keys globally

Signature verification has the advantage that it can make the configuration of dependency verification easier by not having to explicitly list all artifacts like for checksum verification only. In fact, it's common that the same key can be used to sign several artifacts. If this is the case, you can move the trusted key from the artifact level to the global configuration block:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata>
   <configuration>
      <verify-metadata>true</verify-metadata>
      <verify-signatures>true</verify-signatures>
      <trusted-keys>
         <trusted-key id="8756c4f765c9ac3cb6b85d62379ce192d401ab61" group=
"com.github.javaparser"/>
      </trusted-keys>
   </configuration>
   <components/>
</verification-metadata>
```

The configuration above means that for any artifact belonging to the group `com.github.javaparser`, we trust it if it's signed with the `8756c4f765c9ac3cb6b85d62379ce192d401ab61` fingerprint.

The `trusted-key` element works similarly to the trusted-artifact element:

- `group`, the group of the artifact to trust
- `name`, the name of the artifact to trust
- `version`, the version of the artifact to trust
- `file`, the name of the artifact *file* to trust
- `regex`, a boolean saying if the `group`, `name`, `version` and `file` attributes need to be interpreted as regular expressions (defaults to `false`)

WARNING

You should be careful when trusting a key globally: try to limit it to the appropriate groups or artifacts:

- a valid key may have been used to sign artifact `A` which you trust
- later on, the key is stolen and used to sign artifact `B`

It means you can trust the key `A` for the first artifact, probably only up to the released version before the key was stolen, but not for `B`.

Remember that anybody can put an arbitrary name when generating a PGP key, so never trust the key solely based on the key name. Verify if the key is listed at the official site. For example, Apache projects typically provide a KEYS.txt file that you can trust.

Specifying key servers and ignoring keys

Gradle will automatically download the public keys required to verify a signature. For this it uses a list of well known and trusted key servers (the list may change between Gradle versions, please refer to the implementation to figure out what servers are used by default).

You can explicitly set the list of key servers that you want to use by adding them to the configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata>
   <configuration>
      <verify-metadata>true</verify-metadata>
      <verify-signatures>true</verify-signatures>
      <key-servers>
         <key-server uri="hkp://my-key-server.org"/>
         <key-server uri="https://my-other-key-server.org"/>
      </key-servers>
   </configuration>
</verification-metadata>
```

Despite this, it’s possible that a key is not available:

- because it wasn’t published to a public key server
- because it was lost

In this case, you can ignore a key in the configuration block:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata>
   <configuration>
      <verify-metadata>true</verify-metadata>
      <verify-signatures>true</verify-signatures>
      <ignored-keys>
         <ignored-key id="abcdef1234567890" reason="Key is not available in any key
server"/>
      </ignored-keys>
   </configuration>
</verification-metadata>
```

As soon as a key is ignored, it will not be used for verification, even if the signature file mentions it. However, if the signature cannot be verified with at least one other key, Gradle will mandate that you provide a checksum.

Exporting keys for faster verification

Gradle automatically downloads the required keys but this operation can be quite slow and requires everyone to download the keys. To avoid this, Gradle offers the ability to use a local keyring file containing the required public keys.

Gradle supports 2 different file formats for keyrings: a binary format (`.gpg` file) and a plain text format (`.keys`).

There are pros and cons for each of the formats: the binary format is more compact and can be updated directly via GPG commands, but is completely opaque (binary). On the opposite, the plain text format is human readable, can be easily updated by hand and makes it easier to do code reviews thanks to readable diffs.

If the `gradle/verification-keyring.gpg` or `gradle/verification-keyring.keys` file is present, Gradle will search for keys there in priority.

Note that the plain text file will be ignored if there's already a `.gpg` file (the binary version takes precedence).

You can generate the binary version using GPG, for example issuing the following commands (syntax may depend on the tool you use):

```
$ gpg --no-default-keyring --keyring gradle/verification-keyring.gpg --recv-keys
8756c4f765c9ac3cb6b85d62379ce192d401ab61

gpg: keybox 'gradle/verification-keyring.gpg' created
gpg: key 379CE192D401AB61: public key "Bintray (by JFrog) <****>" imported
gpg: Total number processed: 1
gpg:               imported: 1

$ gpg --no-default-keyring --keyring gradle/verification-keyring.gpg --recv-keys
6f538074ccebf35f28af9b066a0975f8b1127b83

gpg: key 0729A0AFF8999A87: public key "Kotlin Release <****>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

The plain text version, on the other hand, can be updated manually. The file must be formatted with the `US-ASCII` encoding and consists of a list of keys in ASCII armored format.

In the example above, you could amend an existing KEYS file by issuing the following commands:

```
$ gpg --no-default-keyring --keyring /tmp/keyring.gpg --recv-keys
8756c4f765c9ac3cb6b85d62379ce192d401ab61

gpg: keybox '/tmp/keyring.gpg' created
gpg: key 379CE192D401AB61: public key "Bintray (by JFrog) <****>" imported
gpg: Total number processed: 1
gpg:               imported: 1

# First let's add a header so that we can recognize the added key
$ gpg --keyring /tmp/keyring.gpg --list-sigs 8756c4f765c9ac3cb6b85d62379ce192d401ab61
> gradle/verification-keyring.keys

# Then write its ASCII armored version
$ gpg --keyring /tmp/keyring.gpg --export --armor
8756c4f765c9ac3cb6b85d62379ce192d401ab61 > gradle/verification-keyring.keys
```

Or, alternatively, you can *ask Gradle to export all keys it used for verification of this build to the keyring* during bootstrapping:

```
./gradlew --write-verification-metadata pgp,sha256 --export-keys
```

NOTE

This command will generate **both** the binary version and the ASCII armored file. You should only pick one for your project.

It's a good idea to commit this file to VCS (as long as you trust your VCS). If you use git and use the binary version, make sure to make it treat this file as binary, by adding this to your `.gitattributes` file:

```
*.gpg          binary
```

Bootstrapping and signature verification

WARNING

Signature verification bootstrapping takes an *optimistic point of view* that signature verification is *enough*. Therefore, if you also care about *integrity*, you **must** first bootstrap using checksum verification, *then* with signature verification.

Similarly to bootstrapping for checksums, Gradle provides a convenience for bootstrapping a configuration file with signature verification enabled. For this, just add the `pgp` option to the list of verifications to generate. However, because there might be verification failures, missing keys or missing signature files, you **must** provide a fallback checksum verification algorithm:

```
./gradlew --write-verification-metadata pgp,sha256
```

this means that Gradle will verify the signatures and fallback to SHA-256 checksums when there's a problem.

When bootstrapping, Gradle performs *optimistic verification* and therefore assumes a sane build environment. It will therefore:

- automatically add the trusted keys as soon as verification passes
- automatically add ignored keys for keys which couldn't be downloaded from public key servers
- automatically generate checksums for artifacts without signatures or ignored keys

If, for some reason, verification fails during the generation, Gradle will automatically generate an ignored key entry but warn you that you must absolutely check what happens.

This situation is common as explained for this section: a typical case is when the POM file for a dependency differs from one repository to the other (often in a non-meaningful way).

In addition, Gradle will try to group keys automatically and generate the `trusted-keys` block which reduced the configuration file size as much as possible.

Forcing use of local keyrings only

The local keyring files (`.gpg` or `.keys`) can be used to avoid reaching out to key servers whenever a key is required to verify an artifact. However, it may be that the local keyring doesn't contain a key,

in which case Gradle would use the key servers to fetch the missing key. If the local keyring file isn't regularly updated, using key export, then it may be that your CI builds, for example, would reach out to key servers too often (especially if you use disposable containers for builds).

To avoid this, Gradle offers the ability to disallow use of key servers altogether: only the local keyring file would be used, and if a key is missing from this file, the build will fail.

To enable this mode, you need to disable key servers in the configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata>
   <configuration>
      <key-servers enabled="false"/>
      ...
   </configuration>
   ...
</verification-metadata>
```

NOTE	If you are asking Gradle to generate a verification metadata file and that an existing verification metadata file sets `enabled` to `false`, then this flag will be ignored, so that potentially missing keys are downloaded.

Troubleshooting dependency verification

Dealing with a verification failure

Dependency verification can fail in different ways, this section explains how you should deal with the various cases.

Missing verification metadata

The simplest failure you can have is when verification metadata is missing from the dependency verification file. This is the case for example if you use checksum verification, then you update a dependency and new versions of the dependency (and potentially its transitive dependencies) are brought in.

Gradle will tell you what metadata is missing:

```
Execution failed for task ':compileJava'.
> Dependency verification failed for configuration ':compileClasspath':
    - On artifact commons-logging-1.2.jar (commons-logging:commons-logging:1.2) in
repository 'MavenRepo': checksum is missing from verification metadata.
```

- the missing module group is `commons-logging`, it's artifact name is `commons-logging` and its version is `1.2`. The corresponding artifact is `commons-logging-1.2.jar` so you need to add the following entry to the verification file:

```
<component group="commons-logging" name="commons-logging" version="1.2">
   <artifact name="commons-logging-1.2.jar">
      <sha256 value="daddea1ea0be0f56978ab3006b8ac92834afeefbd9b7e4e6316fca57df0fa636"
origin="official distribution"/>
   </artifact>
</component>
```

Alternatively, you can ask Gradle to generate the missing information by using the bootstrapping mechanism: existing information in the metadata file will be preserved, Gradle will only add the missing verification metadata.

Incorrect checksums

A more problematic issue is when the actual checksum verification fails:

```
Execution failed for task ':compileJava'.
> Dependency verification failed for configuration ':compileClasspath':
    - On artifact commons-logging-1.2.jar (commons-logging:commons-logging:1.2) in
repository 'MavenRepo': expected a 'sha256' checksum of
'91f7a33096ea69bac2cbaf6d01feb934cac002c48d8c8cfa9c240b40f1ec21df' but was
'daddea1ea0be0f56978ab3006b8ac92834afeefbd9b7e4e6316fca57df0fa636'
```

This time, Gradle tells you what dependency is at fault, what was the expected checksum (the one you declared in the verification metadata file) and the one which was actually computed during verification.

Such a failure indicates that a **dependency may have been compromised**. At this stage, you **must** perform manual verification and check what happens. Several things can happen:

- a dependency was tampered in the local dependency cache of Gradle. This is usually harmless: erase the file from the cache and Gradle would redownload the dependency.
- a dependency is available in multiple sources with slightly different binaries (additional whitespace, ...)
  - please inform the maintainers of the library that they have such an issue
  - you can use `also-trust` to accept the additional checksums
- the dependency was compromised
  - immediately inform the maintainers of the library
  - notify the repository maintainers of the compromised library

Note that a variation of a compromised library is often *name squatting*, when a hacker would use GAV coordinates which *look legit* but are actually different by one character, or *repository shadowing*, when a dependency with the official GAV coordinates is published in a malicious repository which comes first in your build.

Untrusted signatures

If you have signature verification enabled, Gradle will perform verification of the signatures but will not trust them automatically:

```
> Dependency verification failed for configuration ':compileClasspath':
    - On artifact javaparser-core-3.6.11.jar (com.github.javaparser:javaparser-core:3.6.11) in repository 'MavenRepo': Artifact was signed with key '379ce192d401ab61' (Bintray (by JFrog) <****>) and passed verification but the key isn't in your trusted keys list.
```

In this case it means you need to check yourself if the key that was used for verification (and therefore the signature) can be trusted, in which case refer to this section of the documentation to figure out how to declare trusted keys.

Failed signature verification

If Gradle fails to verify a signature, you will need to take action and verify artifacts manually because this **may indicate a compromised dependency**.

If such a thing happens, Gradle will fail with:

```
> Dependency verification failed for configuration ':compileClasspath':
    - On artifact javaparser-core-3.6.11.jar (com.github.javaparser:javaparser-core:3.6.11) in repository 'MavenRepo': Artifact was signed with key '379ce192d401ab61' (Bintray (by JFrog) <****>) but signature didn't match
```

There are several options:

1. signature was wrong in the first place, which happens frequently with dependencies published on different repositories.
2. the signature is correct but the artifact has been compromised (either in the local dependency cache or remotely)

The right approach here is to go to the official site of the dependency and see if they publish signatures for their artifacts. If they do, verify that the signature that Gradle downloaded matches the one published.

If you have checked that the dependency is *not* compromised and that it's "only" the signature which is wrong, you should declare an *artifact level key exclusion*:

```
   <components>
       <component group="com.github.javaparser" name="javaparser-core" version="
3.6.11">
          <artifact name="javaparser-core-3.6.11.pom">
             <ignored-keys>
                <ignored-key id="379ce192d401ab61" reason="internal repo has corrupted
POM"/>
             </ignored-keys>
          </artifact>
       </component>
   </components>
```

However, if you only do so, Gradle will still fail because all keys for this artifact will be ignored and you didn't provide a checksum:

```
   <components>
       <component group="com.github.javaparser" name="javaparser-core" version="
3.6.11">
          <artifact name="javaparser-core-3.6.11.pom">
             <ignored-keys>
                <ignored-key id="379ce192d401ab61" reason="internal repo has corrupted
POM"/>
             </ignored-keys>
             <sha256 value=
"a2023504cfd611332177f96358b6f6db26e43d96e8ef4cff59b0f5a2bee3c1e1"/>
          </artifact>
       </component>
   </components>
```

Manual verification of a dependency

You will likely face a dependency verification failure (either checksum verification or signature verification) and will need to figure out if the dependency has been compromised or not.

In this section we give *an example* how you can manually check if a dependency was compromised.

For this we will take this example failure:

```
> Dependency verification failed for configuration ':compileClasspath':
- On artifact j2objc-annotations-1.1.jar (com.google.j2objc:j2objc-annotations:1.1) in
repository 'MyCompany Mirror': Artifact was signed with key '29579f18fa8fd93b' but
signature didn't match
```

This error message gives us the GAV coordinates of the problematic dependency, as well as an indication of where the dependency was fetched from. Here, the dependency comes from `MyCompany Mirror`, which is a repository declared in our build.

The first thing to do is therefore to download the artifact and its signature manually from the mirror:

```
$ curl https://my-company-mirror.com/repo/com/google/j2objc/j2objc-
annotations/1.1/j2objc-annotations-1.1.jar --output j2objc-annotations-1.1.jar
$ curl https://my-company-mirror.com/repo/com/google/j2objc/j2objc-
annotations/1.1/j2objc-annotations-1.1.jar.asc --output j2objc-annotations-1.1.jar.asc
```

Then we can use the key information provided in the error message to import the key locally:

```
$ gpg --recv-keys B801E2F8EF035068EC1139CC29579F18FA8FD93B
```

And perform verification:

```
$ gpg --verify j2objc-annotations-1.1.jar.asc
gpg: assuming signed data in 'j2objc-annotations-1.1.jar'
gpg: Signature made Thu 19 Jan 2017 12:06:51 AM CET
gpg:                using RSA key 29579F18FA8FD93B
gpg: BAD signature from "Tom Ball <****>" [unknown]
```

What this tells us is that the problem is *not* on the local machine: the repository *already contains a bad signature*.

The next step is to do the same by downloading what is actually on Maven Central:

```
$ curl https://my-company-mirror.com/repo/com/google/j2objc/j2objc-
annotations/1.1/j2objc-annotations-1.1.jar  --output central-j2objc-annotations-
1.1.jar
$ curl https://my-company-mirror.com/repo/com/google/j2objc/j2objc-
annotations/1/1/j2objc-annotations-1.1.jar.asc  --output central-j2objc-annotations-
1.1.jar.asc
```

And we can now check the signature again:

```
$ gpg --verify central-j2objc-annotations-1.1.jar.asc

gpg: assuming signed data in 'central-j2objc-annotations-1.1.jar'
gpg: Signature made Thu 19 Jan 2017 12:06:51 AM CET
gpg:                using RSA key 29579F18FA8FD93B
gpg: Good signature from "Tom Ball <****>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: B801 E2F8 EF03 5068 EC11  39CC 2957 9F18 FA8F D93B
```

This indicates that the dependency is *valid* on Maven Central. At this stage, we already know that

the problem lives in the mirror, it *may* have been compromised, but we need to verify.

A good idea is to compare the 2 artifacts, which you can do with a tool like diffoscope.

We then figure out that the intent wasn't malicious but that somehow a build has been overwritten with a newer version (the version in Central is newer than the one in our repository).

In this case, you can decide to:

- ignore the signature for this artifact and trust the different possible checksums (both for the old artifact and the new version)
- or cleanup your mirror so that it contains the same version as in Maven Central

It's worth noting that if you choose to delete the version from your repository, you will *also* need to remove it from the local Gradle cache.

This is facilitated by the fact the error message tells you were the file is located:

```
> Dependency verification failed for configuration ':compileClasspath':
    - On artifact j2objc-annotations-1.1.jar (com.google.j2objc:j2objc-
annotations:1.1) in repository 'MyCompany Mirror': Artifact was signed with key
'29579f18fa8fd93b' but signature didn't match

  This can indicate that a dependency has been compromised. Please carefully verify
the signatures and checksums.

  For your information here are the path to the files which failed verification:
    - GRADLE_USER_HOME/caches/modules-2/files-2.1/com.google.j2objc/j2objc-
annotations/1.1/976d8d30bebc251db406f2bdb3eb01962b5685b3/j2objc-annotations-1.1.jar
(signature: GRADLE_USER_HOME/caches/modules-2/files-2.1/com.google.j2objc/j2objc-
annotations/1.1/82e922e14f57d522de465fd144ec26eb7da44501/j2objc-annotations-
1.1.jar.asc)

  GRADLE_USER_HOME = /home/jiraya/.gradle
```

You can safely delete the artifact file as Gradle would automatically re-download it:

```
rm -rf ~/.gradle/caches/modules-2/files-2.1/com.google.j2objc/j2objc-annotations/1.1
```

Disabling verification or making it lenient

Dependency verification can be expensive, or sometimes verification could get in the way of day to day development (because of frequent dependency upgrades, for example).

Alternatively, you might want to enable verification on CI servers but not on local machines.

Gradle actually provides 3 different verification modes:

- `strict`, which is the default. Verification fails *as early as possible*, in order to avoid the use of

compromised dependencies during the build.

- `lenient`, which will run the build even if there are verification failures. The verification errors will be displayed during the build without causing a build failure.
- `off` when verification is totally ignored.

All those modes can be activated on the CLI using the `--dependency-verification` flag, for example:

```
./gradlew --dependency-verification lenient build
```

Alternatively, you can set the `org.gradle.dependency.verification` system property, either on the CLI:

```
./gradlew -Dorg.gradle.dependency.verification=lenient build
```

or in a `gradle.properties` file:

```
org.gradle.dependency.verification=lenient
```

Trusting some particular artifacts

You might want to trust some artifacts more than others. For example, it's legitimate to think that artifacts produced in your company and found in your internal repository only are safe, but you want to check every external component.

NOTE	This is a typical *company policy*. In practice, **nothing** prevents your internal repository from being compromised, so it's a good idea to check your internal artifacts too!

For this purpose, Gradle offers a way to automatically trust some artifacts. You can trust all artifacts in a group by adding this to your configuration:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata>
   <configuration>
      <trusted-artifacts>
         <trust group="com.mycompany"/>
      </trusted-artifacts>
   </configuration>
</verification-metadata>
```

This means that all components which group is `com.mycompany` will automatically be trusted. Trusted means that Gradle will not perform any verification whatsoever.

The `trust` element accepts those attributes:

- `group`, the group of the artifact to trust
- `name`, the name of the artifact to trust
- `version`, the version of the artifact to trust
- `file`, the name of the artifact *file* to trust
- `regex`, a boolean saying if the `group`, `name`, `version` and `file` attributes need to be interpreted as regular expressions (defaults to `false`)

In the example above it means that the trusted artifacts would be artifacts in `com.mycompany` but not `com.mycompany.other`. To trust all artifacts in `com.mycompany` and all subgroups, you can use:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata>
   <configuration>
      <trusted-artifacts>
         <trust group="^com[.]mycompany($|([.].*))" regex="true"/>
      </trusted-artifacts>
   </configuration>
</verification-metadata>
```

Trusting multiple checksums for an artifact

It's quite common to have *different checksums for the same artifact* in the wild. How is that possible? Despite progress, it's often the case that developers publish, for example, to Maven Central and another repository separately, using different builds. In general, this is not a problem but sometimes it means that the metadata files would be different (different timestamps, additional whitespaces, ...). Add to this that your build may use several repositories or repository mirrors and it makes it quite likely that a single build can "see" different metadata files for the same component! In general, it's not malicious (but you **must** verify that the artifact is actually correct), so Gradle lets you declare the additional artifact checksums. For example:

```
      <component group="org.apache" name="apache" version="13">
         <artifact name="apache-13.pom">
            <sha256 value=
"2fafa38abefe1b40283016f506ba9e844bfcf18713497284264166a5dbf4b95e">
               <also-trust value=
"ff513db0361fd41237bef4784968bc15aae478d4ec0a9496f811072ccaf3841d"/>
            </sha256>
         </artifact>
      </component>
```

You can have as many `also-trust` entries as needed, but in general you shouldn't have more than 2.

Skipping Javadocs and sources

By default Gradle will verify *all* downloaded artifacts, which includes Javadocs and sources. In general this is not a problem but you might face an issue with IDEs which automatically try to

download them during import: if you didn't set the checksums for those too, importing would fail.

To avoid this, you can configure Gradle to trust automatically all javadocs/sources:

```
<trusted-artifacts>
   <trust file=".*-javadoc[.]jar" regex="true"/>
   <trust file=".*-sources[.]jar" regex="true"/>
</trusted-artifacts>
```

Cleaning up the verification file

If you do nothing, the dependency verification metadata will grow over time as you add new dependencies or change versions: Gradle will not automatically remove *unused* entries from this file. The reason is that there's no way for Gradle to know upfront if a dependency will effectively be used during the build or not.

As a consequence, adding dependencies or changing dependency version can easily lead to more entries in the file, while leaving unnecessary entries out there.

One option to cleanup the file is to move the existing `verification-metadata.xml` file to a different location and call Gradle with the `--dry-run` mode: while not perfect (it will not notice dependencies only resolved at configuration time), it generates *a new file* that you can compare with the existing one.

We need to move the existing file because both the bootstrapping mode and the dry-run mode are incremental: they copy information from the existing metadata verification file (in particular, trusted keys).

Refreshing missing keys

Gradle caches missing keys for 24 hours, meaning it will not attempt to re-download the missing keys for 24 hours after failing.

If you want to retry immediately, you can run with the `--refresh-keys` CLI flag:

```
./gradlew build --refresh-keys
```

Disabling dependency verification for some configurations only

In order to provide the strongest security level possible, dependency verification is enabled globally. This will ensure, for example, that you trust all the plugins you use. However, the plugins themselves may need to resolve additional dependencies that it doesn't make sense to ask the user to accept. For this purpose, Gradle provides an API which allows *disabling dependency verification on some specific configurations*.

WARNING	Disabling dependency verification, if you care about security, is not a good idea. This API mostly exist for cases where it doesn't make sense to check dependencies. However, in order to be on the safe side, Gradle will systematically print a warning whenever verification has been disabled for a specific configuration.

As an example, a plugin may want to check if there are *newer* versions of a library available and list those versions. It doesn't make sense, in this context, to ask the user to put the checksums of the POM files of the newer releases because by definition, they don't know about them. So the plugin might need to run its code *independently of the dependency verification configuration*.

To do this, you need to call the `ResolutionStrategy#disableDependencyVerification` method:

Example 302. Disabling dependency verification

build.gradle

```
configurations {
    myPluginClasspath {
        resolutionStrategy {
            disableDependencyVerification()
        }
    }
}
```

build.gradle.kts

```
configurations {
    "myPluginClasspath" {
        resolutionStrategy {
            disableDependencyVerification()
        }
    }
}
```

It's also possible to disable verification on detached configurations like in the following example:

Example 303. Disabling dependency verification

build.gradle

```
tasks.register("checkDetachedDependencies") {
    doLast {
        def detachedConf = configurations.detachedConfiguration(dependencies
.create("org.apache.commons:commons-lang3:3.3.1"))
        detachedConf.resolutionStrategy.disableDependencyVerification()
        println(detachedConf.files)
    }
}
```

build.gradle.kts

```
tasks.register("checkDetachedDependencies") {
    doLast {
        val detachedConf =
configurations.detachedConfiguration(dependencies.create("org.apache.commons:
commons-lang3:3.3.1"))
        detachedConf.resolutionStrategy.disableDependencyVerification()
        println(detachedConf.files)
    }
}
```

Declaring Versions

Declaring Versions and Ranges

The simplest version declaration is a *simple string* representing the version to use. Gradle supports different ways of declaring a version string:

- An exact version: e.g. `1.3`, `1.3.0-beta3`, `1.0-20150201.131010-1`
- A Maven-style version range: e.g. `[1.0,)`, `[1.1, 2.0)`, `(1.2, 1.5]`
  - The `[` and `]` symbols indicate an inclusive bound; `(` and `)` indicate an exclusive bound.
  - When the upper or lower bound is missing, the range has no upper or lower bound.
  - The symbol `]` can be used instead of `(` for an exclusive lower bound, and `[` instead of `)` for exclusive upper bound. e.g `]1.0, 2.0[`
  - An upper bound exclude acts as a prefix exclude. This means that `[1.0, 2.0[` will also exclude all versions starting with `2.0` that are smaller than `2.0`. For example versions like `2.0-dev1` or `2.0-SNAPSHOT` are no longer included in the range.
- A *prefix* version range: e.g. `1.+`, `1.3.+`

    - Only versions exactly matching the portion before the `+` are included.
    - The range `+` on it's own will include any version.
- A `latest-status` version: e.g. `latest.integration`, `latest.release`
    - Will match the highest versioned module with the specified status. See ComponentMetadata.getStatus().
- A Maven `SNAPSHOT` version identifier: e.g. `1.0-SNAPSHOT`, `1.4.9-beta1-SNAPSHOT`

Version ordering

Versions have an implicit ordering. Version ordering is used to:

- Determine if a particular version is included in a range.
- Determine which version is 'newest' when performing conflict resolution (watch out though, conflict resolution uses "base versions").

Versions are ordered based on the following rules:

- Each version is split into it's constituent "parts":
    - The characters `[. - _ +]` are used to separate the different "parts" of a version.
    - Any part that contains both digits and letters is split into separate parts for each: `1a1 == 1.a.1`
    - Only the parts of a version are compared. The actual separator characters are not significant: `1.a.1 == 1-a+1 == 1.a-1 == 1a1` (watch out though, in the context of conflict resolution there are exceptions to this rule).
- The equivalent parts of 2 versions are compared using the following rules:
    - If both parts are numeric, the highest numeric value is **higher**: `1.1` < `1.2`
    - If one part is numeric, it is considered **higher** than the non-numeric part: `1.a` < `1.1`
    - If both are non-numeric, the parts are compared **alphabetically**, in a **case-sensitive** manner: `1.A` < `1.B` < `1.a` < `1.b`
    - A version with an extra numeric part is considered **higher** than a version without (even when it's zero): `1.1` < `1.1.0`
    - A version with an extra non-numeric part is considered **lower** than a version without: `1.1.a` < `1.1`
- Certain non-numeric parts have special meaning for the purposes of ordering:
    - `dev` is consider **lower** than any other non-numeric part: `1.0-dev` < `1.0-ALPHA` < `1.0-alpha` < `1.0-rc`.
    - The strings `rc`, `snapshot`, `final`, `ga`, `release` and `sp` are considered **higher** than any other string part (sorted in this order): `1.0-zeta` < `1.0-rc` < `1.0-snapshot` < `1.0-final` < `1.0-ga` < `1.0-release` < `1.0-sp` < `1.0`.
    - These special values are **NOT case sensitive**, as opposed to regular string parts and they do not depend on the separator used around them: `1.0-RC-1` == `1.0.rc.1`

Simple version declaration semantics

When you declare a version using the short-hand notation, for example:

Example 304. A simple declaration

build.gradle

```
dependencies {
    implementation('org.slf4j:slf4j-api:1.7.15')
}
```

build.gradle.kts

```
dependencies {
    implementation("org.slf4j:slf4j-api:1.7.15")
}
```

Then the version is considered a required version which means that it should *minimally* be `1.7.15` but can be upgraded by the engine (optimistic upgrade).

There is, however, a shorthand notation for strict versions, using the `!!` notation:

Example 305. Shorthand notation for strict dependencies

build.gradle

```
dependencies {
    // short-hand notation with !!
    implementation('org.slf4j:slf4j-api:1.7.15!!')
    // is equivalent to
    implementation("org.slf4j:slf4j-api") {
        version {
           strictly '1.7.15'
        }
    }

    // or...
    implementation('org.slf4j:slf4j-api:[1.7, 1.8[!!1.7.25')
    // is equivalent to
    implementation('org.slf4j:slf4j-api') {
        version {
           strictly '[1.7, 1.8['
           prefer '1.7.25'
        }
    }
}
```

build.gradle.kts

```
dependencies {
    // short-hand notation with !!
    implementation("org.slf4j:slf4j-api:1.7.15!!")
    // is equivalent to
    implementation("org.slf4j:slf4j-api") {
        version {
           strictly("1.7.15")
        }
    }

    // or...
    implementation("org.slf4j:slf4j-api:[1.7, 1.8[!!1.7.25")
    // is equivalent to
    implementation("org.slf4j:slf4j-api") {
        version {
           strictly("[1.7, 1.8[")
           prefer("1.7.25")
        }
    }
}
```

A strict version *cannot be upgraded* and overrides whatever transitive dependencies originating from this dependency provide. It is recommended to use ranges for strict versions.

The notation `[1.7, 1.8[!!1.7.25` above is equivalent to:

- strictly `[1.7, 1.8[`
- prefer `1.7.25`

which means that the engine **must** select a version between 1.7 (included) and 1.8 (excluded), and that if no other component in the graph needs a different version, it should *prefer* `1.7.25`.

Declaring a dependency without version

A recommended practice for larger projects is to declare dependencies without versions and use dependency constraints for version declaration. The advantage is that dependency constraints allow you to manage versions of all dependencies, including transitive ones, in one place.

Example 306. Declaring a dependency without version

build.gradle

```
dependencies {
    implementation 'org.springframework:spring-web'
}

dependencies {
    constraints {
        implementation 'org.springframework:spring-web:5.0.2.RELEASE'
    }
}
```

build.gradle.kts

```
dependencies {
    implementation("org.springframework:spring-web")
}

dependencies {
    constraints {
        implementation("org.springframework:spring-web:5.0.2.RELEASE")
    }
}
```

Declaring Rich Versions

Gradle supports a rich model for declaring versions, which allows to combine different level of version information. The terms and their meaning are explained below, from the strongest to the weakest:

`strictly`

Any version not matched by this version notation will be excluded. This is the strongest version declaration. On a declared dependency, a `strictly` can downgrade a version. When on a transitive dependency, it will cause dependency resolution to fail if no version acceptable by this clause can be selected. See overriding dependency version for details. This term supports dynamic versions.

When defined, this overrides any previous `require` declaration and clears previous `reject`.

`require`

Implies that the selected version cannot be lower than what `require` accepts but could be higher through conflict resolution, even if higher has an exclusive higher bound. This is what a direct

dependency translates to. This term supports dynamic versions.

When defined, this overrides any previous `strictly` declaration and clears previous `reject`.

`prefer`

This is a very soft version declaration. It applies only if there is no stronger non dynamic opinion on a version for the module. This term does not support dynamic versions.

Definition can complement `strictly` or `require`.

When defined, this overrides any previous `prefer` declaration and clears previous `reject`.

There is also an additional term outside of the level hierarchy:

`reject`

Declares that specific version(s) are not accepted for the module. This will cause dependency resolution to fail if the only versions selectable are also rejected. This term supports dynamic versions.

The following table illustrates a number of use cases and how to combine the different terms for rich version declaration:

Table 16. Rich version use cases

Which version(s) of this dependency are acceptable?	`strictly`	`require`	`prefer`	`rejects`	**Selection result**
Tested with version `1.5`, believe all future versions should work.		1.5			Any version starting from `1.5`, equivalent of `org:foo:1.5`. An upgrade to `2.4` is accepted.
Tested with `1.5`, soft constraint upgrades according to semantic versioning.		[1.0, 2.0[	1.5		Any version between `1.0` and `2.0`, `1.5` if nobody else cares. An upgrade to `2.4` is accepted. ▯
Tested with `1.5`, but follows semantic versioning.	[1.0, 2.0[		1.5		Any version between `1.0` and `2.0` (exclusive), `1.5` if nobody else cares. Overwrites versions from transitive dependencies. ▯
Same as above, with `1.4` known broken.	[1.0, 2.0[		1.5	1.4	Any version between `1.0` and `2.0` (exclusive) except for `1.4`, `1.5` if nobody else cares. Overwrites versions from transitive dependencies. ▯
No opinion, works with `1.5`.			1.5		`1.5` if no other opinion, any otherwise.
No opinion, prefer latest release.			`latest.release`		The latest release at build time. ▯

Which version(s) of this dependency are acceptable?	`strictly`	`require`	`prefer`	`rejects`	Selection result
On the edge, latest release, no downgrade.		`latest.release`			The latest release at build time. 🔒
No other version than 1.5.	1.5				1.5, or failure if another `strict` or higher `require` constraint disagrees. Overwrites versions from transitive dependencies.
`1.5` or a patch version of it exclusively.	[1.5, 1.6[				Latest `1.5.x` patch release, or failure if another `strict` or higher `require` constraint disagrees. Overwrites versions from transitive dependencies. 🔒

Lines annotated with a lock (🔒) indicate that leveraging dependency locking makes sense in this context. Another concept that relates with rich version declaration is the ability to publish resolved versions instead of declared ones.

Using `strictly`, especially for a library, must be a well thought process as it has an impact on downstream consumers. At the same time, used correctly, it will help consumers understand what combination of libraries do not work together in their context. See overriding dependency version for more information.

NOTE Rich version information will be preserved in the Gradle Module Metadata format. However conversion to Ivy or Maven metadata formats will be lossy. The highest level will be published, that is `strictly` or `require` over `prefer`. In addition, any `reject` will be ignored.

Rich version declaration is accessed through the `version` DSL method on a dependency or constraint declaration which gives access to MutableVersionConstraint.

Example 307. Rich version declaration

build.gradle

```
dependencies {
    implementation('org.slf4j:slf4j-api') {
        version {
            strictly '[1.7, 1.8[']
            prefer '1.7.25'
        }
    }

    constraints {
        implementation('org.springframework:spring-core') {
            version {
                require '4.2.9.RELEASE'
                reject '4.3.16.RELEASE'
            }
        }
    }
}
```

build.gradle.kts

```
dependencies {
    implementation("org.slf4j:slf4j-api") {
        version {
            strictly("[1.7, 1.8[")
            prefer("1.7.25")
        }
    }

    constraints {
        add("implementation", "org.springframework:spring-core") {
            version {
                require("4.2.9.RELEASE")
                reject("4.3.16.RELEASE")
            }
        }
    }
}
```

Handling versions which change over time

There are many situations when you want to use the latest version of a particular module

dependency, or the latest in a range of versions. This can be a requirement during development, or you may be developing a library that is designed to work with a range of dependency versions. You can easily depend on these constantly changing dependencies by using a *dynamic version*. A dynamic version can be either a version range (e.g. `2.+`) or it can be a placeholder for the latest version available e.g. `latest.integration`.

Alternatively, the module you request can change over time even for the same version, a so-called changing version. An example of this type of *changing module* is a Maven `SNAPSHOT` module, which always points at the latest artifact published. In other words, a standard Maven snapshot is a module that is continually evolving, it is a "changing module".

NOTE	Using dynamic versions and changing modules can lead to unreproducible builds. As new versions of a particular module are published, its API may become incompatible with your source code. Use this feature with caution!

Declaring a dynamic version

Projects might adopt a more aggressive approach for consuming dependencies to modules. For example you might want to always integrate the latest version of a dependency to consume cutting edge features at any given time. A *dynamic version* allows for resolving the latest version or the latest version of a version range for a given module.

NOTE	Using dynamic versions in a build bears the risk of potentially breaking it. As soon as a new version of the dependency is released that contains an incompatible API change your source code might stop compiling.

Example 308. Declaring a dependency with a dynamic version

build.gradle

```
plugins {
    id 'java-library'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework:spring-web:5.+'
}
```

build.gradle.kts

```
plugins {
    `java-library`
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.springframework:spring-web:5.+")
}
```

A build scan can effectively visualize dynamic dependency versions and their respective, selected versions.

compileClasspath - 0.819s

org.springframework:spring-web:5.+ → 5.0.2.RELEASE

org.springframework:spring-beans:5.0.2.RELEASE

org.springframework:spring-core:5.0.2.RELEASE

org.springframework:spring-jcl:5.0.2.RELEASE

org.springframework:spring-core:5.0.2.RELEASE

org.springframework:spring-jcl:5.0.2.RELEASE

Figure 21. Dynamic dependencies in build scan

By default, Gradle caches dynamic versions of dependencies for 24 hours. Within this time frame, Gradle does not try to resolve newer versions from the declared repositories. The threshold can be configured as needed for example if you want to resolve new versions earlier.

Declaring a changing version

A team might decide to implement a series of features before releasing a new version of the application or library. A common strategy to allow consumers to integrate an unfinished version of their artifacts early and often is to release a module with a so-called *changing version*. A changing version indicates that the feature set is still under active development and hasn't released a stable version for general availability yet.

In Maven repositories, changing versions are commonly referred to as snapshot versions. Snapshot versions contain the suffix `-SNAPSHOT`. The following example demonstrates how to declare a snapshot version on the Spring dependency.

Example 309. Declaring a dependency with a changing version

build.gradle

```
plugins {
    id 'java-library'
}

repositories {
    mavenCentral()
    maven {
        url 'https://repo.spring.io/snapshot/'
    }
}

dependencies {
    implementation 'org.springframework:spring-web:5.0.3.BUILD-SNAPSHOT'
}
```

build.gradle.kts

```
plugins {
    `java-library`
}

repositories {
    mavenCentral()
    maven {
        url = uri("https://repo.spring.io/snapshot/")
    }
}

dependencies {
    implementation("org.springframework:spring-web:5.0.3.BUILD-SNAPSHOT")
}
```

By default, Gradle caches changing versions of dependencies for 24 hours. Within this time frame, Gradle does not try to resolve newer versions from the declared repositories. The threshold can be configured as needed for example if you want to resolve new snapshot versions earlier.

Gradle is flexible enough to treat any version as changing version e.g. if you wanted to model snapshot behavior for an Ivy module. All you need to do is to set the property ExternalModuleDependency.setChanging(boolean) to `true`.

Controlling dynamic version caching

By default, Gradle caches dynamic versions and changing modules for 24 hours. During that time frame Gradle does not contact any of the declared, remote repositories for new versions. If you want Gradle to check the remote repository more frequently or with every execution of your build, then you will need to change the time to live (TTL) threshold.

NOTE	Using a short TTL threshold for dynamic or changing versions may result in longer build times due to the increased number of HTTP(s) calls.

You can override the default cache modes using command line options. You can also change the cache expiry times in your build programmatically using the resolution strategy.

Controlling dependency caching programmatically

You can fine-tune certain aspects of caching programmatically using the ResolutionStrategy for a configuration. The programmatic approach is useful if you would like to change the settings permanently.

By default, Gradle caches dynamic versions for 24 hours. To change how long Gradle will cache the resolved version for a dynamic version, use:

Example 310. Dynamic version cache control

build.gradle

```
configurations.all {
    resolutionStrategy.cacheDynamicVersionsFor 10, 'minutes'
}
```

build.gradle.kts

```
configurations.all {
    resolutionStrategy.cacheDynamicVersionsFor(10, "minutes")
}
```

By default, Gradle caches changing modules for 24 hours. To change how long Gradle will cache the meta-data and artifacts for a changing module, use:

Example 311. Changing module cache control

build.gradle

```
configurations.all {
    resolutionStrategy.cacheChangingModulesFor 4, 'hours'
}
```

build.gradle.kts

```
configurations.all {
    resolutionStrategy.cacheChangingModulesFor(4, "hours")
}
```

Controlling dependency caching from the command line

Avoiding network access with offline mode

The `--offline` command line switch tells Gradle to always use dependency modules from the cache, regardless if they are due to be checked again. When running with offline, Gradle will never attempt to access the network to perform dependency resolution. If required modules are not present in the dependency cache, build execution will fail.

Refreshing dependencies

You can control the behavior of dependency caching for a distinct build invocation from the command line. Command line options are helpful for making a selective, ad-hoc choice for a single execution of the build.

At times, the Gradle Dependency Cache can become out of sync with the actual state of the configured repositories. Perhaps a repository was initially misconfigured, or perhaps a "non-changing" module was published incorrectly. To refresh all dependencies in the dependency cache, use the `--refresh-dependencies` option on the command line.

The `--refresh-dependencies` option tells Gradle to ignore all cached entries for resolved modules and artifacts. A fresh resolve will be performed against all configured repositories, with dynamic versions recalculated, modules refreshed, and artifacts downloaded. However, where possible Gradle will check if the previously downloaded artifacts are valid before downloading again. This is done by comparing published SHA1 values in the repository with the SHA1 values for existing downloaded artifacts.

- new versions of dynamic dependencies
- new versions of changing modules (modules which use the same version string but can have different contents)

NOTE

Refreshing dependencies will cause Gradle to invalidate its listing caches. However:

- it will perform HTTP HEAD requests on metadata files but *will not re-download them* if they are identical
- it will perform HTTP HEAD requests on artifact files but *will not re-download them* if they are identical

In other words, refreshing dependencies *only* has an impact if you actually use dynamic dependencies *or* that you have changing dependencies that you were not aware of (in which case it is your responsibility to declare them correctly to Gradle as changing dependencies).

It's a common misconception to think that using `--refresh-dependencies` will force download of dependencies. This is **not** the case: Gradle will only perform what is strictly required to refresh the dynamic dependencies. This *may* involve downloading new listing or metadata files, or even artifacts, but if nothing changed, the impact is minimal.

Using component selection rules

Component selection rules may influence which component instance should be selected when multiple versions are available that match a version selector. Rules are applied against every available version and allow the version to be explicitly rejected by rule. This allows Gradle to ignore any component instance that does not satisfy conditions set by the rule. Examples include:

- For a dynamic version like `1.+` certain versions may be explicitly rejected from selection.
- For a static version like `1.4` an instance may be rejected based on extra component metadata such as the Ivy branch attribute, allowing an instance from a subsequent repository to be used.

Rules are configured via the ComponentSelectionRules object. Each rule configured will be called with a ComponentSelection object as an argument which contains information about the candidate version being considered. Calling ComponentSelection.reject(java.lang.String) causes the given candidate version to be explicitly rejected, in which case the candidate will not be considered for the selector.

The following example shows a rule that disallows a particular version of a module but allows the dynamic version to choose the next best candidate.

Example 312. Component selection rule

build.gradle

```
configurations {
    rejectConfig {
        resolutionStrategy {
            componentSelection {
                // Accept the highest version matching the requested version
that isn't '1.5'
                all { ComponentSelection selection ->
                    if (selection.candidate.group == 'org.sample' &&
selection.candidate.module == 'api' && selection.candidate.version == '1.5')
{
                        selection.reject("version 1.5 is broken for
'org.sample:api'")
                    }
                }
            }
        }
    }
}

dependencies {
    rejectConfig "org.sample:api:1.+"
}
```

build.gradle.kts

```
configurations {
    create("rejectConfig") {
        resolutionStrategy {
            componentSelection {
                // Accept the highest version matching the requested version
that isn't '1.5'
                all {
                    if (candidate.group == "org.sample" && candidate.module
== "api" && candidate.version == "1.5") {
                        reject("version 1.5 is broken for 'org.sample:api'")
                    }
                }
            }
        }
    }
}

dependencies {
    "rejectConfig"("org.sample:api:1.+")
}
```

Note that version selection is applied starting with the highest version first. The version selected will be the first version found that all component selection rules accept. A version is considered accepted if no rule explicitly rejects it.

Similarly, rules can be targeted at specific modules. Modules must be specified in the form of `group:module`.

Example 313. Component selection rule with module target

build.gradle

```
configurations {
    targetConfig {
        resolutionStrategy {
            componentSelection {
                withModule("org.sample:api") { ComponentSelection selection
->
                    if (selection.candidate.version == "1.5") {
                        selection.reject("version 1.5 is broken for
'org.sample:api'")
                    }
                }
            }
        }
    }
}
```

build.gradle.kts

```
configurations {
    create("targetConfig") {
        resolutionStrategy {
            componentSelection {
                withModule("org.sample:api") {
                    if (candidate.version == "1.5") {
                        reject("version 1.5 is broken for 'org.sample:api'")
                    }
                }
            }
        }
    }
}
```

Component selection rules can also consider component metadata when selecting a version. Possible additional metadata that can be considered are ComponentMetadata and IvyModuleDescriptor. Note that this extra information may not always be available and thus should be checked for `null` values.

Example 314. Component selection rule with metadata

build.gradle

```
configurations {
    metadataRulesConfig {
        resolutionStrategy {
            componentSelection {
                // Reject any versions with a status of 'experimental'
                all { ComponentSelection selection ->
                    if (selection.candidate.group == 'org.sample' &&
selection.metadata?.status == 'experimental') {
                        selection.reject("don't use experimental candidates
from 'org.sample'")
                    }
                }
                // Accept the highest version with either a "release" branch
or a status of 'milestone'
                withModule('org.sample:api') { ComponentSelection selection
->
                    if (selection.getDescriptor(IvyModuleDescriptor)?.branch
!= "release" && selection.metadata?.status != 'milestone') {
                        selection.reject("'org.sample:api' must be a release
branch or have milestone status")
                    }
                }
            }
        }
    }
}
```

build.gradle.kts

```
configurations {
    create("metadataRulesConfig") {
        resolutionStrategy {
            componentSelection {
                // Reject any versions with a status of 'experimental'
                all {
                    if (candidate.group == "org.sample" && metadata?.status
== "experimental") {
                        reject("don't use experimental candidates from
'org.sample'")
                    }
                }
                // Accept the highest version with either a "release" branch
or a status of 'milestone'
                withModule("org.sample:api") {
                    if (getDescriptor(IvyModuleDescriptor::class)?.branch !=
"release" && metadata?.status != "milestone") {
                        reject("'org.sample:api' must have testing branch or
milestone status")
                    }
                }
            }
        }
    }
}
```

Note that a ComponentSelection argument is *always* required as parameter when declaring a component selection rule.

Locking dependency versions

Use of dynamic dependency versions (e.g. `1.+` or `[1.0,2.0)`) makes builds non-deterministic. This causes builds to break without any obvious change, and worse, can be caused by a transitive dependency that the build author has no control over.

To achieve reproducible builds, it is necessary to *lock* versions of dependencies and transitive dependencies such that a build with the same inputs will always resolve the same module versions. This is called *dependency locking*.

It enables, amongst others, the following scenarios:

- Companies dealing with multi repositories no longer need to rely on `-SNAPSHOT` or changing dependencies, which sometimes result in cascading failures when a dependency introduces a bug or incompatibility. Now dependencies can be declared against major or minor version range, enabling to test with the latest versions on CI while leveraging locking for stable

developer builds.

- Teams that want to always use the latest of their dependencies can use dynamic versions, locking their dependencies only for releases. The release tag will contain the lock states, allowing that build to be fully reproducible when bug fixes need to be developed.

Combined with publishing resolved versions, you can also replace the declared dynamic version part at publication time. Consumers will instead see the versions that your release resolved.

Locking is enabled per dependency configuration. Once enabled, you must create an initial lock state. It will cause Gradle to verify that resolution results do not change, resulting in the same selected dependencies even if newer versions are produced. Modifications to your build that would impact the resolved set of dependencies will cause it to fail. This makes sure that changes, either in published dependencies or build definitions, do not alter resolution without adapting the lock state.

NOTE Dependency locking makes sense only with dynamic versions. It will have no impact on changing versions (like `-SNAPSHOT`) whose coordinates remain the same, though the content may change. Gradle will even emit a warning when persisting lock state and changing dependencies are present in the resolution result.

Enabling locking on configurations

Locking of a configuration happens through the ResolutionStrategy:

Example 315. Locking a specific configuration

build.gradle

```
configurations {
    compileClasspath {
        resolutionStrategy.activateDependencyLocking()
    }
}
```

build.gradle.kts

```
configurations {
    compileClasspath {
        resolutionStrategy.activateDependencyLocking()
    }
}
```

NOTE Only configurations that can be resolved will have lock state attached to them. Applying locking on non resolvable-configurations is simply a no-op.

Or the following, as a way to lock all configurations:

Example 316. Locking all configurations

build.gradle

```
dependencyLocking {
    lockAllConfigurations()
}
```

build.gradle.kts

```
dependencyLocking {
    lockAllConfigurations()
}
```

NOTE The above will lock all *project* configurations, but not the *buildscript* ones.

You can also disable locking on a specific configuration. This can be useful if a plugin configured locking on all configurations but you happen to add one that should not be locked.

Example 317. Unlocking a specific configuration

build.gradle

```
configurations {
    compileClasspath {
        resolutionStrategy.deactivateDependencyLocking()
    }
}
```

build.gradle.kts

```
configurations.compileClasspath {
    resolutionStrategy.deactivateDependencyLocking()
}
```

Locking buildscript classpath configuration

If you apply plugins to your build, you may want to leverage dependency locking there as well. In

order to lock the `classpath` configuration used for script plugins, do the following:

Example 318. Locking buildscript classpath configuration

build.gradle

```
buildscript {
    configurations.classpath {
        resolutionStrategy.activateDependencyLocking()
    }
}
```

build.gradle.kts

```
buildscript {
    configurations.classpath {
        resolutionStrategy.activateDependencyLocking()
    }
}
```

Generating and updating dependency locks

In order to generate or update lock state, you specify the `--write-locks` command line argument in addition to the normal tasks that would trigger configurations to be resolved. This will cause the creation of lock state for each resolved configuration in that build execution. Note that if lock state existed previously, it is overwritten.

NOTE	Gradle will not write lock state to disk if the build fails. This prevents persisting possibly invalid state.

Lock all configurations in one build execution

When locking multiple configurations, you may want to lock them all at once, during a single build execution.

For this, you have two options:

- Run `gradle dependencies --write-locks`. This will effectively lock all resolvable configurations that have locking enabled. Note that in a multi project setup, `dependencies` only is executed on *one* project, the root one in this case.
- Declare a custom task that resolves all configurations. This does not work for Android projects.

Example 319. Resolving all configurations

build.gradle

```
tasks.register('resolveAndLockAll') {
    doFirst {
        assert gradle.startParameter.writeDependencyLocks
    }
    doLast {
        configurations.findAll {
            // Add any custom filtering on the configurations to be resolved
            it.canBeResolved
        }.each { it.resolve() }
    }
}
```

build.gradle.kts

```
tasks.register("resolveAndLockAll") {
    doFirst {
        require(gradle.startParameter.isWriteDependencyLocks)
    }
    doLast {
        configurations.filter {
            // Add any custom filtering on the configurations to be resolved
            it.isCanBeResolved
        }.forEach { it.resolve() }
    }
}
```

That second option, with proper selection of configurations, can be the only option in the native world, where not all configurations can be resolved on a single platform.

Lock state location and format

Lock state will be preserved in a file located at the root of the project or subproject directory. Each file is named `gradle.lockfile`. The one exception to this rule is for the lock file for the buildscript itself. In that case the file will be named `buildscript-gradle.lockfile`.

The lockfile will have the following content:

gradle.lockfile

```
# This is a Gradle generated file for dependency locking.
# Manual edits can break the build and are not advised.
# This file is expected to be part of source control.
org.springframework:spring-beans:5.0.5.RELEASE=compileClasspath, runtimeClasspath
org.springframework:spring-core:5.0.5.RELEASE=compileClasspath, runtimeClasspath
org.springframework:spring-jcl:5.0.5.RELEASE=compileClasspath, runtimeClasspath
empty=annotationProcessor
```

- Each line still represents a single dependency in the `group:artifact:version` notation
- It then lists all configurations that contain the given dependency
- Module and configurations are ordered alphabetically, to ease diffs
- The last line of the file lists all empty configurations, that is configurations known to have no dependencies

which matches the following dependency declaration:

Example 320. Dynamic dependency declaration

build.gradle

```
configurations {
    compileClasspath {
        resolutionStrategy.activateDependencyLocking()
    }
    runtimeClasspath {
        resolutionStrategy.activateDependencyLocking()
    }
    annotationProcessor {
        resolutionStrategy.activateDependencyLocking()
    }
}

dependencies {
    implementation 'org.springframework:spring-beans:[5.0,6.0)'
}
```

build.gradle.kts

```
configurations {
    compileClasspath {
        resolutionStrategy.activateDependencyLocking()
    }
    runtimeClasspath {
        resolutionStrategy.activateDependencyLocking()
    }
    annotationProcessor {
        resolutionStrategy.activateDependencyLocking()
    }
}

dependencies {
    implementation("org.springframework:spring-beans:[5.0,6.0)")
}
```

Migrating from the lockfile per configuration format

If your project uses the legacy lock file format of a file per locked configuration, follow these instructions to migrate to the new format:

- Follow the documentation for writing or updating dependency lock state.
- Upon writing the single lock file per project, Gradle will also delete all lock files per

configuration for which the state was transferred.

NOTE	Migration can be done one configuration at a time. Gradle will keep sourcing the lock state from the per configuration files as long as there is no information for that configuration in the single lock file.

Configuring the per project lock file name and location

When using the single lock file per project, you can configure its name and location. The main reason for providing this is to enable having a file name that is determined by some project properties, effectively allowing a single project to store different lock state for different execution contexts. One trivial example in the JVM ecosystem is the Scala version that is often found in artifact coordinates.

Example 321. Changing the lock file name

build.gradle

```
def scalaVersion = "2.12"
dependencyLocking {
    lockFile = file("$projectDir/locking/gradle-${scalaVersion}.lockfile")
}
```

build.gradle.kts

```
val scalaVersion = "2.12"
dependencyLocking {
    lockFile.set(file("$projectDir/locking/gradle-${scalaVersion}.lockfile"))
}
```

Running a build with lock state present

The moment a build needs to resolve a configuration that has locking enabled and it finds a matching lock state, it will use it to verify that the given configuration still resolves the same versions.

A successful build indicates that the same dependencies are used as stored in the lock state, regardless if new versions matching the dynamic selector have been produced.

The complete validation is as follows:

- Existing entries in the lock state must be matched in the build
  - A version mismatch or missing resolved module causes a build failure
- Resolution result must not contain extra dependencies compared to the lock state

Fine tuning dependency locking behaviour with lock mode

While the default lock mode behaves as described above, two other modes are available:

Strict mode

In this mode, in addition to the validations above, dependency locking will fail if a configuration marked as *locked* does not have lock state associated with it.

Lenient mode

In this mode, dependency locking will still pin dynamic versions but otherwise changes to the dependency resolution are no longer errors.

The lock mode can be controlled from the `dependencyLocking` block as shown below:

Example 322. Setting the lock mode

build.gradle

```
dependencyLocking {
    lockMode = LockMode.STRICT
}
```

build.gradle.kts

```
dependencyLocking {
    lockMode.set(LockMode.STRICT)
}
```

Selectively updating lock state entries

In order to update only specific modules of a configuration, you can use the `--update-locks` command line flag. It takes a comma (`,`) separated list of module notations. In this mode, the existing lock state is still used as input to resolution, filtering out the modules targeted by the update.

```
 gradle classes --update-locks org.apache.commons:commons-lang3,org.slf4j:slf4j-api
```

Wildcards, indicated with `*`, can be used in the group or module name. They can be the only character or appear at the end of the group or module respectively. The following wildcard notation examples are valid:

- `org.apache.commons:*`: will let all modules belonging to group `org.apache.commons` update
- `*:guava`: will let all modules named `guava`, whatever their group, update

- `org.springframework.spring*:spring*`: will let all modules having their group starting with `org.springframework.spring` and name starting with `spring` update

NOTE	The resolution may cause other module versions to update, as dictated by the Gradle resolution rules.

Disabling dependency locking

1. Make sure that the configuration for which you no longer want locking is not configured with locking.
2. Next time you update the save lock state, Gradle will automatically clean up all stale lock state from it.

Gradle needs to resolve a configuration, no longer marked as locked, to detect that associated lock state can be dropped.

Ignoring specific dependencies from the lock state

Dependency locking can be used in cases where reproducibility is not the main goal. As a build author, you may want to have different frequency of dependency version updates, based on their origin for example. In that case, it might be convenient to ignore some dependencies because you always want to use the latest version for those. An example is the internal dependencies in an organization which should always use the latest version as opposed to third party dependencies which have a different upgrade cycle.

WARNING	This feature can break reproducibility and should be used with caution. There are scenarios that are better served with leveraging different lock modes or using different names for lock files.

You can configure ignored dependencies in the `dependencyLocking` project extension:

Example 323. Ignoring dependencies for the lock state

build.gradle

```
dependencyLocking {
    ignoredDependencies.add('com.example:*')
}
```

build.gradle.kts

```
dependencyLocking {
    ignoredDependencies.add("com.example:*")
}
```

The notation is a `<group>:<name>` dependency notation, where `*` can be used as a trailing wildcard. See the description on updating lock files for more details. Note that the value `*:*` is not accepted as it is equivalent to disabling locking.

Ignoring dependencies will have the following effects:

- An ignored dependency applies to all locked configurations. The setting is project scoped.
- Ignoring a dependency does not mean lock state ignores its transitive dependencies.
- There is no validation that an ignored dependency is present in any configuration resolution.
- If the dependency is present in lock state, loading it will filter out the dependency.
- If the dependency is present in the resolution result, it will be ignored when validating that resolution matches the lock state.
- Finally, if the dependency is present in the resolution result and the lock state is persisted, it will be absent from the written lock state.

Locking limitations

- Locking cannot yet be applied to source dependencies.

Nebula locking plugin

This feature is inspired by the Nebula Gradle dependency lock plugin.

Controlling Transitive Dependencies

Upgrading versions of transitive dependencies

Direct dependencies vs dependency constraints

A component may have two different kinds of dependencies:

- direct dependencies are *directly required by the component*. A direct dependency is also referred to as a *first level dependency*. For example, if your project source code requires Guava, Guava should be declared as *direct dependency*.
- transitive dependencies are dependencies that your component needs, but only because another dependency needs them.

It's quite common that issues with dependency management are about *transitive dependencies*. Often developers incorrectly fix transitive dependency issues by adding *direct dependencies*. To avoid this, Gradle provides the concept of *dependency constraints*.

Adding constraints on transitive dependencies

Dependency constraints allow you to define the version or the version range of both dependencies declared in the build script and transitive dependencies. It is the preferred method to express constraints that should be applied to all dependencies of a configuration. When Gradle attempts to resolve a dependency to a module version, all dependency declarations with version, all transitive

dependencies and all dependency constraints for that module are taken into consideration. The highest version that matches all conditions is selected. If no such version is found, Gradle fails with an error showing the conflicting declarations. If this happens you can adjust your dependencies or dependency constraints declarations, or make other adjustments to the transitive dependencies if needed. Similar to dependency declarations, dependency constraint declarations are scoped by configurations and can therefore be selectively defined for parts of a build. If a dependency constraint influenced the resolution result, any type of dependency resolve rules may still be applied afterwards.

Example 324. Define dependency constraints

build.gradle

```
dependencies {
    implementation 'org.apache.httpcomponents:httpclient'
    constraints {
        implementation('org.apache.httpcomponents:httpclient:4.5.3') {
            because 'previous versions have a bug impacting this application'
        }
        implementation('commons-codec:commons-codec:1.11') {
            because 'version 1.9 pulled from httpclient has bugs affecting
this application'
        }
    }
}
```

build.gradle.kts

```
dependencies {
    implementation("org.apache.httpcomponents:httpclient")
    constraints {
        implementation("org.apache.httpcomponents:httpclient:4.5.3") {
            because("previous versions have a bug impacting this
application")
        }
        implementation("commons-codec:commons-codec:1.11") {
            because("version 1.9 pulled from httpclient has bugs affecting
this application")
        }
    }
}
```

In the example, all versions are omitted from the dependency declaration. Instead, the versions are defined in the constraints block. The version definition for `commons-codec:1.11` is only taken into account if `commons-codec` is brought in as transitive dependency, since `commons-codec` is not defined

as dependency in the project. Otherwise, the constraint has no effect. Dependency constraints can also define a rich version constraint and support strict versions to enforce a version even if it contradicts with the version defined by a transitive dependency (e.g. if the version needs to be downgraded).

NOTE	Dependency constraints are only published when using Gradle Module Metadata. This means that currently they are only fully supported if Gradle is used for publishing and consuming (i.e. they are 'lost' when consuming modules with Maven or Ivy).

Dependency constraints themselves can also be added transitively.

Downgrading versions and excluding dependencies

Overriding transitive dependency versions

Gradle resolves any dependency version conflicts by selecting the latest version found in the dependency graph. Some projects might need to divert from the default behavior and enforce an earlier version of a dependency e.g. if the source code of the project depends on an older API of a dependency than some of the external libraries.

WARNING	Forcing a version of a dependency requires a conscious decision. Changing the version of a transitive dependency might lead to runtime errors if external libraries do not properly function without them. Consider upgrading your source code to use a newer version of the library as an alternative approach.

In general, forcing dependencies is done to downgrade a dependency. There might be different use cases for downgrading:

- a bug was discovered in the latest release
- your code depends on a lower version which is not binary compatible
- your code doesn’t depend on the code paths which need a higher version of a dependency

In all situations, this is best expressed saying that your code *strictly depends on* a version of a transitive. Using strict versions, you will effectively depend on the version you declare, even if a transitive dependency says otherwise.

NOTE

Strict dependencies are to some extent similar to Maven's *nearest first* strategy, but there are subtle differences:

- *strict dependencies* don't suffer an ordering problem: they are applied transitively to the subgraph, and it doesn't matter in which order dependencies are declared.
- conflicting strict dependencies will trigger a build failure that you have to resolve
- strict dependencies can be used with rich versions, meaning that it's better to express the requirement in terms of a *strict range* combined with a single *preferred version*.

Let's say a project uses the HttpClient library for performing HTTP calls. HttpClient pulls in Commons Codec as transitive dependency with version 1.10. However, the production source code of the project requires an API from Commons Codec 1.9 which is not available in 1.10 anymore. A dependency version can be enforced by declaring it as strict it in the build script:

Example 325. Setting a strict version

build.gradle

```
dependencies {
    implementation 'org.apache.httpcomponents:httpclient:4.5.4'
    implementation('commons-codec:commons-codec') {
        version {
            strictly '1.9'
        }
    }
}
```

build.gradle.kts

```
dependencies {
    implementation("org.apache.httpcomponents:httpclient:4.5.4")
    implementation("commons-codec:commons-codec") {
        version {
            strictly("1.9")
        }
    }
}
```

Consequences of using strict versions

Using a strict version must be carefully considered, in particular by library authors. As the

producer, a strict version will effectively behave like a *force*: the version declaration takes precedence over whatever is found in the transitive dependency graph. In particular, a *strict version* will override any other *strict version* on the same module found transitively.

However, for consumers, strict versions are still considered globally during graph resolution and *may trigger an error* if the consumer disagrees.

For example, imagine that your project `B` *strictly* depends on `C:1.0`. Now, a consumer, `A`, depends on both `B` and `C:1.1`.

Then this would trigger a resolution error because `A` says it needs `C:1.1` but `B`, *within its subgraph*, strictly needs `1.0`. This means that if you choose a *single version* in a strict constraint, then the version can *no longer be upgraded*, unless the consumer also sets a strict version constraint on the same module.

In the example above, `A` would have to say it *strictly depends on 1.1*.

For this reason, a good practice is that if you use *strict versions*, you should express them in terms of ranges and a preferred version within this range. For example, `B` might say, instead of `strictly 1.0`, that it *strictly depends* on the `[1.0, 2.0[` range, but *prefers* `1.0`. Then if a consumer chooses 1.1 (or any other version in the range), the build will *no longer fail* (constraints are resolved).

Forced dependencies vs strict dependencies

WARNING	Forcing dependencies via ExternalDependency.setForce(boolean) is deprecated and no longer recommended: forced dependencies suffer an ordering issue which can be hard to diagnose and will not work well together with other rich version constraints. You should prefer strict versions instead. If you are authoring and publishing a library, you also need to be aware that `force` is **not** published.

If, for some reason, you can't use strict versions, you can force a dependency doing this:

Example 326. Enforcing a dependency version

build.gradle

```
dependencies {
    implementation 'org.apache.httpcomponents:httpclient:4.5.4'
    implementation('commons-codec:commons-codec:1.9') {
        force = true
    }
}
```

build.gradle.kts

```
dependencies {
    implementation("org.apache.httpcomponents:httpclient:4.5.4")
    implementation("commons-codec:commons-codec:1.9") {
        isForce = true
    }
}
```

If the project requires a specific version of a dependency on a configuration-level then it can be achieved by calling the method ResolutionStrategy.force(java.lang.Object[]).

Example 327. Enforcing a dependency version on the configuration-level

build.gradle

```
configurations {
    compileClasspath {
        resolutionStrategy.force 'commons-codec:commons-codec:1.9'
    }
}

dependencies {
    implementation 'org.apache.httpcomponents:httpclient:4.5.4'
}
```

build.gradle.kts

```
configurations {
    "compileClasspath" {
        resolutionStrategy.force("commons-codec:commons-codec:1.9")
    }
}

dependencies {
    implementation("org.apache.httpcomponents:httpclient:4.5.4")
}
```

Excluding transitive dependencies

While the previous section showed how you can enforce a certain version of a transitive dependency, this section covers *excludes* as a way to remove a transitive dependency completely.

WARNING	Similar as forcing a version of a dependency, excluding a dependency completely requires a conscious decision. Excluding a transitive dependency might lead to runtime errors if external libraries do not properly function without them. If you use excludes, make sure that you do not utilise any code path requiring the excluded dependency by sufficient test coverage.

Transitive dependencies can be excluded on the level of a declared dependency. Exclusions are spelled out as a key/value pair via the attributes `group` and/or `module` as shown in the example below. For more information, refer to ModuleDependency.exclude(java.util.Map).

Example 328. Excluding a transitive dependency for a particular dependency declaration

build.gradle

```
dependencies {
    implementation('commons-beanutils:commons-beanutils:1.9.4') {
        exclude group: 'commons-collections', module: 'commons-collections'
    }
}
```

build.gradle.kts

```
dependencies {
    implementation("commons-beanutils:commons-beanutils:1.9.4") {
        exclude(group = "commons-collections", module = "commons-
collections")
    }
}
```

In this example, we add a dependency to `commons-beanutils` but exclude the transitive dependency `commons-collections`. In our code, shown below, we only use one method from the beanutils library, `PropertyUtils.setSimpleProperty()`. Using this method for existing setters does not require any functionality from `commons-collections` as we verified through test coverage.

Example 329. Using a utility from the beanutils library

src/main/java/Main.java

```
import org.apache.commons.beanutils.PropertyUtils;

public class Main {
    public static void main(String[] args) throws Exception {
        Object person = new Person();
        PropertyUtils.setSimpleProperty(person, "name", "Bart Simpson");
        PropertyUtils.setSimpleProperty(person, "age", 38);
    }
}
```

Effectively, we are expressing that we only use a *subset* of the library, which does not require the `commons-collection` library. This can be seen as implicitly defining a feature variant that has not been explicitly declared by `commons-beanutils` itself. However, the risk of breaking an untested code path increased by doing this.

For example, here we use the `setSimpleProperty()` method to modify properties defined by setters in the `Person` class, which works fine. If we would attempt to set a property not existing on the class, we *should* get an error like `Unknown property on class Person`. However, because the error handling path uses a class from `commons-collections`, the error we now get is `NoClassDefFoundError: org/apache/commons/collections/FastHashMap`. So if our code would be more dynamic, and we would forget to cover the error case sufficiently, consumers of our library might be confronted with unexpected errors.

This is only an example to illustrate potential pitfalls. In practice, larger libraries or frameworks can bring in a huge set of dependencies. If those libraries fail to declare features separately and can only be consumed in a "all or nothing" fashion, excludes can be a valid method to reduce the library to the feature set actually required.

On the upside, Gradle's exclude handling is, in contrast to Maven, taking the whole dependency graph into account. So if there are multiple dependencies on a library, excludes are only exercised if all dependencies agree on them. For example, if we add `opencsv` as another dependency to our project above, which also depends on `commons-beanutils`, `commons-collection` is no longer excluded as `opencsv` itself does **not** exclude it.

Example 330. Excludes only apply if all dependency declarations agree on an exclude

build.gradle

```
    dependencies {
        implementation('commons-beanutils:commons-beanutils:1.9.4') {
            exclude group: 'commons-collections', module: 'commons-
collections'
        }
        implementation 'com.opencsv:opencsv:4.6' // depends on 'commons-
beanutils' without exclude and brings back 'commons-collections'
    }
```

build.gradle.kts

```
dependencies {
    implementation("commons-beanutils:commons-beanutils:1.9.4") {
        exclude(group = "commons-collections", module = "commons-
collections")
    }
    implementation("com.opencsv:opencsv:4.6") // depends on 'commons-
beanutils' without exclude and brings back 'commons-collections'
}
```

If we still want to have `commons-collections` excluded, because our combined usage of `commons-beanutils` and `opencsv` does not need it, we need to exclude it from the transitive dependencies of

`opencsv` as well.

Example 331. Excluding a transitive dependency for multiple dependency declaration

build.gradle

```
    dependencies {
        implementation('commons-beanutils:commons-beanutils:1.9.4') {
            exclude group: 'commons-collections', module: 'commons-
collections'
        }
        implementation('com.opencsv:opencsv:4.6') {
            exclude group: 'commons-collections', module: 'commons-
collections'
        }
    }
```

build.gradle.kts

```
    dependencies {
        implementation("commons-beanutils:commons-beanutils:1.9.4") {
            exclude(group = "commons-collections", module = "commons-
collections")
        }
        implementation("com.opencsv:opencsv:4.6") {
            exclude(group = "commons-collections", module = "commons-
collections")
        }
    }
```

Historically, excludes were also used as a band aid to fix other issues not supported by some dependency management systems. Gradle however, offers a variety of features that might be better suited to solve a certain use case. You may consider to look into the following features:

- Update or downgrade dependency versions: If versions of dependencies clash, it is usually better to adjust the version through a dependency constraint, instead of attempting to exclude the dependency with the undesired version.
- Component Metadata Rules: If a library's metadata is clearly wrong, for example if it includes a compile time dependency which is never needed at compile time, a possible solution is to remove the dependency in a component metadata rule. By this, you tell Gradle that a dependency between two modules is never needed — i.e. the metadata was wrong — and therefore should **never** be considered. If you are developing a library, you have to be aware that this information is not published, and so sometimes an *exclude* can be the better alternative.

- Resolving mutually exclusive dependency conflicts: Another situation that you often see solved by excludes is that two dependencies cannot be used together because they represent two implementations of the same thing (the same capability). Some popular examples are clashing logging API implementations (like `log4j` and `log4j-over-slf4j`) or modules that have different coordinates in different versions (like `com.google.collections` and `guava`). In these cases, if this information is not known to Gradle, it is recommended to add the missing capability information via component metadata rules as described in the declaring component capabilities section. Even if you are developing a library, and your consumers will have to deal with resolving the conflict again, it is often the right solution to leave the decision to the final consumers of libraries. I.e. you as a library author should not have to decide which logging implementation your consumers use in the end.

Sharing dependency versions between projects

Central declaration of dependencies

Using a version catalog

A *version catalog* is a list of dependencies, represented as dependency coordinates, that a user can pick from when declaring dependencies in a build script.

For example, instead of declaring a dependency using a string notation, the dependency coordinates can be picked from a *version catalog*:

Example 332. Using a library declared in a version catalog

build.gradle

```
dependencies {
    implementation(libs.groovy.core)
}
```

build.gradle.kts

```
dependencies {
    implementation(libs.groovy.core)
}
```

In this context, `libs` is a catalog and `groovy` represents a dependency available in this catalog. A version catalog provides a number of advantages over declaring the dependencies directly in build scripts:

- For each catalog, Gradle generates *type-safe accessors* so that you can easily add dependencies with autocompletion in the IDE.

- Each catalog is visible to all projects of a build. It is a central place to declare a version of a dependency and to make sure that a change to that version applies to every subproject.
- Catalogs can declare dependency bundles, which are "groups of dependencies" that are commonly used together.
- Catalogs can separate the group and name of a dependency from its actual version and use version references instead, making it possible to share a version declaration between multiple dependencies.

Adding a dependency using the `libs.someLib` notation works exactly like if you had hardcoded the group, artifact and version directly in the build script.

WARNING	A dependency catalog doesn't enforce the version of a dependency: like a regular dependency notation, it declares the requested version or a rich version. That version is not necessarily the version that is selected during conflict resolution.

Declaring a version catalog

Version catalogs can be declared in the `settings.gradle(.kts)` file. In the example above, in order to make `groovy` available via the `libs` catalog, we need to associate an alias with GAV (group, artifact, version) coordinates:

Example 333. Declaring a version catalog

settings.gradle

```
dependencyResolutionManagement {
    versionCatalogs {
        libs {
            library('groovy-core', 'org.codehaus.groovy:groovy:3.0.5')
            library('groovy-json', 'org.codehaus.groovy:groovy-json:3.0.5')
            library('groovy-nio', 'org.codehaus.groovy:groovy-nio:3.0.5')
            library('commons-lang3', 'org.apache.commons', 'commons-lang3')
.version {
                strictly '[3.8, 4.0['
                prefer '3.9'
            }
        }
    }
}
```

settings.gradle.kts

```
    dependencyResolutionManagement {
        versionCatalogs {
            create("libs") {
                library("groovy-core", "org.codehaus.groovy:groovy:3.0.5")
                library("groovy-json", "org.codehaus.groovy:groovy-
json:3.0.5")
                library("groovy-nio", "org.codehaus.groovy:groovy-nio:3.0.5")
                library("commons-lang3", "org.apache.commons", "commons-
lang3").version {
                    strictly("[3.8, 4.0[")
                    prefer("3.9")
                }
            }
        }
    }
```

Aliases and their mapping to type safe accessors

Aliases must consist of a series of identifiers separated by a dash (`-`, recommended), an underscore (`_`) or a dot (`.`). Identifiers themselves must consist of ascii characters, preferably lowercase, eventually followed by numbers.

For example:

- `guava` is a valid alias

- `groovy-core` is a valid alias
- `commons-lang3` is a valid alias
- `androidx.awesome.lib` is also a valid alias
- but `this.#is.not!`

Then type safe accessors are generated for *each subgroup*. For example, given the following aliases in a version catalog named `libs`:

`guava`, `groovy-core`, `groovy-xml`, `groovy-json`, `androidx.awesome.lib`

We would generate the following type-safe accessors:

- `libs.guava`
- `libs.groovy.core`
- `libs.groovy.xml`
- `libs.groovy.json`
- `libs.androidx.awesome.lib`

Where the `libs` prefix comes from the version catalog name.

In case you want to avoid the generation of a subgroup accessor, we recommend relying on case to differentiate. For example the aliases `groovyCore`, `groovyJson` and `groovyXml` would be mapped to the `libs.groovyCore`, `libs.groovyJson` and `libs.groovyXml` accessors respectively.

When declaring aliases, it's worth noting that any of the `-`, `_` and `.` characters can be used as separators, but the generated catalog will have all normalized to `.`: for example `foo-bar` as an alias is converted to `foo.bar` automatically.

WARNING

Some keywords are reserved, so they cannot be used as an alias. Next words cannot be used as an alias:

- extensions
- class
- convention

Additional to that next words cannot be used as a first subgroup of an alias for dependencies (for bundles, versions and plugins this restriction doesn't apply):

- bundles
- versions
- plugins

So for example for dependencies an alias `versions-dependency` is not valid, but `versionsDependency` or `dependency-versions` are valid.

Dependencies with same version numbers

In the first example in declaring a version catalog, we can see that we declare 3 aliases for various components of the `groovy` library and that all of them share the same version number.

Instead of repeating the same version number, we can declare a version and reference it:

Example 334. Declaring versions separately from libraries

settings.gradle

```
dependencyResolutionManagement {
    versionCatalogs {
        libs {
            version('groovy', '3.0.5')
            version('checkstyle', '8.37')
            library('groovy-core', 'org.codehaus.groovy', 'groovy')
.versionRef('groovy')
            library('groovy-json', 'org.codehaus.groovy', 'groovy-json')
.versionRef('groovy')
            library('groovy-nio', 'org.codehaus.groovy', 'groovy-nio')
.versionRef('groovy')
            library('commons-lang3', 'org.apache.commons', 'commons-lang3')
.version {
                strictly '[3.8, 4.0['
                prefer '3.9'
            }
        }
    }
}
```

settings.gradle.kts

```
    dependencyResolutionManagement {
        versionCatalogs {
            create("libs") {
                version("groovy", "3.0.5")
                version("checkstyle", "8.37")
                library("groovy-core", "org.codehaus.groovy",
"groovy").versionRef("groovy")
                library("groovy-json", "org.codehaus.groovy", "groovy-
json").versionRef("groovy")
                library("groovy-nio", "org.codehaus.groovy", "groovy-
nio").versionRef("groovy")
                library("commons-lang3", "org.apache.commons", "commons-
lang3").version {
                    strictly("[3.8, 4.0[")
                    prefer("3.9")
                }
            }
        }
    }
```

Versions declared separately are *also* available via type-safe accessors, making them usable for more use cases than dependency versions, in particular for tooling:

Example 335. Using a version declared in a version catalog

build.gradle

```
checkstyle {
    // will use the version declared in the catalog
    toolVersion = libs.versions.checkstyle.get()
}
```

build.gradle.kts

```
checkstyle {
    // will use the version declared in the catalog
    toolVersion = libs.versions.checkstyle.get()
}
```

If the alias of a declared version is also a prefix of some more specific alias, as in `libs.versions.zinc` and `libs.versions.zinc.apiinfo`, then the value of the more generic version is available via `asProvider()` on the type-safe accessor:

Example 336. Using a version from a version catalog when there are more specific aliases

build.gradle

```
scala {
    zincVersion = libs.versions.zinc.asProvider().get()
}
```

build.gradle.kts

```
scala {
    zincVersion.set(libs.versions.zinc.asProvider().get())
}
```

Dependencies declared in a catalog are exposed to build scripts via an extension corresponding to their name. In the example above, because the catalog declared in settings is named `libs`, the extension is available via the name `libs` in all build scripts of the current build. Declaring

dependencies using the following notation...

Example 337. Dependency notation correspondance

build.gradle

```
dependencies {
    implementation libs.groovy.core
    implementation libs.groovy.json
    implementation libs.groovy.nio
}
```

build.gradle.kts

```
dependencies {
    implementation(libs.groovy.core)
    implementation(libs.groovy.json)
    implementation(libs.groovy.nio)
}
```

...has **exactly the same** effect as writing:

Example 338. Dependency notation correspondance

build.gradle

```
dependencies {
    implementation 'org.codehaus.groovy:groovy:3.0.5'
    implementation 'org.codehaus.groovy:groovy-json:3.0.5'
    implementation 'org.codehaus.groovy:groovy-nio:3.0.5'
}
```

build.gradle.kts

```
dependencies {
    implementation("org.codehaus.groovy:groovy:3.0.5")
    implementation("org.codehaus.groovy:groovy-json:3.0.5")
    implementation("org.codehaus.groovy:groovy-nio:3.0.5")
}
```

Versions declared in the catalog are rich versions. Please refer to the version catalog builder API for the full version declaration support documentation.

Dependency bundles

Because it's frequent that some dependencies are systematically used together in different projects, a version catalog offers the concept of a "dependency bundle". A bundle is basically an alias for several dependencies. For example, instead of declaring 3 individual dependencies like above, you could write:

Example 339. Using a dependency bundle

build.gradle

```
dependencies {
    implementation libs.bundles.groovy
}
```

build.gradle.kts

```
dependencies {
    implementation(libs.bundles.groovy)
}
```

The bundle named `groovy` needs to be declared in the catalog:

Example 340. Declaring a dependency bundle

settings.gradle

```
dependencyResolutionManagement {
    versionCatalogs {
        libs {
            version('groovy', '3.0.5')
            version('checkstyle', '8.37')
            library('groovy-core', 'org.codehaus.groovy', 'groovy')
.versionRef('groovy')
            library('groovy-json', 'org.codehaus.groovy', 'groovy-json')
.versionRef('groovy')
            library('groovy-nio', 'org.codehaus.groovy', 'groovy-nio')
.versionRef('groovy')
            library('commons-lang3', 'org.apache.commons', 'commons-lang3')
.version {
                strictly '[3.8, 4.0['
                prefer '3.9'
            }
            bundle('groovy', ['groovy-core', 'groovy-json', 'groovy-nio'])
        }
    }
}
```

settings.gradle.kts

```
    dependencyResolutionManagement {
        versionCatalogs {
            create("libs") {
                version("groovy", "3.0.5")
                version("checkstyle", "8.37")
                library("groovy-core", "org.codehaus.groovy",
"groovy").versionRef("groovy")
                library("groovy-json", "org.codehaus.groovy", "groovy-
json").versionRef("groovy")
                library("groovy-nio", "org.codehaus.groovy", "groovy-
nio").versionRef("groovy")
                library("commons-lang3", "org.apache.commons", "commons-
lang3").version {
                    strictly("[3.8, 4.0[")
                    prefer("3.9")
                }
                bundle("groovy", listOf("groovy-core", "groovy-json",
"groovy-nio"))
            }
        }
    }
```

The semantics are again equivalent: adding a single bundle is equivalent to adding all dependencies which are part of the bundle individually.

Plugins

In addition to libraries, version catalog supports declaring plugin versions. While libraries are represented by their group, artifact and version coordinates, Gradle plugins are identified by their id and version only. Therefore, they need to be declared separately:

Example 341. Declaring a plugin version

settings.gradle

```
dependencyResolutionManagement {
    versionCatalogs {
        libs {
            plugin('jmh', 'me.champeau.jmh').version('0.6.5')
        }
    }
}
```

settings.gradle.kts

```
    dependencyResolutionManagement {
        versionCatalogs {
            create("libs") {
                plugin("jmh", "me.champeau.jmh").version("0.6.5")
            }
        }
    }
```

Then the plugin is accessible in the `plugins` block and can be consumed in any project of the build using:

Example 342. Using a plugin declared in a catalog

build.gradle

```
plugins {
    id 'java-library'
    id 'checkstyle'
    // Use the plugin `jmh` as declared in the `libs` version catalog
    alias(libs.plugins.jmh)
}
```

build.gradle.kts

```
plugins {
    `java-library`
    checkstyle
    alias(libs.plugins.jmh)
}
```

Using multiple catalogs

Aside from the conventional `libs` catalog, you can declare any number of catalogs through the `Settings` API. This allows you to separate dependency declarations in multiple sources in a way that makes sense for your projects.

Example 343. Using a custom catalog

settings.gradle

```
dependencyResolutionManagement {
    versionCatalogs {
        testLibs {
            def junit5 = version('junit5', '5.7.1')
            library('junit-api', 'org.junit.jupiter', 'junit-jupiter-api')
.versionRef(junit5)
            library('junit-engine', 'org.junit.jupiter', 'junit-jupiter-
engine').versionRef(junit5)
        }
    }
}
```

settings.gradle.kts

```
dependencyResolutionManagement {
    versionCatalogs {
        create("testLibs") {
            val junit5 = version("junit5", "5.7.1")
            library("junit-api", "org.junit.jupiter", "junit-jupiter-
api").versionRef(junit5)
            library("junit-engine", "org.junit.jupiter", "junit-jupiter-
engine").versionRef(junit5)
        }
    }
}
```

NOTE Each catalog will generate an extension applied to all projects for accessing its content. As such it makes sense to reduce the chance of collisions by picking a name that reduces the potential conflicts. As an example, one option is to pick a name that ends with `Libs`.

The libs.versions.toml file

In addition to the settings API above, Gradle offers a conventional file to declare a catalog. If a `libs.versions.toml` file is found in the `gradle` subdirectory of the root build, then a catalog will be automatically declared with the contents of this file.

WARNING

Declaring a `libs.versions.toml` file doesn't make it the single source of truth for dependencies: it's a conventional location where dependencies can be declared. As soon as you start using catalogs, it's strongly recommended to declare all your dependencies in a catalog and not hardcode group/artifact/version strings in build scripts. Be aware that it may happen that plugins add dependencies, which are dependencies defined outside of this file.

Just like `src/main/java` is a convention to find the Java sources, which doesn't prevent additional source directories to be declared (either in a build script or a plugin), the presence of the `libs.versions.toml` file doesn't prevent the declaration of dependencies elsewhere.

The presence of this file does, however, suggest that most dependencies, if not all, will be declared in this file. Therefore, updating a dependency version, for most users, should only consists of changing a line in this file.

By default, the `libs.versions.toml` file will be an input to the `libs` catalog. It is possible to change the name of the default catalog, for example if you already have an extension with the same name:

Example 344. Changing the default extension name

settings.gradle

```
dependencyResolutionManagement {
    defaultLibrariesExtensionName.set('projectLibs')
}
```

settings.gradle.kts

```
dependencyResolutionManagement {
    defaultLibrariesExtensionName.set("projectLibs")
}
```

The version catalog TOML file format

The TOML file consists of 4 major sections:

- the `[versions]` section is used to declare versions which can be referenced by dependencies
- the `[libraries]` section is used to declare the aliases to coordinates
- the `[bundles]` section is used to declare dependency bundles
- the `[plugins]` section is used to declare plugins

For example:

The libs.versions.toml file

```
[versions]
groovy = "3.0.5"
checkstyle = "8.37"

[libraries]
groovy-core = { module = "org.codehaus.groovy:groovy", version.ref = "groovy" }
groovy-json = { module = "org.codehaus.groovy:groovy-json", version.ref = "groovy" }
groovy-nio = { module = "org.codehaus.groovy:groovy-nio", version.ref = "groovy" }
commons-lang3 = { group = "org.apache.commons", name = "commons-lang3", version = {
strictly = "[3.8, 4.0[", prefer="3.9" } }

[bundles]
groovy = ["groovy-core", "groovy-json", "groovy-nio"]

[plugins]
jmh = { id = "me.champeau.jmh", version = "0.6.5" }
```

Versions can be declared either as a single string, in which case they are interpreted as a *required* version, or as a rich versions:

```
[versions]
my-lib = { strictly = "[1.0, 2.0[", prefer = "1.2" }
```

Supported members of a version declaration are:

- `require`: the required version
- `strictly`: the strict version
- `prefer`: the preferred version
- `reject`: the list of rejected versions
- `rejectAll`: a boolean to reject all versions

Dependency declaration can either be declared as a simple string, in which case they are interpreted as `group:artifact:version` coordinates, or separating the version declaration from the group and name:

NOTE	For aliases, the rules described in the section aliases and their mapping to type safe accessors apply as well.

Different dependency notations

```
[versions]
common = "1.4"

[libraries]
my-lib = "com.mycompany:mylib:1.4"
my-other-lib = { module = "com.mycompany:other", version = "1.4" }
my-other-lib2 = { group = "com.mycompany", name = "alternate", version = "1.4" }
mylib-full-format = { group = "com.mycompany", name = "alternate", version = { require
= "1.4" } }

[plugins]
short-notation = "some.plugin.id:1.4"
long-notation = { id = "some.plugin.id", version = "1.4" }
reference-notation = { id = "some.plugin.id", version.ref = "common" }
```

In case you want to reference a version declared in the `[versions]` section, you should use the `version.ref` property:

```
[versions]
some = "1.4"

[libraries]
my-lib = { group = "com.mycompany", name="mylib", version.ref="some" }
```

NOTE

The TOML file format is very lenient and lets you write "dotted" properties as shortcuts to full object declarations. For example, this:

`a.b.c="d"`

is equivalent to:

`a.b = { c = "d" }`

or

`a = { b = { c = "d" } }`

See the TOML specification for details.

Type unsafe API

Version catalogs can be accessed through a type unsafe API. This API is available in situations where generated accessors are not. It is accessed through the version catalog extension:

build.gradle

```
def versionCatalog = extensions.getByType(VersionCatalogsExtension).named(
"libs")
println "Library aliases: ${versionCatalog.libraryAliases}"
dependencies {
    versionCatalog.findLibrary("groovy-json").ifPresent {
        implementation(it)
    }
}
```

build.gradle.kts

```
val versionCatalog =
extensions.getByType<VersionCatalogsExtension>().named("libs")
println("Library aliases: ${versionCatalog.libraryAliases}")
dependencies {
    versionCatalog.findLibrary("groovy-json").ifPresent {
        implementation(it)
    }
}
```

Check the version catalog API for all supported methods.

Sharing catalogs

Version catalogs are used in a single build (possibly multi-project build) but may also be shared between builds. For example, an organization may want to create a catalog of dependencies that different projects, from different teams, may use.

Importing a catalog from a TOML file

The version catalog builder API supports including a model from an external file. This makes it possible to reuse the catalog of the main build for `buildSrc`, if needed. For example, the `buildSrc/settings.gradle(.kts)` file can include this file using:

Example 345. Sharing the dependency catalog with buildSrc

settings.gradle

```
dependencyResolutionManagement {
    versionCatalogs {
        libs {
            from(files("../gradle/libs.versions.toml"))
        }
    }
}
```

settings.gradle.kts

```
dependencyResolutionManagement {
    versionCatalogs {
        create("libs") {
            from(files("../gradle/libs.versions.toml"))
        }
    }
}
```

WARNING

Only a single file will be accepted when using the VersionCatalogBuilder.from(Object dependencyNotation) method. This means that notations like Project.files(java.lang.Object...) must refer to a single file, otherwise the build will fail.

If a more complicated structure is required (version catalogs imported from multiple files), it's advisable to use a code-based approach, instead of TOML file.

This technique can therefore be used to declare multiple catalogs from different files:

Example 346. Declaring additional catalogs

settings.gradle

```
dependencyResolutionManagement {
    versionCatalogs {
        // declares an additional catalog, named 'testLibs', from the 'test-
libs.versions.toml' file
        testLibs {
            from(files('gradle/test-libs.versions.toml'))
        }
    }
}
```

settings.gradle.kts

```
dependencyResolutionManagement {
    versionCatalogs {
        // declares an additional catalog, named 'testLibs', from the 'test-
libs.versions.toml' file
        create("testLibs") {
            from(files("gradle/test-libs.versions.toml"))
        }
    }
}
```

The version catalog plugin

While importing catalogs from local files is convenient, it doesn't solve the problem of sharing a catalog in an organization or for external consumers. One option to share a catalog is to write a settings plugin, publish it on the Gradle plugin portal or an internal repository, and let the consumers apply the plugin on their settings file.

Alternatively, Gradle offers a *version catalog* plugin, which offers the ability to declare, then publish a catalog.

To do this, you need to apply the `version-catalog` plugin:

Example 347. Applying the version catalog plugin

build.gradle

```
plugins {
    id 'version-catalog'
    id 'maven-publish'
}
```

build.gradle.kts

```
plugins {
    `version-catalog`
    `maven-publish`
}
```

This plugin will then expose the catalog extension that you can use to declare a catalog:

Example 348. Definition of a catalog

build.gradle

```
catalog {
    // declare the aliases, bundles and versions in this block
    versionCatalog {
        library('my-lib', 'com.mycompany:mylib:1.2')
    }
}
```

build.gradle.kts

```
catalog {
    // declare the aliases, bundles and versions in this block
    versionCatalog {
        library("my-lib", "com.mycompany:mylib:1.2")
    }
}
```

Such a catalog can then be published by applying either the `maven-publish` or `ivy-publish` plugin and configuring the publication to use the `versionCatalog` component:

Example 349. Publishing a catalog

build.gradle

```
publishing {
    publications {
        maven(MavenPublication) {
            from components.versionCatalog
        }
    }
}
```

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("maven") {
            from(components["versionCatalog"])
        }
    }
}
```

When publishing such a project, a `libs.versions.toml` file will automatically be generated (and uploaded), which can then be consumed from other Gradle builds.

Importing a published catalog

A catalog produced by the version catalog plugin can be imported via the settings API:

Example 350. Using a published catalog

settings.gradle

```
dependencyResolutionManagement {
    versionCatalogs {
        libs {
            from("com.mycompany:catalog:1.0")
        }
    }
}
```

settings.gradle.kts

```
    dependencyResolutionManagement {
        versionCatalogs {
            create("libs") {
                from("com.mycompany:catalog:1.0")
            }
        }
    }
```

Overwriting catalog versions

In case a catalog declares a version, you can overwrite the version when importing the catalog:

Example 351. Overwriting versions declared in a published catalog

settings.gradle

```
dependencyResolutionManagement {
    versionCatalogs {
        amendedLibs {
            from("com.mycompany:catalog:1.0")
            // overwrite the "groovy" version declared in the imported
catalog
            version("groovy", "3.0.6")
        }
    }
}
```

settings.gradle.kts

```
    dependencyResolutionManagement {
        versionCatalogs {
            create("amendedLibs") {
                from("com.mycompany:catalog:1.0")
                // overwrite the "groovy" version declared in the imported
catalog
                version("groovy", "3.0.6")
            }
        }
    }
```

In the example above, any dependency which was using the `groovy` version as reference will be automatically updated to use `3.0.6`.

NOTE Again, overwriting a version doesn't mean that the actual *resolved* dependency version will be the same: this only changes what is *imported*, that is to say what is used when declaring a dependency. The actual version will be subject to traditional conflict resolution, if any.

Using a platform to control transitive versions

A platform is a special software component which can be used to control transitive dependency versions. In most cases it's exclusively composed of dependency constraints which will either *suggest* dependency versions or *enforce* some versions. As such, this is a perfect tool whenever you need to *share dependency versions between projects*. In this case, a project will typically be organized this way:

- a `platform` project which defines constraints for the various dependencies found in the different

sub-projects

- a number of sub-projects which *depend on* the platform and declare dependencies *without version*

In the Java ecosystem, Gradle provides a plugin for this purpose.

It's also common to find platforms published as Maven BOMs which Gradle supports natively.

A dependency on a platform is created using the `platform` keyword:

Example 352. Getting versions declared in a platform

build.gradle

```
dependencies {
    // get recommended versions from the platform project
    api platform(project(':platform'))
    // no version required
    api 'commons-httpclient:commons-httpclient'
}
```

build.gradle.kts

```
dependencies {
    // get recommended versions from the platform project
    api(platform(project(":platform")))
    // no version required
    api("commons-httpclient:commons-httpclient")
}
```

NOTE

This `platform` notation is a short-hand notation which actually performs several operations under the hood:

- it sets the org.gradle.category attribute to `platform`, which means that Gradle will select the *platform* component of the dependency.
- it sets the endorseStrictVersions behavior by default, meaning that if the platform declares strict dependencies, they will be enforced.

This means that by default, a dependency to a platform triggers the inheritance of all strict versions defined in that platform, which can be useful for platform authors to make sure that all consumers respect their decisions in terms of versions of dependencies. This can be turned off by explicitly calling the `doNotEndorseStrictVersions` method.

Importing Maven BOMs

Gradle provides support for importing bill of materials (BOM) files, which are effectively `.pom` files that use `<dependencyManagement>` to control the dependency versions of direct and transitive dependencies. The BOM support in Gradle works similar to using `<scope>import</scope>` when depending on a BOM in Maven. In Gradle however, it is done via a regular dependency declaration on the BOM:

Example 353. Depending on a BOM to import its dependency constraints

build.gradle

```
dependencies {
    // import a BOM
    implementation platform('org.springframework.boot:spring-boot-
dependencies:1.5.8.RELEASE')

    // define dependencies without versions
    implementation 'com.google.code.gson:gson'
    implementation 'dom4j:dom4j'
}
```

build.gradle.kts

```
dependencies {
    // import a BOM
    implementation(platform("org.springframework.boot:spring-boot-
dependencies:1.5.8.RELEASE"))

    // define dependencies without versions
    implementation("com.google.code.gson:gson")
    implementation("dom4j:dom4j")
}
```

In the example, the versions of `gson` and `dom4j` are provided by the Spring Boot BOM. This way, if you are developing for a platform like Spring Boot, you do not have to declare any versions yourself but can rely on the versions the platform provides.

Gradle treats all entries in the `<dependencyManagement>` block of a BOM similar to Gradle's dependency constraints. This means that any version defined in the `<dependencyManagement>` block can impact the dependency resolution result. In order to qualify as a BOM, a `.pom` file needs to have `<packaging>pom</packaging>` set.

However often BOMs are not only providing versions as recommendations, but also a way to override any other version found in the graph. You can enable this behavior by using the

`enforcedPlatform` keyword, instead of `platform`, when importing the BOM:

Example 354. Importing a BOM, making sure the versions it defines override any other version found

build.gradle

```
dependencies {
    // import a BOM. The versions used in this file will override any other
version found in the graph
    implementation enforcedPlatform('org.springframework.boot:spring-boot-
dependencies:1.5.8.RELEASE')

    // define dependencies without versions
    implementation 'com.google.code.gson:gson'
    implementation 'dom4j:dom4j'

    // this version will be overridden by the one found in the BOM
    implementation 'org.codehaus.groovy:groovy:1.8.6'
}
```

build.gradle.kts

```
dependencies {
    // import a BOM. The versions used in this file will override any other
version found in the graph
    implementation(enforcedPlatform("org.springframework.boot:spring-boot-
dependencies:1.5.8.RELEASE"))

    // define dependencies without versions
    implementation("com.google.code.gson:gson")
    implementation("dom4j:dom4j")

    // this version will be overridden by the one found in the BOM
    implementation("org.codehaus.groovy:groovy:1.8.6")
}
```

WARNING

Using `enforcedPlatform` needs to be considered with care if your software component can be consumed by others. This declaration is effectively transitive and so will apply to the dependency graph of your consumers. Unfortunately they will have to use `exclude` if they happen to disagree with one of the forced versions.

Instead, if your reusable software component has a strong opinion on some third party dependency versions, consider using a rich version declaration with a `strictly`.

Should I use a platform or a catalog?

Because platforms and catalogs both talk about dependency versions and can both be used to share dependency versions in a project, there might be a confusion regarding what to use and if one is preferable to the other.

In short, you should:

- use catalogs to only define dependencies and their versions for projects and to generate type-safe accessors
- use platform to apply versions to dependency graph and to affect dependency resolution

A catalog helps with centralizing the dependency versions and is only, as it name implies, a catalog of dependencies you can pick from. We recommend using it to declare the coordinates of your dependencies, in all cases. It will be used by Gradle to generate type-safe accessors, present short-hand notations for external dependencies and it allows sharing those coordinates between different projects easily. Using a catalog will not have any kind of consequence on downstream consumers: it's transparent to them.

A platform is a more heavyweight construct: it's a component of a dependency graph, like any other library. If you depend on a platform, that platform is itself a component in the graph. It means, in particular, that:

- Constraints defined in a platform can influence *transitive* dependencies, not only the direct dependencies of your project.
- A platform is versioned, and a transitive dependency in the graph can depend on a different version of the platform, causing various dependency upgrades.
- A platform can tie components together, and in particular can be used as a construct for aligning versions.
- A dependency on a platform is "inherited" by the consumers of your dependency: it means that a dependency on a platform can influence what versions of libraries would be used by your consumers even if you don't directly, or transitively, depend on components the platform references.

In summary, using a catalog is always a good engineering practice as it centralizes common definitions, allows sharing of dependency versions or plugin versions, but it is an "implementation detail" of the build: it will not be visible to consumers and unused elements of a catalog are just ignored.

A platform is meant to influence the dependency resolution graph, for example by adding constraints on transitive dependencies: it's a solution for structuring a dependency graph and influencing the resolution result.

In practice, your project can both use a catalog *and* declare a platform which itself uses the catalog:

Example 355. Using a catalog within a platform definition

build.gradle

```
plugins {
    id 'java-platform'
}

dependencies {
    constraints {
        api(libs.mylib)
    }
}
```

build.gradle.kts

```
plugins {
    `java-platform`
}

dependencies {
    constraints {
        api(libs.mylib)
    }
}
```

Aligning dependency versions

Dependency version alignment allows different modules belonging to the same logical group (a *platform*) to have identical versions in a dependency graph.

Handling inconsistent module versions

Gradle supports aligning versions of modules which belong to the same "platform". It is often preferable, for example, that the API and implementation modules of a component are using the same version. However, because of the game of transitive dependency resolution, it is possible that different modules belonging to the same platform end up using different versions. For example, your project may depend on the `jackson-databind` and `vert.x` libraries, as illustrated below:

Example 356. Declaring dependencies

build.gradle

```
dependencies {
    // a dependency on Jackson Databind
    implementation 'com.fasterxml.jackson.core:jackson-databind:2.8.9'

    // and a dependency on vert.x
    implementation 'io.vertx:vertx-core:3.5.3'
}
```

build.gradle.kts

```
dependencies {
    // a dependency on Jackson Databind
    implementation("com.fasterxml.jackson.core:jackson-databind:2.8.9")

    // and a dependency on vert.x
    implementation("io.vertx:vertx-core:3.5.3")
}
```

Because `vert.x` depends on `jackson-core`, we would actually resolve the following dependency versions:

- `jackson-core` version `2.9.5` (brought by `vertx-core`)
- `jackson-databind` version `2.9.5` (by conflict resolution)
- `jackson-annotation` version `2.9.0` (dependency of `jackson-databind:2.9.5`)

It's easy to end up with a set of versions which do not work well together. To fix this, Gradle supports dependency version alignment, which is supported by the concept of platforms. A platform represents a set of modules which "work well together". Either because they are actually published as a whole (when one of the members of the platform is published, all other modules are also published with the same version), or because someone tested the modules and indicates that they work well together (typically, the Spring Platform).

Aligning versions natively with Gradle

Gradle natively supports alignment of modules produced by Gradle. This is a direct consequence of the transitivity of dependency constraints. So if you have a multi-project build, and you wish that consumers get the same version of all your modules, Gradle provides a simple way to do this using the Java Platform Plugin.

For example, if you have a project that consists of 3 modules:

- `lib`
- `utils`
- `core`, depending on `lib` and `utils`

And a consumer that declares the following dependencies:

- `core` version 1.0
- `lib` version 1.1

Then by default resolution would select `core:1.0` and `lib:1.1`, because `lib` has no dependency on `core`. We can fix this by adding a new module in our project, a *platform*, that will add constraints on all the modules of your project:

Example 357. The platform module

build.gradle

```
plugins {
    id 'java-platform'
}

dependencies {
    // The platform declares constraints on all components that
    // require alignment
    constraints {
        api(project(":core"))
        api(project(":lib"))
        api(project(":utils"))
    }
}
```

build.gradle.kts

```
plugins {
    `java-platform`
}

dependencies {
    // The platform declares constraints on all components that
    // require alignment
    constraints {
        api(project(":core"))
        api(project(":lib"))
        api(project(":utils"))
    }
}
```

Once this is done, we need to make sure that all modules now *depend on the platform*, like this:

Example 358. Declaring a dependency on the platform

build.gradle

```
dependencies {
    // Each project has a dependency on the platform
    api(platform(project(":platform")))

    // And any additional dependency required
    implementation(project(":lib"))
    implementation(project(":utils"))
}
```

build.gradle.kts

```
dependencies {
    // Each project has a dependency on the platform
    api(platform(project(":platform")))

    // And any additional dependency required
    implementation(project(":lib"))
    implementation(project(":utils"))
}
```

It is important that the platform contains a constraint on all the components, but also that each component has a dependency on the platform. By doing this, whenever Gradle will add a dependency to a module of the platform on the graph, it will *also* include constraints on the other modules of the platform. This means that if we see another module belonging to the same platform, we will automatically upgrade to the same version.

In our example, it means that we first see `core:1.0`, which brings a platform `1.0` with constraints on `lib:1.0` and `lib:1.0`. Then we add `lib:1.1` which has a dependency on `platform:1.1`. By conflict resolution, we select the `1.1` platform, which has a constraint on `core:1.1`. Then we conflict resolve between `core:1.0` and `core:1.1`, which means that `core` and `lib` are now aligned properly.

NOTE	This behavior is enforced for published components only if you use Gradle Module Metadata.

Aligning versions of modules not published with Gradle

Whenever the publisher doesn't use Gradle, like in our Jackson example, we can explain to Gradle that all Jackson modules "belong to" the same platform and benefit from the same behavior as with native alignment. There are two options to express that a set of modules belong to a platform:

1. A platform is **published** as a BOM and can be used: For example,

`com.fasterxml.jackson:jackson-bom` can be used as platform. The information missing to Gradle in that case is that the platform should be added to the dependencies if one of its members is used.

2. No existing platform can be used. Instead, a **virtual platform** should be created by Gradle: In this case, Gradle builds up the platform itself based on all the members that are used.

To provide the missing information to Gradle, you can define component metadata rules as explained in the following.

Align versions of modules using a published BOM

Example 359. A dependency version alignment rule

build.gradle

```
abstract class JacksonBomAlignmentRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext ctx) {
        ctx.details.with {
            if (id.group.startsWith("com.fasterxml.jackson")) {
                // declare that Jackson modules belong to the platform
defined by the Jackson BOM
                belongsTo("com.fasterxml.jackson:jackson-bom:${id.version}",
false)
            }
        }
    }
}
```

build.gradle.kts

```
abstract class JacksonBomAlignmentRule: ComponentMetadataRule {
    override fun execute(ctx: ComponentMetadataContext) {
        ctx.details.run {
            if (id.group.startsWith("com.fasterxml.jackson")) {
                // declare that Jackson modules belong to the platform
defined by the Jackson BOM
                belongsTo("com.fasterxml.jackson:jackson-bom:${id.version}",
false)
            }
        }
    }
}
```

By using the `belongsTo` with `false` (**not** virtual), we declare that all modules belong to the same *published platform*. In this case, the platform is `com.fasterxml.jackson:jackson-bom` and Gradle will

look for it, as for any other module, in the declared repositories.

Example 360. Making use of a dependency version alignment rule

build.gradle

```
dependencies {
    components.all(JacksonBomAlignmentRule)
}
```

build.gradle.kts

```
dependencies {
    components.all<JacksonBomAlignmentRule>()
}
```

Using the rule, the versions in the example above align to whatever the selected version of `com.fasterxml.jackson:jackson-bom` defines. In this case, `com.fasterxml.jackson:jackson-bom:2.9.5` will be selected as `2.9.5` is the highest version of a module selected. In that BOM, the following versions are defined and will be used: `jackson-core:2.9.5`, `jackson-databind:2.9.5` and `jackson-annotation:2.9.0`. The lower versions of `jackson-annotation` here might be the desired result as it is what the BOM recommends.

NOTE This behavior is working reliable since Gradle 6.1. Effectively, it is similar to a component metadata rule that adds a platform dependency to all members of the platform using `withDependencies`.

Align versions of modules without a published platform

Example 361. A dependency version alignment rule

build.gradle

```
abstract class JacksonAlignmentRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext ctx) {
        ctx.details.with {
            if (id.group.startsWith("com.fasterxml.jackson")) {
                // declare that Jackson modules all belong to the Jackson
virtual platform
                belongsTo("com.fasterxml.jackson:jackson-virtual-platform:
${id.version}")
            }
        }
    }
}
```

build.gradle.kts

```
abstract class JacksonAlignmentRule: ComponentMetadataRule {
    override fun execute(ctx: ComponentMetadataContext) {
        ctx.details.run {
            if (id.group.startsWith("com.fasterxml.jackson")) {
                // declare that Jackson modules all belong to the Jackson
virtual platform
                belongsTo("com.fasterxml.jackson:jackson-virtual-
platform:${id.version}")
            }
        }
    }
}
```

By using the `belongsTo` keyword without further parameter (platform **is** virtual), we declare that all modules belong to the same *virtual platform*, which is treated specially by the engine. A virtual platform will not be retrieved from a repository. The identifier, in this case `com.fasterxml.jackson:jackson-virtual-platform`, is something you as the build author define yourself. The "content" of the platform is then created by Gradle on the fly by collecting all `belongsTo` statements pointing at the same virtual platform.

Example 362. Making use of a dependency version alignment rule

build.gradle

```
dependencies {
    components.all(JacksonAlignmentRule)
}
```

build.gradle.kts

```
dependencies {
    components.all<JacksonAlignmentRule>()
}
```

Using the rule, all versions in the example above would align to `2.9.5`. In this case, also `jackson-annotation:2.9.5` will be taken, as that is how we defined our local virtual platform.

For both published and virtual platforms, Gradle lets you override the version choice of the platform itself by specifying an *enforced* dependency on the platform:

Example 363. Forceful platform downgrade

build.gradle

```
dependencies {
    // Forcefully downgrade the virtual Jackson platform to 2.8.9
    implementation enforcedPlatform('com.fasterxml.jackson:jackson-virtual-platform:2.8.9')
}
```

build.gradle.kts

```
dependencies {
    // Forcefully downgrade the virtual Jackson platform to 2.8.9
    implementation(enforcedPlatform("com.fasterxml.jackson:jackson-virtual-platform:2.8.9"))
}
```

Handling mutually exclusive dependencies

Introduction to component capabilities

Often a dependency graph would accidentally contain multiple implementations of the same API. This is particularly common with logging frameworks, where multiple bindings are available, and that one library chooses a binding when another transitive dependency chooses another. Because those implementations live at different GAV coordinates, the build tool has usually no way to find out that there's a conflict between those libraries. To solve this, Gradle provides the concept of *capability*.

It's illegal to find two components providing the same *capability* in a single dependency graph. Intuitively, it means that if Gradle finds two components that provide the same thing on classpath, it's going to fail with an error indicating what modules are in conflict. In our example, it means that different bindings of a logging framework provide the same capability.

Capability coordinates

A *capability* is defined by a `(group, module, version)` triplet. Each component defines an implicit capability corresponding to its GAV coordinates (group, artifact, version). For example, the `org.apache.commons:commons-lang3:3.8` module has an implicit capability with group `org.apache.commons`, name `commons-lang3` and version `3.8`. It is important to realize that capabilities are *versioned*.

Declaring component capabilities

By default, Gradle will fail if two components in the dependency graph provide the same capability. Because most modules are currently published without Gradle Module Metadata, capabilities are not always automatically discovered by Gradle. It is however interesting to use *rules* to declare component capabilities in order to discover conflicts as soon as possible, during the build instead of runtime.

A typical example is whenever a component is relocated at different coordinates in a new release. For example, the ASM library lived at `asm:asm` coordinates until version `3.3.1`, then changed to `org.ow2.asm:asm` since `4.0`. It is illegal to have both ASM <= 3.3.1 and 4.0+ on the classpath, because they provide the same feature, it's just that the component has been relocated. Because each component has an implicit capability corresponding to its GAV coordinates, we can "fix" this by having a rule that will declare that the `asm:asm` module provides the `org.ow2.asm:asm` capability:

Example 364. Conflict resolution by capability

build.gradle

```
@CompileStatic
class AsmCapability implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        context.details.with {
            if (id.group == "asm" && id.name == "asm") {
                allVariants {
                    it.withCapabilities {
                        // Declare that ASM provides the org.ow2.asm:asm
capability, but with an older version
                        it.addCapability("org.ow2.asm", "asm", id.version)
                    }
                }
            }
        }
    }
}
```

build.gradle.kts

```
class AsmCapability : ComponentMetadataRule {
    override
    fun execute(context: ComponentMetadataContext) = context.details.run {
        if (id.group == "asm" && id.name == "asm") {
            allVariants {
                withCapabilities {
                    // Declare that ASM provides the org.ow2.asm:asm
capability, but with an older version
                    addCapability("org.ow2.asm", "asm", id.version)
                }
            }
        }
    }
}
```

Now the build is going to *fail* whenever the two components are found in the same dependency graph.

NOTE	At this stage, Gradle will *only* make more builds fail. It will **not** automatically fix the problem for you, but it helps you realize that you have a problem. It is recommended to write such rules in *plugins* which are then applied to your builds. Then, users *have to* express their preferences, if possible, or fix the problem of having incompatible things on the classpath, as explained in the following section.

Selecting between candidates

At some point, a dependency graph is going to include either *incompatible modules*, or modules which are *mutually exclusive*. For example, you may have different logger implementations and you need to choose one binding. Capabilities help *realizing* that you have a conflict, but Gradle also provides tools to express how to solve the conflicts.

Selecting between different capability candidates

In the relocation example above, Gradle was able to tell you that you have two versions of the same API on classpath: an "old" module and a "relocated" one. Now we can solve the conflict by automatically choosing the component which has the highest capability version:

Example 365. Conflict resolution by capability versioning

build.gradle

```
configurations.all {
    resolutionStrategy.capabilitiesResolution.withCapability('
org.ow2.asm:asm') {
        selectHighestVersion()
    }
}
```

build.gradle.kts

```
configurations.all {

resolutionStrategy.capabilitiesResolution.withCapability("org.ow2.asm:asm") {
        selectHighestVersion()
    }
}
```

However, fixing by choosing the highest capability version conflict resolution is not always suitable. For a logging framework, for example, it doesn't matter what version of the logging frameworks we use, we should always select Slf4j.

In this case, we can fix it by explicitly selecting slf4j as the winner:

Example 366. Substitute log4j with slf4j

build.gradle

```
    configurations.all {
        resolutionStrategy.capabilitiesResolution.withCapability("
log4j:log4j") {
            def toBeSelected = candidates.find { it.id instanceof
ModuleComponentIdentifier && it.id.module == 'log4j-over-slf4j' }
            if (toBeSelected != null) {
                select(toBeSelected)
            }
            because 'use slf4j in place of log4j'
        }
    }
```

build.gradle.kts

```
    configurations.all {

resolutionStrategy.capabilitiesResolution.withCapability("log4j:log4j") {
            val toBeSelected = candidates.firstOrNull { it.id.let { id -> id
is ModuleComponentIdentifier && id.module == "log4j-over-slf4j" } }
            if (toBeSelected != null) {
                select(toBeSelected)
            }
            because("use slf4j in place of log4j")
        }
    }
```

Note that this approach works also well if you have multiple *Slf4j bindings* on the classpath: bindings are basically different logger implementations and you need only one. However, the selected implementation may depend on the configuration being resolved. For example, for tests, `slf4j-simple` may be enough but for production, `slf4-over-log4j` may be better.

NOTE

Resolution can only be made in favor of a module *found* in the graph.

The `select` method only accepts a module found in the *current* candidates. If the module you want to select is not part of the conflict, you can abstain from performing a selection, effectively not resolving *this* conflict. It might be that another conflict exists in the graph for the same capability and will have the module you want to select.

If no resolution is given for all conflicts on a given capability, the build will fail given the module chosen for resolution was not part of the graph at all.

In addition `select(null)` will result in an error and so should be avoided.

For more information, check out the the capabilities resolution API.

Fixing metadata with component metadata rules

Each module that is pulled from a repository has metadata associated with it, such as its group, name, version as well as the different variants it provides with their artifacts and dependencies. Sometimes, this metadata is incomplete or incorrect. To manipulate such incomplete metadata from within the build script, Gradle offers an API to write *component metadata rules*. These rules take effect after a module's metadata has been downloaded, but before it is used in dependency resolution.

Basics of writing a component metadata rule

Component metadata rules are applied in the components (ComponentMetadataHandler) section of the dependencies block (DependencyHandler) of a build script or in the settings script. The rules can be defined in two different ways:

1. As an action directly when they are applied in the *components* section
2. As an isolated class implementing the ComponentMetadataRule interface

While defining rules inline as action can be convenient for experimentation, it is generally recommended to define rules as separate classes. Rules that are written as isolated classes can be annotated with `@CacheableRule` to cache the results of their application such that they do not need to be re-executed each time dependencies are resolved.

Example 367. Example of a configurable component metadata rule

build.gradle

```
@CacheableRule
abstract class TargetJvmVersionRule implements ComponentMetadataRule {
    final Integer jvmVersion
    @Inject TargetJvmVersionRule(Integer jvmVersion) {
        this.jvmVersion = jvmVersion
    }

    @Inject abstract ObjectFactory getObjects()

    void execute(ComponentMetadataContext context) {
        context.details.withVariant("compile") {
            attributes {
                attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
jvmVersion)
                attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage, Usage
.JAVA_API))
            }
        }
    }
}
dependencies {
    components {
        withModule("commons-io:commons-io", TargetJvmVersionRule) {
            params(7)
        }
        withModule("commons-collections:commons-collections",
TargetJvmVersionRule) {
            params(8)
        }
    }
    implementation("commons-io:commons-io:2.6")
    implementation("commons-collections:commons-collections:3.2.2")
}
```

build.gradle.kts

```
@CacheableRule
abstract class TargetJvmVersionRule @Inject constructor(val jvmVersion: Int)
: ComponentMetadataRule {
    @get:Inject abstract val objects: ObjectFactory

    override fun execute(context: ComponentMetadataContext) {
        context.details.withVariant("compile") {
            attributes {
                attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
jvmVersion)
                attribute(Usage.USAGE_ATTRIBUTE,
objects.named(Usage.JAVA_API))
            }
        }
    }
}
dependencies {
    components {
        withModule<TargetJvmVersionRule>("commons-io:commons-io") {
            params(7)
        }
        withModule<TargetJvmVersionRule>("commons-collections:commons-
collections") {
            params(8)
        }
    }
    implementation("commons-io:commons-io:2.6")
    implementation("commons-collections:commons-collections:3.2.2")
}
```

As can be seen in the examples above, component metadata rules are defined by implementing ComponentMetadataRule which has a single `execute` method receiving an instance of ComponentMetadataContext as parameter. In this example, the rule is also further configured through an ActionConfiguration. This is supported by having a constructor in your implementation of `ComponentMetadataRule` accepting the parameters that were configured and the services that need injecting.

Gradle enforces isolation of instances of `ComponentMetadataRule`. This means that all parameters must be `Serializable` or known Gradle types that can be isolated.

In addition, Gradle services can be injected into your `ComponentMetadataRule`. Because of this, the moment you have a constructor, it must be annotated with `@javax.inject.Inject`. A commonly required service is ObjectFactory to create instances of strongly typed value objects like a value for setting an Attribute. A service which is helpful for advanced usage of component metadata rules with custom metadata is the RepositoryResourceAccessor.

A component metadata rule can be applied to all modules — `all(rule)` — or to a selected module — `withModule(groupAndName, rule)`. Usually, a rule is specifically written to enrich metadata of one specific module and hence the `withModule` API should be preferred.

Declaring rules in a central place

NOTE	Declaring component metadata rules in settings is an incubating feature

Instead of declaring rules for each subproject individually, it is possible to declare rules in the `settings.gradle(.kts)` file for the whole build. Rules declared in settings are the *conventional* rules applied to each project: if the project doesn't declare any rules, the rules from the settings script will be used.

Example 368. Declaring a rule in settings

settings.gradle

```
dependencyResolutionManagement {
    components {
        withModule("com.google.guava:guava", GuavaRule)
    }
}
```

settings.gradle.kts

```
dependencyResolutionManagement {
    components {
        withModule<GuavaRule>("com.google.guava:guava")
    }
}
```

By default, rules declared in a project will **override** whatever is declared in settings. It is possible to change this default, for example to always prefer the settings rules:

Example 369. Preferring rules declared in settings

settings.gradle

```
dependencyResolutionManagement {
    rulesMode.set(RulesMode.PREFER_SETTINGS)
}
```

settings.gradle.kts

```
dependencyResolutionManagement {
    rulesMode.set(RulesMode.PREFER_SETTINGS)
}
```

If this method is called and that a project or plugin declares rules, a warning will be issued. You can make this a failure instead by using this alternative:

Example 370. Enforcing rules declared in settings

settings.gradle

```
dependencyResolutionManagement {
    rulesMode.set(RulesMode.FAIL_ON_PROJECT_RULES)
}
```

settings.gradle.kts

```
dependencyResolutionManagement {
    rulesMode.set(RulesMode.FAIL_ON_PROJECT_RULES)
}
```

The default behavior is equivalent to calling this method:

Example 371. Preferring rules declared in projects

settings.gradle

```
dependencyResolutionManagement {
    rulesMode.set(RulesMode.PREFER_PROJECT)
}
```

settings.gradle.kts

```
dependencyResolutionManagement {
    rulesMode.set(RulesMode.PREFER_PROJECT)
}
```

Which parts of metadata can be modified?

The component metadata rules API is oriented at the features supported by Gradle Module Metadata and the *dependencies* API in build scripts. The main difference between writing rules and defining dependencies and artifacts in the build script is that component metadata rules, following the structure of Gradle Module Metadata, operate on variants directly. On the contrary, in build scripts you often influence the shape of multiple variants at once (e.g. an *api* dependency is added to the *api* and *runtime* variant of a Java library, the artifact produced by the *jar* task is also added to these two variants).

Variants can be addressed for modification through the following methods:

- `allVariants`: modify all variants of a component
- `withVariant(name)`: modify a single variant identified by its name
- `addVariant(name)` or `addVariant(name, base)`: add a new variant to the component either *from scratch* or by *copying* the details of an existing variant (base)

The following details of each variant can be adjusted:

- The attributes that identify the variant — `attributes {}` block
- The capabilities the variant provides — `withCapabilities { }` block
- The dependencies of the variant, including rich versions — `withDependencies {}` block
- The dependency constraints of the variant, including rich versions — `withDependencyConstraints {}` block
- The location of the published files that make up the actual content of the variant — `withFiles { }` block

There are also a few properties of the whole component that can be changed:

- The *component level attributes*, currently the only meaningful attribute there is `org.gradle.status`
- The *status scheme* to influence interpretation of the `org.gradle.status` attribute during version selection
- The *belongsTo* property for version alignment through virtual platforms

Depending on the format of the metadata of a module, it is mapped differently to the variant-centric representation of the metadata:

- If the module has Gradle Module Metadata, the data structure the rule operates on is very similar to what you find in the module's `.module` file.
- If the module was published only with `.pom` metadata, a number of fixed variants is derived as explained in the mapping of POM files to variants section.
- If the module was published only with an `ivy.xml` file, the *Ivy configurations* defined in the file can be accessed instead of variants. Their dependencies, dependency constraints and files can be modified. Additionally, the `addVariant(name, baseVariantOrConfiguration) { }` API can be used to derive variants from *Ivy configurations* if desired (for example, *compile* and *runtime* variants for the Java library plugin can be defined with this).

When to use Component Metadata Rules?

In general, if you consider using component metadata rules to adjust the metadata of a certain module, you should check first if that module was published with Gradle Module Metadata (`.module` file) or traditional metadata only (`.pom` or `ivy.xml`).

If a module was published with Gradle Module Metadata, the metadata is likely complete although there can still be cases where something is just plainly wrong. For these modules you should only use component metadata rules if you have clearly identified a problem with the metadata itself. If you have an issue with the dependency resolution result, you should first check if you can solve the issue by declaring dependency constraints with rich versions. In particular, if you are developing a library that you publish, you should remember that dependency constraints, in contrast to component metadata rules, are published as part of the metadata of your own library. So with dependency constraints, you automatically share the solution of dependency resolution issues with your consumers, while component metadata rules are only applied to your own build.

If a module was published with traditional metadata (`.pom` or `ivy.xml` only, no `.module` file) it is more likely that the metadata is incomplete as features such as variants or dependency constraints are not supported in these formats. Still, conceptually such modules can contain different variants or might have dependency constraints they just omitted (or wrongly defined as dependencies). In the next sections, we explore a number existing oss modules with such incomplete metadata and the rules for adding the missing metadata information.

As a rule of thumb, you should contemplate if the rule you are writing also works out of context of your build. That is, does the rule still produce a correct and useful result if applied in any other build that uses the module(s) it affects?

Fixing wrong dependency details

Let's consider as an example the publication of the Jaxen XPath Engine on Maven central. The pom of version 1.1.3 declares a number of dependencies in the compile scope which are not actually needed for compilation. These have been removed in the 1.1.4 pom. Assuming that we need to work with 1.1.3 for some reason, we can fix the metadata with the following rule:

Example 372. Rule to remove unused dependencies of Jaxen metadata

build.gradle

```
@CacheableRule
abstract class JaxenDependenciesRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        context.details.allVariants {
            withDependencies {
                removeAll { it.group in ["dom4j", "jdom", "xerces",  "maven-
plugins", "xml-apis", "xom"] }
            }
        }
    }
}
```

build.gradle.kts

```
@CacheableRule
abstract class JaxenDependenciesRule: ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        context.details.allVariants {
            withDependencies {
                removeAll { it.group in listOf("dom4j", "jdom", "xerces",
"maven-plugins", "xml-apis", "xom") }
            }
        }
    }
}
```

Within the `withDependencies` block you have access to the full list of dependencies and can use all methods available on the Java collection interface to inspect and modify that list. In addition, there are `add(notation, configureAction)` methods accepting the usual notations similar to declaring dependencies in the build script. Dependency constraints can be inspected and modified the same way in the `withDependencyConstraints` block.

If we take a closer look at the Jaxen 1.1.4 pom, we observe that the *dom4j*, *jdom* and *xerces* dependencies are still there but marked as *optional*. Optional dependencies in poms are not

automatically processed by Gradle nor Maven. The reason is that they indicate that there are optional feature variants provided by the Jaxen library which require one or more of these dependencies, but the information what these features are and which dependency belongs to which is missing. Such information cannot be represented in pom files, but in Gradle Module Metadata through variants and capabilities. Hence, we can add this information in a rule as well.

Example 373. Rule to add optional feature to Jaxen metadata

build.gradle

```
@CacheableRule
abstract class JaxenCapabilitiesRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        context.details.addVariant("runtime-dom4j", "runtime") {
            withCapabilities {
                removeCapability("jaxen", "jaxen")
                addCapability("jaxen", "jaxen-dom4j", context.details.id
.version)
            }
            withDependencies {
                add("dom4j:dom4j:1.6.1")
            }
        }
    }
}
```

build.gradle.kts

```
@CacheableRule
abstract class JaxenCapabilitiesRule: ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        context.details.addVariant("runtime-dom4j", "runtime") {
            withCapabilities {
                removeCapability("jaxen", "jaxen")
                addCapability("jaxen", "jaxen-dom4j",
context.details.id.version)
            }
            withDependencies {
                add("dom4j:dom4j:1.6.1")
            }
        }
    }
}
```

Here, we first use the `addVariant(name, baseVariant)` method to create an additional variant, which

we identify as *feature variant* by defining a new capability *jaxen-dom4j* to represent the optional dom4j integration feature of Jaxen. This works similar to defining optional feature variants in build scripts. We then use one of the `add` methods for adding dependencies to define which dependencies this optional feature needs.

In the build script, we can then add a dependency to the optional feature and Gradle will use the enriched metadata to discover the correct transitive dependencies.

Example 374. Applying and utilising rules for Jaxen metadata

build.gradle

```
dependencies {
    components {
        withModule("jaxen:jaxen", JaxenDependenciesRule)
        withModule("jaxen:jaxen", JaxenCapabilitiesRule)
    }
    implementation("jaxen:jaxen:1.1.3")
    runtimeOnly("jaxen:jaxen:1.1.3") {
        capabilities { requireCapability("jaxen:jaxen-dom4j") }
    }
}
```

build.gradle.kts

```
dependencies {
    components {
        withModule<JaxenDependenciesRule>("jaxen:jaxen")
        withModule<JaxenCapabilitiesRule>("jaxen:jaxen")
    }
    implementation("jaxen:jaxen:1.1.3")
    runtimeOnly("jaxen:jaxen:1.1.3") {
        capabilities { requireCapability("jaxen:jaxen-dom4j") }
    }
}
```

Making variants published as classified jars explicit

While in the previous example, all variants, "main variants" and optional features, were packaged in one jar file, it is common to publish certain variants as separate files. In particular, when the variants are mutual exclusive — i.e. they are **not** feature variants, but different variants offering alternative choices. One example **all** pom-based libraries already have are the *runtime* and *compile* variants, where Gradle can choose only one depending on the task at hand. Another of such alternatives discovered often in the Java ecosystems are jars targeting different Java versions.

As example, we look at version 0.7.9 of the asynchronous programming library Quasar published on Maven central. If we inspect the directory listing, we discover that a `quasar-core-0.7.9-jdk8.jar` was published, in addition to `quasar-core-0.7.9.jar`. Publishing additional jars with a *classifier* (here *jdk8*) is common practice in maven repositories. And while both Maven and Gradle allow you to reference such jars by classifier, they are not mentioned at all in the metadata. Thus, there is no information that these jars exist and if there are any other differences, like different dependencies, between the variants represented by such jars.

In Gradle Module Metadata, this variant information would be present and for the already published Quasar library, we can add it using the following rule:

Example 375. Rule to add JDK 8 variants to Quasar metadata

build.gradle

```
@CacheableRule
abstract class QuasarRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        ["compile", "runtime"].each { base ->
            context.details.addVariant("jdk8${base.capitalize()}", base) {
                attributes {
                    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
8)
                }
                withFiles {
                    removeAllFiles()
                    addFile("${context.details.id.name}-${context.details.id
.version}-jdk8.jar")
                }
            }
            context.details.withVariant(base) {
                attributes {
                    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
7)
                }
            }
        }
    }
}
```

build.gradle.kts

```
@CacheableRule
abstract class QuasarRule: ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        listOf("compile", "runtime").forEach { base ->
            context.details.addVariant("jdk8${base.capitalize()}", base) {
                attributes {
                    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
8)
                }
                withFiles {
                    removeAllFiles()
                    addFile("${context.details.id.name}-
${context.details.id.version}-jdk8.jar")
                }
            }
            context.details.withVariant(base) {
                attributes {
                    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
7)
                }
            }
        }
    }
}
```

In this case, it is pretty clear that the classifier stands for a target Java version, which is a known Java ecosystem attribute. Because we also need both a *compile* and *runtime* for Java 8, we create two new variants but use the existing *compile* and *runtime* variants as *base*. This way, all other Java ecosystem attributes are already set correctly and all dependencies are carried over. Then we set the `TARGET_JVM_VERSION_ATTRIBUTE` to `8` for both variants, remove any existing file from the new variants with `removeAllFiles()`, and add the jdk8 jar file with `addFile()`. The `removeAllFiles()` is needed, because the reference to the main jar `quasar-core-0.7.5.jar` is copied from the corresponding base variant.

We also enrich the existing *compile* and *runtime* variants with the information that they target Java 7 — `attribute(TARGET_JVM_VERSION_ATTRIBUTE, 7)`.

Now, we can request a Java 8 versions for all of our dependencies on the compile classpath in the build script and Gradle will automatically select the best fitting variant for each library. In the case of Quasar this will now be the *jdk8Compile* variant exposing the `quasar-core-0.7.9-jdk8.jar`.

Example 376. Applying and utilising rule for Quasar metadata

build.gradle

```
configurations.compileClasspath.attributes {
    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE, 8)
}
dependencies {
    components {
        withModule("co.paralleluniverse:quasar-core", QuasarRule)
    }
    implementation("co.paralleluniverse:quasar-core:0.7.9")
}
```

build.gradle.kts

```
configurations["compileClasspath"].attributes {
    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE, 8)
}
dependencies {
    components {
        withModule<QuasarRule>("co.paralleluniverse:quasar-core")
    }
    implementation("co.paralleluniverse:quasar-core:0.7.9")
}
```

Making variants encoded in versions explicit

Another solution to publish multiple alternatives for the same library is the usage of a a versioning pattern as done by the popular Guava library. Here, each new version is published twice by appending the classifier to the version instead of the jar artifact. In the case of Guava 28 for example, we can find a *28.0-jre* (Java 8) and *28.0-android* (Java 6) version on Maven central. The advantage of using this pattern when working only with pom metadata is that both variants are discoverable through the version. The disadvantage is that there is no information what the different version suffixes mean semantically. So in the case of conflict, Gradle would just pick the highest version when comparing the version strings.

Turning this into proper variants is a bit more tricky, as Gradle first selects a version of a module and then selects the best fitting variant. So the concept that variants are encoded as versions is not supported directly. However, since both variants are always published together we can assume that the files are physically located in the same repository. And since they are published with Maven repository conventions, we know the location of each file if we know module name and version. We can write the following rule:

Example 377. Rule to add JDK 6 and JDK 8 variants to Guava metadata

build.gradle

```
@CacheableRule
abstract class GuavaRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        def variantVersion = context.details.id.version
        def version = variantVersion.substring(0, variantVersion.indexOf("-"
))
        ["compile", "runtime"].each { base ->
            [6: "android", 8: "jre"].each { targetJvmVersion, jarName ->
                context.details.addVariant("jdk$targetJvmVersion${base
.capitalize()}", base) {
                    attributes {
                        attributes.attribute(TargetJvmVersion
.TARGET_JVM_VERSION_ATTRIBUTE, targetJvmVersion)
                    }
                    withFiles {
                        removeAllFiles()
                        addFile("guava-$version-${jarName}.jar", "../$version
-$jarName/guava-$version-${jarName}.jar")
                    }
                }
            }
        }
    }
}
```

build.gradle.kts

```
@CacheableRule
abstract class GuavaRule: ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        val variantVersion = context.details.id.version
        val version = variantVersion.substring(0, variantVersion.indexOf("-
"))
        listOf("compile", "runtime").forEach { base ->
            mapOf(6 to "android", 8 to "jre").forEach { (targetJvmVersion,
jarName) ->

context.details.addVariant("jdk$targetJvmVersion${base.capitalize()}", base)
{
                    attributes {

attributes.attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
targetJvmVersion)
                    }
                    withFiles {
                        removeAllFiles()
                        addFile("guava-$version-$jarName.jar", "../$version-
$jarName/guava-$version-$jarName.jar")
                    }
                }
            }
        }
    }
}
```

Similar to the previous example, we add runtime and compile variants for both Java versions. In the `withFiles` block however, we now also specify a relative path for the corresponding jar file which allows Gradle to find the file no matter if it has selected a *-jre* or *-android* version. The path is always relative to the location of the metadata (in this case `pom`) file of the selection module version. So with this rules, both Guava 28 "versions" carry both the *jdk6* and *jdk8* variants. So it does not matter to which one Gradle resolves. The variant, and with it the correct jar file, is determined based on the requested `TARGET_JVM_VERSION_ATTRIBUTE` value.

Example 378. Applying and utilising rule for Guava metadata

build.gradle

```
configurations.compileClasspath.attributes {
    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE, 6)
}
dependencies {
    components {
        withModule("com.google.guava:guava", GuavaRule)
    }
    // '23.3-android' and '23.3-jre' are now the same as both offer both
variants
    implementation("com.google.guava:guava:23.3+")
}
```

build.gradle.kts

```
configurations["compileClasspath"].attributes {
    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE, 6)
}
dependencies {
    components {
        withModule<GuavaRule>("com.google.guava:guava")
    }
    // '23.3-android' and '23.3-jre' are now the same as both offer both
variants
    implementation("com.google.guava:guava:23.3+")
}
```

Adding variants for native jars

Jars with classifiers are also used to separate parts of a library for which multiple alternatives exists, for example native code, from the main artifact. This is for example done by the Lightweight Java Game Library (LWGJ), which publishes several platform specific jars to Maven central from which always one is needed, in addition to the main jar, at runtime. It is not possible to convey this information in pom metadata as there is no concept of putting multiple artifacts in relation through the metadata. In Gradle Module Metadata, each variant can have arbitrary many files and we can leverage that by writing the following rule:

Example 379. Rule to add native runtime variants to LWGJ metadata

build.gradle

```
@CacheableRule
abstract class LwjglRule implements ComponentMetadataRule { //val os: String,
val arch: String, val classifier: String)
    private def nativeVariants = [
        [os: OperatingSystemFamily.LINUX,   arch: "arm32",  classifier:
"natives-linux-arm32"],
        [os: OperatingSystemFamily.LINUX,   arch: "arm64",  classifier:
"natives-linux-arm64"],
        [os: OperatingSystemFamily.WINDOWS, arch: "x86",    classifier:
"natives-windows-x86"],
        [os: OperatingSystemFamily.WINDOWS, arch: "x86-64", classifier:
"natives-windows"],
        [os: OperatingSystemFamily.MACOS,   arch: "x86-64", classifier:
"natives-macos"]
    ]

    @Inject abstract ObjectFactory getObjects()

    void execute(ComponentMetadataContext context) {
        context.details.withVariant("runtime") {
            attributes {
                attributes.attribute(OperatingSystemFamily
.OPERATING_SYSTEM_ATTRIBUTE, objects.named(OperatingSystemFamily, "none"))
                attributes.attribute(MachineArchitecture
.ARCHITECTURE_ATTRIBUTE, objects.named(MachineArchitecture, "none"))
            }
        }
        nativeVariants.each { variantDefinition ->
            context.details.addVariant("${variantDefinition.classifier}
-runtime", "runtime") {
                attributes {
                    attributes.attribute(OperatingSystemFamily
.OPERATING_SYSTEM_ATTRIBUTE, objects.named(OperatingSystemFamily,
variantDefinition.os))
                    attributes.attribute(MachineArchitecture
.ARCHITECTURE_ATTRIBUTE, objects.named(MachineArchitecture,
variantDefinition.arch))
                }
                withFiles {
                    addFile("${context.details.id.name}-${context.details.id
.version}-${variantDefinition.classifier}.jar")
                }
            }
        }
    }
}
```

build.gradle.kts

```
@CacheableRule
abstract class LwjglRule: ComponentMetadataRule {
    data class NativeVariant(val os: String, val arch: String, val
classifier: String)

    private val nativeVariants = listOf(
        NativeVariant(OperatingSystemFamily.LINUX,   "arm32",  "natives-
linux-arm32"),
        NativeVariant(OperatingSystemFamily.LINUX,   "arm64",  "natives-
linux-arm64"),
        NativeVariant(OperatingSystemFamily.WINDOWS, "x86",    "natives-
windows-x86"),
        NativeVariant(OperatingSystemFamily.WINDOWS, "x86-64", "natives-
windows"),
        NativeVariant(OperatingSystemFamily.MACOS,   "x86-64", "natives-
macos")
    )

    @get:Inject abstract val objects: ObjectFactory

    override fun execute(context: ComponentMetadataContext) {
        context.details.withVariant("runtime") {
            attributes {

attributes.attribute(OperatingSystemFamily.OPERATING_SYSTEM_ATTRIBUTE,
objects.named("none"))

attributes.attribute(MachineArchitecture.ARCHITECTURE_ATTRIBUTE,
objects.named("none"))
            }
        }
        nativeVariants.forEach { variantDefinition ->
            context.details.addVariant("${variantDefinition.classifier}-
runtime", "runtime") {
                attributes {

attributes.attribute(OperatingSystemFamily.OPERATING_SYSTEM_ATTRIBUTE,
objects.named(variantDefinition.os))

attributes.attribute(MachineArchitecture.ARCHITECTURE_ATTRIBUTE,
objects.named(variantDefinition.arch))
                }
                withFiles {
                    addFile("${context.details.id.name}-
${context.details.id.version}-${variantDefinition.classifier}.jar")
                }
            }
        }
```

```
    }
}
```

This rule is quite similar to the Quasar library example above. Only this time we have five different runtime variants we add and nothing we need to change for the compile variant. The runtime variants are all based on the existing *runtime* variant and we do not change any existing information. All Java ecosystem attributes, the dependencies and the main jar file stay part of each of the runtime variants. We only set the additional attributes `OPERATING_SYSTEM_ATTRIBUTE` and `ARCHITECTURE_ATTRIBUTE` which are defined as part of Gradle's native support. And we add the corresponding native jar file so that each runtime variant now carries two files: the main jar and the native jar.

In the build script, we can now request a specific variant and Gradle will fail with a selection error if more information is needed to make a decision.

Example 380. Applying and utilising rule for LWGJ metadata

build.gradle

```
configurations["runtimeClasspath"].attributes {
    attribute(OperatingSystemFamily.OPERATING_SYSTEM_ATTRIBUTE, objects.
named(OperatingSystemFamily, "windows"))
}
dependencies {
    components {
        withModule("org.lwjgl:lwjgl", LwjglRule)
    }
    implementation("org.lwjgl:lwjgl:3.2.3")
}
```

build.gradle.kts

```
configurations["runtimeClasspath"].attributes {
    attribute(OperatingSystemFamily.OPERATING_SYSTEM_ATTRIBUTE,
objects.named("windows"))
}
dependencies {
    components {
        withModule<LwjglRule>("org.lwjgl:lwjgl")
    }
    implementation("org.lwjgl:lwjgl:3.2.3")
}
```

Gradle fails to select a variant because a machine architecture needs to be chosen

```
> Could not resolve all files for configuration ':runtimeClasspath'.
   > Could not resolve org.lwjgl:lwjgl:3.2.3.
     Required by:
         project :
      > Cannot choose between the following variants of org.lwjgl:lwjgl:3.2.3:
          - natives-windows-runtime
          - natives-windows-x86-runtime
```

Making different flavors of a library available through capabilities

Because it is difficult to model optional feature variants as separate jars with pom metadata, libraries sometimes compose different jars with a different feature set. That is, instead of composing your flavor of the library from different feature variants, you select one of the pre-composed variants (offering everything in one jar). One such library is the well-known dependency injection framework Guice, published on Maven central, which offers a complete flavor (the main jar) and a reduced variant without aspect-oriented programming support (`guice-4.2.2-no_aop.jar`). That second variant with a classifier is not mentioned in the pom metadata. With the following rule, we create compile and runtime variants based on that file and make it selectable through a capability named `com.google.inject:guice-no_aop`.

Example 381. Rule to add no_aop feature variant to Guice metadata

build.gradle

```
@CacheableRule
abstract class GuiceRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        ["compile", "runtime"].each { base ->
            context.details.addVariant("noAop${base.capitalize()}", base) {
                withCapabilities {
                    addCapability("com.google.inject", "guice-no_aop",
context.details.id.version)
                }
                withFiles {
                    removeAllFiles()
                    addFile("guice-${context.details.id.version}-no_aop.jar")
                }
                withDependencies {
                    removeAll { it.group == "aopalliance" }
                }
            }
        }
    }
}
```

build.gradle.kts

```
@CacheableRule
abstract class GuiceRule: ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        listOf("compile", "runtime").forEach { base ->
            context.details.addVariant("noAop${base.capitalize()}", base) {
                withCapabilities {
                    addCapability("com.google.inject", "guice-no_aop",
context.details.id.version)
                }
                withFiles {
                    removeAllFiles()
                    addFile("guice-${context.details.id.version}-no_aop.jar")
                }
                withDependencies {
                    removeAll { it.group == "aopalliance" }
                }
            }
        }
    }
}
```

The new variants also have the dependency on the standardized aop interfaces library `aopalliance:aopalliance` removed, as this is clearly not needed by these variants. Again, this is information that cannot be expressed in pom metadata. We can now select a `guice-no_aop` variant and will get the correct jar file **and** the correct dependencies.

Example 382. Applying and utilising rule for Guice metadata

build.gradle

```
dependencies {
    components {
        withModule("com.google.inject:guice", GuiceRule)
    }
    implementation("com.google.inject:guice:4.2.2") {
        capabilities { requireCapability("com.google.inject:guice-no_aop") }
    }
}
```

build.gradle.kts

```
dependencies {
    components {
        withModule<GuiceRule>("com.google.inject:guice")
    }
    implementation("com.google.inject:guice:4.2.2") {
        capabilities { requireCapability("com.google.inject:guice-no_aop") }
    }
}
```

Adding missing capabilities to detect conflicts

Another usage of capabilities is to express that two different modules, for example `log4j` and `log4j-over-slf4j`, provide alternative implementations of the same thing. By declaring that both provide the same capability, Gradle only accepts one of them in a dependency graph. This example, and how it can be tackled with a component metadata rule, is described in detail in the feature modelling section.

Making Ivy modules variant-aware

Modules with Ivy metadata, do not have variants by default. However, *Ivy configurations* can be mapped to variants as the `addVariant(name, baseVariantOrConfiguration)` accepts any Ivy configuration that was published as base. This can be used, for example, to define runtime and compile variants. An example of a corresponding rule can be found here. Ivy details of Ivy configurations (e.g. dependencies and files) can also be modified using the `withVariant(configurationName)` API. However, modifying attributes or capabilities on Ivy configurations has no effect.

For very Ivy specific use cases, the component metadata rules API also offers access to other details only found in Ivy metadata. These are available through the IvyModuleDescriptor interface and can be accessed using `getDescriptor(IvyModuleDescriptor)` on the ComponentMetadataContext.

Example 383. Ivy component metadata rule

build.gradle

```
@CacheableRule
abstract class IvyComponentRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        def descriptor = context.getDescriptor(IvyModuleDescriptor)
        if (descriptor != null && descriptor.branch == "testing") {
            context.details.status = "rc"
        }
    }
}
```

build.gradle.kts

```
@CacheableRule
abstract class IvyComponentRule : ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        val descriptor = context.getDescriptor(IvyModuleDescriptor::class)
        if (descriptor != null && descriptor.branch == "testing") {
            context.details.status = "rc"
        }
    }
}
```

Filter using Maven metadata

For Maven specific use cases, the component metadata rules API also offers access to other details only found in POM metadata. These are available through the PomModuleDescriptor interface and can be accessed using `getDescriptor(PomModuleDescriptor)` on the ComponentMetadataContext.

Example 384. Access pom packaging type in component metadata rule

build.gradle

```
@CacheableRule
abstract class MavenComponentRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        def descriptor = context.getDescriptor(PomModuleDescriptor)
        if (descriptor != null && descriptor.packaging == "war") {
            // ...
        }
    }
}
```

build.gradle.kts

```
@CacheableRule
abstract class MavenComponentRule : ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        val descriptor = context.getDescriptor(PomModuleDescriptor::class)
        if (descriptor != null && descriptor.packaging == "war") {
            // ...
        }
    }
}
```

Modifying metadata on the component level for alignment

While all the examples above made modifications to variants of a component, there is also a limited set of modifications that can be done to the metadata of the component itself. This information can influence the version selection process for a module during dependency resolution, which is performed *before* one or multiple variants of a component are selected.

The first API available on the component is `belongsTo()` to create virtual platforms for aligning versions of multiple modules without Gradle Module Metadata. It is explained in detail in the section on aligning versions of modules not published with Gradle.

Modifying metadata on the component level for version selection based on status

Gradle and Gradle Module Metadata also allow attributes to be set on the whole component instead of a single variant. Each of these attributes carries special semantics as they influence version selection which is done *before* variant selection. While variant selection can handle any custom attribute, version selection only considers attributes for which specific semantics are implemented. At the moment, the only attribute with meaning here is `org.gradle.status`. It is therefore recommended to only modify this attribute, if any, on the component level. A dedicated API

`setStatus(value)` is available for this. To modify another attribute for all variants of a component `withAllVariants { attributes {} }` should be utilised instead.

A module's status is taken into consideration when a *latest* version selector is resolved. Specifically, `latest.someStatus` will resolve to the highest module version that has status `someStatus` or a more mature status. For example, `latest.integration` will select the highest module version regardless of its status (because `integration` is the least mature status as explained below), whereas `latest.release` will select the highest module version with status `release`.

The interpretation of the status can be influenced by changing a module's *status scheme* through the `setStatusScheme(valueList)` API. This concept models the different levels of maturity that a module transitions through over time with different publications. The default status scheme, ordered from least to most mature status, is `integration`, `milestone`, `release`. The `org.gradle.status` attribute must be set, to one of the values in the components status scheme. Thus each component always has a status which is determined from the metadata as follows:

- Gradle Module Metadata: the value that was published for the `org.gradle.status` attribute on the component
- Ivy metadata: `status` defined in the ivy.xml, defaults to `integration` if missing
- Pom metadata: `integration` for modules with a SNAPSHOT version, `release` for all others

The following example demonstrates `latest` selectors based on a custom status scheme declared in a component metadata rule that applies to all modules:

Example 385. Custom status scheme

build.gradle

```
@CacheableRule
abstract class CustomStatusRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        context.details.statusScheme = ["nightly", "milestone", "rc",
"release"]
        if (context.details.status == "integration") {
            context.details.status = "nightly"
        }
    }
}

dependencies {
    components {
        all(CustomStatusRule)
    }
    implementation("org.apache.commons:commons-lang3:latest.rc")
}
```

build.gradle.kts

```
@CacheableRule
abstract class CustomStatusRule : ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        context.details.statusScheme = listOf("nightly", "milestone", "rc",
"release")
        if (context.details.status == "integration") {
            context.details.status = "nightly"
        }
    }
}

dependencies {
    components {
        all<CustomStatusRule>()
    }
    implementation("org.apache.commons:commons-lang3:latest.rc")
}
```

Compared to the default scheme, the rule inserts a new status `rc` and replaces `integration` with `nightly`. Existing modules with the state `integration` are mapped to `nightly`.

Customizing resolution of a dependency directly

WARNING	This section covers mechanisms Gradle offers to directly influence the behavior of the dependency resolution engine. In contrast to the other concepts covered in this chapter, like dependency constraints or component metadata rules, which are all **inputs** to resolution, the following mechanisms allow you to write rules which are directly injected into the resolution engine. Because of this, they can be seen as *brute force* solutions, that may hide future problems (e.g. if new dependencies are added). Therefore, the general advice is to only use the following mechanisms if other means are not sufficient. If you are authoring a library, you should always prefer dependency constraints as they are published for your consumers.

Using dependency resolve rules

A dependency resolve rule is executed for each resolved dependency, and offers a powerful api for manipulating a requested dependency prior to that dependency being resolved. The feature currently offers the ability to change the group, name and/or version of a requested dependency, allowing a dependency to be substituted with a completely different module during resolution.

Dependency resolve rules provide a very powerful way to control the dependency resolution process, and can be used to implement all sorts of advanced patterns in dependency management. Some of these patterns are outlined below. For more information and code samples see the ResolutionStrategy class in the API documentation.

Implementing a custom versioning scheme

In some corporate environments, the list of module versions that can be declared in Gradle builds is maintained and audited externally. Dependency resolve rules provide a neat implementation of this pattern:

- In the build script, the developer declares dependencies with the module group and name, but uses a placeholder version, for example: `default`.
- The `default` version is resolved to a specific version via a dependency resolve rule, which looks up the version in a corporate catalog of approved modules.

This rule implementation can be neatly encapsulated in a corporate plugin, and shared across all builds within the organisation.

Example 386. Using a custom versioning scheme

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.version == 'default') {
            def version = findDefaultVersionInCatalog(details.requested.
group, details.requested.name)
            details.useVersion version.version
            details.because version.because
        }
    }
}

def findDefaultVersionInCatalog(String group, String name) {
    //some custom logic that resolves the default version into a specific
version
    [version: "1.0", because: 'tested by QA']
}
```

build.gradle.kts

```
configurations.all {
    resolutionStrategy.eachDependency {
        if (requested.version == "default") {
            val version = findDefaultVersionInCatalog(requested.group,
requested.name)
            useVersion(version.version)
            because(version.because)
        }
    }
}

data class DefaultVersion(val version: String, val because: String)

fun findDefaultVersionInCatalog(group: String, name: String): DefaultVersion
{
    //some custom logic that resolves the default version into a specific
version
    return DefaultVersion(version = "1.0", because = "tested by QA")
}
```

Denying a particular version with a replacement

Dependency resolve rules provide a mechanism for denying a particular version of a dependency and providing a replacement version. This can be useful if a certain dependency version is broken and should not be used, where a dependency resolve rule causes this version to be replaced with a known good version. One example of a broken module is one that declares a dependency on a library that cannot be found in any of the public repositories, but there are many other reasons why a particular module version is unwanted and a different version is preferred.

In example below, imagine that version `1.2.1` contains important fixes and should always be used in preference to `1.2`. The rule provided will enforce just this: any time version `1.2` is encountered it will be replaced with `1.2.1`. Note that this is different from a forced version as described above, in that any other versions of this module would not be affected. This means that the 'newest' conflict resolution strategy would still select version `1.3` if this version was also pulled transitively.

Example 387. Example: Blacklisting a version with a replacement

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.group == 'org.software' && details.requested
.name == 'some-library' && details.requested.version == '1.2') {
            details.useVersion '1.2.1'
            details.because 'fixes critical bug in 1.2'
        }
    }
}
```

build.gradle.kts

```
configurations.all {
    resolutionStrategy.eachDependency {
        if (requested.group == "org.software" && requested.name == "some-
library" && requested.version == "1.2") {
            useVersion("1.2.1")
            because("fixes critical bug in 1.2")
        }
    }
}
```

NOTE	There's a difference with using the *reject* directive of rich version constraints: rich versions will cause the build to fail if a rejected version is found in the graph, or select a non rejected version when using dynamic dependencies. Here, we *manipulate the requested versions* in order to select a different version when we find a rejected one. In other words, this is a *solution* to rejected versions, while rich version constraints allow declaring the *intent* (you should not use this version).

Using module replacement rules

It is preferable to express module conflicts in terms of capabilities conflicts. However, if there's no such rule declared or that you are working on versions of Gradle which do not support capabilities, Gradle provides tooling to work around those issues.

Module replacement rules allow a build to declare that a legacy library has been replaced by a new one. A good example when a new library replaced a legacy one is the `google-collections` -> `guava` migration. The team that created google-collections decided to change the module name from `com.google.collections:google-collections` into `com.google.guava:guava`. This is a legal scenario in the industry: teams need to be able to change the names of products they maintain, including the module coordinates. Renaming of the module coordinates has impact on conflict resolution.

To explain the impact on conflict resolution, let's consider the `google-collections` -> `guava` scenario. It may happen that both libraries are pulled into the same dependency graph. For example, *our project* depends on `guava` but some of *our dependencies* pull in a legacy version of `google-collections`. This can cause runtime errors, for example during test or application execution. Gradle does not automatically resolve the `google-collections` -> `guava` conflict because it is not considered as a *version conflict*. It's because the module coordinates for both libraries are completely different and conflict resolution is activated when `group` and `module` coordinates are the same but there are different versions available in the dependency graph (for more info, refer to the section on conflict resolution). Traditional remedies to this problem are:

- Declare exclusion rule to avoid pulling in `google-collections` to graph. It is probably the most popular approach.
- Avoid dependencies that pull in legacy libraries.
- Upgrade the dependency version if the new version no longer pulls in a legacy library.
- Downgrade to `google-collections`. It's not recommended, just mentioned for completeness.

Traditional approaches work but they are not general enough. For example, an organisation wants to resolve the `google-collections` -> `guava` conflict resolution problem in all projects. It is possible to declare that certain module was replaced by other. This enables organisations to include the information about module replacement in the corporate plugin suite and resolve the problem holistically for all Gradle-powered projects in the enterprise.

Example 388. Declaring a module replacement

build.gradle

```
dependencies {
    modules {
        module("com.google.collections:google-collections") {
            replacedBy("com.google.guava:guava", "google-collections is now
part of Guava")
        }
    }
}
```

build.gradle.kts

```
dependencies {
    modules {
        module("com.google.collections:google-collections") {
            replacedBy("com.google.guava:guava", "google-collections is now
part of Guava")
        }
    }
}
```

For more examples and detailed API, refer to the DSL reference for ComponentMetadataHandler.

What happens when we declare that `google-collections` is replaced by `guava`? Gradle can use this information for conflict resolution. Gradle will consider every version of `guava` newer/better than any version of `google-collections`. Also, Gradle will ensure that only guava jar is present in the classpath / resolved file list. Note that if only `google-collections` appears in the dependency graph (e.g. no `guava`) Gradle will not eagerly replace it with `guava`. Module replacement is an information that Gradle uses for resolving conflicts. If there is no conflict (e.g. only `google-collections` or only `guava` in the graph) the replacement information is not used.

Currently it is not possible to declare that a given module is replaced by a set of modules. However, it is possible to declare that multiple modules are replaced by a single module.

Using dependency substitution rules

Dependency substitution rules work similarly to dependency resolve rules. In fact, many capabilities of dependency resolve rules can be implemented with dependency substitution rules. They allow project and module dependencies to be transparently substituted with specified replacements. Unlike dependency resolve rules, dependency substitution rules allow project and module dependencies to be substituted interchangeably.

Adding a dependency substitution rule to a configuration changes the timing of when that configuration is resolved. Instead of being resolved on first use, the configuration is instead resolved when the task graph is being constructed. This can have unexpected consequences if the configuration is being further modified during task execution, or if the configuration relies on modules that are published during execution of another task.

To explain:

- A `Configuration` can be declared as an input to any Task, and that configuration can include project dependencies when it is resolved.
- If a project dependency is an input to a Task (via a configuration), then tasks to build the project artifacts must be added to the task dependencies.
- In order to determine the project dependencies that are inputs to a task, Gradle needs to resolve the `Configuration` inputs.
- Because the Gradle task graph is fixed once task execution has commenced, Gradle needs to perform this resolution prior to executing any tasks.

In the absence of dependency substitution rules, Gradle knows that an external module dependency will never transitively reference a project dependency. This makes it easy to determine the full set of project dependencies for a configuration through simple graph traversal. With this functionality, Gradle can no longer make this assumption, and must perform a full resolve in order to determine the project dependencies.

Substituting an external module dependency with a project dependency

One use case for dependency substitution is to use a locally developed version of a module in place of one that is downloaded from an external repository. This could be useful for testing a local, patched version of a dependency.

The module to be replaced can be declared with or without a version specified.

Example 389. Substituting a module with a project

build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute module("org.utils:api") using project(":api") because "we
work with the unreleased development version"
        substitute module("org.utils:util:2.5") using project(":util")
    }
}
```

build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(module("org.utils:api"))
            .using(project(":api")).because("we work with the unreleased
development version")
        substitute(module("org.utils:util:2.5")).using(project(":util"))
    }
}
```

Note that a project that is substituted must be included in the multi-project build (via `settings.gradle`). Dependency substitution rules take care of replacing the module dependency with the project dependency and wiring up any task dependencies, but do not implicitly include the project in the build.

Substituting a project dependency with a module replacement

Another way to use substitution rules is to replace a project dependency with a module in a multi-project build. This can be useful to speed up development with a large multi-project build, by allowing a subset of the project dependencies to be downloaded from a repository rather than being built.

The module to be used as a replacement must be declared with a version specified.

Example 390. Substituting a project with a module

build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute project(":api") using module("org.utils:api:1.3") because
"we use a stable version of org.utils:api"
    }
}
```

build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(project(":api"))
            .using(module("org.utils:api:1.3")).because("we use a stable
version of org.utils:api")
    }
}
```

When a project dependency has been replaced with a module dependency, that project is still included in the overall multi-project build. However, tasks to build the replaced dependency will not be executed in order to resolve the depending `Configuration`.

Conditionally substituting a dependency

A common use case for dependency substitution is to allow more flexible assembly of sub-projects within a multi-project build. This can be useful for developing a local, patched version of an external dependency or for building a subset of the modules within a large multi-project build.

The following example uses a dependency substitution rule to replace any module dependency with the group `org.example`, but only if a local project matching the dependency name can be located.

Example 391. Conditionally substituting a dependency

build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution.all { DependencySubstitution
dependency ->
        if (dependency.requested instanceof ModuleComponentSelector &&
dependency.requested.group == "org.example") {
            def targetProject = findProject(":${dependency.requested.module}
")
            if (targetProject != null) {
                dependency.useTarget targetProject
            }
        }
    }
}
```

build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution.all {
        requested.let {
            if (it is ModuleComponentSelector && it.group == "org.example") {
                val targetProject = findProject(":${it.module}")
                if (targetProject != null) {
                    useTarget(targetProject)
                }
            }
        }
    }
}
```

Note that a project that is substituted must be included in the multi-project build (via `settings.gradle`). Dependency substitution rules take care of replacing the module dependency with the project dependency, but do not implicitly include the project in the build.

Substituting a dependency with another variant

Gradle's dependency management engine is variant-aware meaning that for a single component, the engine may select different artifacts and transitive dependencies.

What to select is determined by the attributes of the consumer configuration and the attributes of the variants found on the producer side. It is, however, possible that some specific dependencies override attributes from the configuration itself. This is typically the case when using the Java

Platform plugin: this plugin builds a special kind of component which is called a "platform" and can be addressed by setting the component category attribute to `platform`, in opposition to typical dependencies which are targetting libraries.

Therefore, you may face situations where you want to substitute a platform dependency with a regular dependency, or the other way around.

Substituting a dependency with attributes

Let's imagine that you want to substitute a platform dependency with a regular dependency. This means that the library you are consuming declared something like this:

Example 392. An incorrect dependency on a platform

lib/build.gradle

```
dependencies {
    // This is a platform dependency but you want the library
    implementation platform('com.google.guava:guava:28.2-jre')
}
```

lib/build.gradle.kts

```
dependencies {
    // This is a platform dependency but you want the library
    implementation(platform("com.google.guava:guava:28.2-jre"))
}
```

The `platform` keyword is actually a short-hand notation for a *dependency with attributes*. If we want to substitute this dependency with a regular dependency, then we need to select precisely the dependencies which have the `platform` attribute.

This can be done by using a substitution rule:

Example 393. Substitute a platform dependency with a regular dependency

consumer/build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(platform(module('com.google.guava:guava:28.2-jre'))).
            using module('com.google.guava:guava:28.2-jre')
    }
}
```

consumer/build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(platform(module("com.google.guava:guava:28.2-jre")))
            .using(module("com.google.guava:guava:28.2-jre"))
    }
}
```

The same rule *without* the `platform` keyword would try to substitute *regular dependencies* with a regular dependency, which is not what you want, so it's important to understand that the substitution rules apply on a *dependency specification*: it matches the requested dependency (`substitute XXX`) with a substitute (`using YYY`).

You can have attributes on both the requested dependency *or* the substitute and the substitution is not limited to `platform`: you can actually specify the whole set of dependency attributes using the `variant` notation. The following rule is *strictly equivalent* to the rule above:

Example 394. Substitute a platform dependency with a regular dependency using the variant notation

consumer/build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute variant(module('com.google.guava:guava:28.2-jre')) {
            attributes {
                attribute(Category.CATEGORY_ATTRIBUTE, objects.named(
Category, Category.REGULAR_PLATFORM))
            }
        } using module('com.google.guava:guava:28.2-jre')
    }
}
```

consumer/build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(variant(module("com.google.guava:guava:28.2-jre")) {
            attributes {
                attribute(Category.CATEGORY_ATTRIBUTE,
objects.named(Category.REGULAR_PLATFORM))
            }
        }).using(module("com.google.guava:guava:28.2-jre"))
    }
}
```

Please refer to the Substitution DSL API docs for a complete reference of the variant substitution API.

WARNING	In composite builds, the rule that you have to match the exact requested dependency attributes is not applied: when using composites, Gradle will automatically match the requested attributes. In other words, it is implicit that if you include another build, you are substituting *all variants* of the substituted module with an equivalent variant in the included build.

Substituting a dependency with a dependency with capabilities

Similarly to attributes substitution, Gradle lets you substitute a dependency with or without capabilities with another dependency with or without capabilities.

For example, let's imagine that you need to substitute a regular dependency with its test fixtures instead. You can achieve this by using the following dependency substitution rule:

Example 395. Substitute a dependency with its test fixtures

build.gradle

```
configurations.testCompileClasspath {
    resolutionStrategy.dependencySubstitution {
        substitute(module('com.acme:lib:1.0'))
            .using variant(module('com.acme:lib:1.0')) {
            capabilities {
                requireCapability('com.acme:lib-test-fixtures')
            }
        }
    }
}
```

build.gradle.kts

```
configurations.testCompileClasspath {
    resolutionStrategy.dependencySubstitution {

substitute(module("com.acme:lib:1.0")).using(variant(module("com.acme:lib:1.0
")) {
            capabilities {
                requireCapability("com.acme:lib-test-fixtures")
            }
        })
    }
}
```

Capabilities which are declared in a substitution rule on the requested dependency constitute part of the dependency match specification, and therefore dependencies which do *not* require the capabilities will not be matched.

Please refer to the Substitution DSL API docs for a complete reference of the variant substitution API.

WARNING	In composite builds, the rule that you have to match the exact requested dependency capabilities is not applied: when using composites, Gradle will automatically match the requested capabilities. In other words, it is implicit that if you include another build, you are substituting *all variants* of the substituted module with an equivalent variant in the included build.

Substituting a dependency with a classifier or artifact

While external modules are in general addressed via their group/artifact/version coordinates, it is

common that such modules are published with additional artifacts that you may want to use in place of the main artifact. This is typically the case for *classified* artifacts, but you may also need to select an artifact with a different file type or extension. Gradle discourages use of classifiers in dependencies and prefers to model such artifacts as additional variants of a module. There are lots of advantages of using variants instead of classified artifacts, including, but not only, a different set of dependencies for those artifacts.

However, in order to help bridging the two models, Gradle provides means to change or remove a classifier in a substitution rule.

Example 396. Dependencies which will lead to a resolution error

consumer/build.gradle

```
dependencies {
    implementation 'com.google.guava:guava:28.2-jre'
    implementation 'co.paralleluniverse:quasar-core:0.8.0'
    implementation project(':lib')
}
```

consumer/build.gradle.kts

```
dependencies {
    implementation("com.google.guava:guava:28.2-jre")
    implementation("co.paralleluniverse:quasar-core:0.8.0")
    implementation(project(":lib"))
}
```

In the example above, the first level dependency on `quasar` makes us think that Gradle would resolve `quasar-core-0.8.0.jar` but it's not the case: the build would fail with this message:

```
Execution failed for task ':resolve'.
> Could not resolve all files for configuration ':runtimeClasspath'.
   > Could not find quasar-core-0.8.0-jdk8.jar (co.paralleluniverse:quasar-
core:0.8.0).
     Searched in the following locations:
         https://repo1.maven.org/maven2/co/paralleluniverse/quasar-core/0.8.0/quasar-
core-0.8.0-jdk8.jar
```

That's because there's a dependency on another project, `lib`, which itself depends on a different version of `quasar-core`:

Example 397. A "classified" dependency

lib/build.gradle

```
dependencies {
    implementation "co.paralleluniverse:quasar-core:0.7.12_r3:jdk8"
}
```

lib/build.gradle.kts

```
dependencies {
    implementation("co.paralleluniverse:quasar-core:0.7.12_r3:jdk8")
}
```

What happens is that Gradle would perform conflict resolution between `quasar-core` 0.8.0 and `quasar-core` 0.7.12_r3. Because 0.8.0 is higher, we select this version, but the dependency in `lib` has a classifier, `jdk8` and this classifier *doesn't exist anymore* in release 0.8.0.

To fix this problem, you can ask Gradle to resolve both dependencies *without classifier*:

Example 398. A resolution rule to disable selection of a classifier

consumer/build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute module('co.paralleluniverse:quasar-core') using module(
'co.paralleluniverse:quasar-core:0.8.0') withoutClassifier()
    }
}
```

consumer/build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(module("co.paralleluniverse:quasar-core"))
            .using(module("co.paralleluniverse:quasar-core:0.8.0"))
            .withoutClassifier()
    }
}
```

This rule effectively replaces any dependency on `quasar-core` found in the graph with a dependency without classifier.

Alternatively, it's possible to select a dependency *with* a specific classifier or, for more specific use cases, substitute with a very specific artifact (type, extension and classifier).

For more information, please refer to the following API documentation:

- artifact selection via the Substitution DSL
- artifact selection via the DependencySubstitution API
- artifact selection via the ResolutionStrategy API

Disabling transitive resolution

By default Gradle resolves all transitive dependencies specified by the dependency metadata. Sometimes this behavior may not be desirable e.g. if the metadata is incorrect or defines a large graph of transitive dependencies. You can tell Gradle to disable transitive dependency management for a dependency by setting ModuleDependency.setTransitive(boolean) to `false`. As a result only the main artifact will be resolved for the declared dependency.

Example 399. Disabling transitive dependency resolution for a declared dependency

build.gradle

```
dependencies {
    implementation('com.google.guava:guava:23.0') {
        transitive = false
    }
}
```

build.gradle.kts

```
dependencies {
    implementation("com.google.guava:guava:23.0") {
        isTransitive = false
    }
}
```

NOTE Disabling transitive dependency resolution will likely require you to declare the necessary runtime dependencies in your build script which otherwise would have been resolved automatically. Not doing so might lead to runtime classpath issues.

A project can decide to disable transitive dependency resolution completely. You either don't want to rely on the metadata published to the consumed repositories or you want to gain full control

over the dependencies in your graph. For more information, see Configuration.setTransitive(boolean).

Example 400. Disabling transitive dependency resolution on the configuration-level

build.gradle

```
configurations.all {
    transitive = false
}

dependencies {
    implementation 'com.google.guava:guava:23.0'
}
```

build.gradle.kts

```
configurations.all {
    isTransitive = false
}

dependencies {
    implementation("com.google.guava:guava:23.0")
}
```

Changing configuration dependencies prior to resolution

At times, a plugin may want to modify the dependencies of a configuration before it is resolved. The `withDependencies` method permits dependencies to be added, removed or modified programmatically.

Example 401. Modifying dependencies on a configuration

build.gradle

```
configurations {
    implementation {
        withDependencies { DependencySet dependencies ->
            ExternalModuleDependency dep = dependencies.find { it.name ==
'to-modify' } as ExternalModuleDependency
            dep.version {
                strictly "1.2"
            }
        }
    }
}
```

build.gradle.kts

```
configurations {
    create("implementation") {
        withDependencies {
            val dep = this.find { it.name == "to-modify" } as
ExternalModuleDependency
            dep.version {
                strictly("1.2")
            }
        }
    }
}
```

Setting default configuration dependencies

A configuration can be configured with default dependencies to be used if no dependencies are explicitly set for the configuration. A primary use case of this functionality is for developing plugins that make use of versioned tools that the user might override. By specifying default dependencies, the plugin can use a default version of the tool only if the user has not specified a particular version to use.

Example 402. Specifying default dependencies on a configuration

build.gradle

```
configurations {
    pluginTool {
        defaultDependencies { dependencies ->
            dependencies.add(project.dependencies.create("org.gradle:my-util:1.0"))
        }
    }
}
```

build.gradle.kts

```
configurations {
    create("pluginTool") {
        defaultDependencies {
            add(project.dependencies.create("org.gradle:my-util:1.0"))
        }
    }
}
```

Excluding a dependency from a configuration completely

Similar to excluding a dependency in a dependency declaration, you can exclude a transitive dependency for a particular configuration completely by using Configuration.exclude(java.util.Map). This will automatically exclude the transitive dependency for all dependencies declared on the configuration.

Example 403. Excluding transitive dependency for a particular configuration

build.gradle

```
configurations {
    implementation {
        exclude group: 'commons-collections', module: 'commons-collections'
    }
}

dependencies {
    implementation 'commons-beanutils:commons-beanutils:1.9.4'
    implementation 'com.opencsv:opencsv:4.6'
}
```

build.gradle.kts

```
configurations {
    "implementation" {
        exclude(group = "commons-collections", module = "commons-
collections")
    }
}

dependencies {
    implementation("commons-beanutils:commons-beanutils:1.9.4")
    implementation("com.opencsv:opencsv:4.6")
}
```

Matching dependencies to repositories

Gradle exposes an API to declare what a repository may or may not contain. This feature offers a fine grained control on which repository serve which artifacts, which can be one way of controlling the source of dependencies.

Head over to the section on repository content filtering to know more about this feature.

Enabling Ivy dynamic resolve mode

Gradle's Ivy repository implementations support the equivalent to Ivy's dynamic resolve mode. Normally, Gradle will use the `rev` attribute for each dependency definition included in an `ivy.xml` file. In dynamic resolve mode, Gradle will instead prefer the `revConstraint` attribute over the `rev` attribute for a given dependency definition. If the `revConstraint` attribute is not present, the `rev` attribute is used instead.

To enable dynamic resolve mode, you need to set the appropriate option on the repository definition. A couple of examples are shown below. Note that dynamic resolve mode is only available for Gradle's Ivy repositories. It is not available for Maven repositories, or custom Ivy `DependencyResolver` implementations.

Example 404. Enabling dynamic resolve mode

build.gradle

```
// Can enable dynamic resolve mode when you define the repository
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        resolve.dynamicMode = true
    }
}

// Can use a rule instead to enable (or disable) dynamic resolve mode for all
repositories
repositories.withType(IvyArtifactRepository) {
    resolve.dynamicMode = true
}
```

build.gradle.kts

```
// Can enable dynamic resolve mode when you define the repository
repositories {
    ivy {
        url = uri("http://repo.mycompany.com/repo")
        resolve.isDynamicMode = true
    }
}

// Can use a rule instead to enable (or disable) dynamic resolve mode for all
repositories
repositories.withType<IvyArtifactRepository> {
    resolve.isDynamicMode = true
}
```

Producing and Consuming Variants of Libraries

Declaring Capabilities of a Library

Capabilities as first-level concept

Components provide a number of features which are often orthogonal to the software architecture used to provide those features. For example, a library may include several features in a single artifact. However, such a library would be published at single GAV (group, artifact and version) coordinates. This means that, at single coordinates, potentially co-exist different "features" of a component.

With Gradle it becomes interesting to explicitly declare what features a component provides. For this, Gradle provides the concept of capability.

A feature is often built by combining different *capabilities*.

In an ideal world, components shouldn't declare dependencies on explicit GAVs, but rather express their requirements in terms of capabilities:

- "give me a component which provides logging"
- "give me a scripting engine"
- "give me a scripting engine that supports Groovy"

By modeling *capabilities*, the dependency management engine can be smarter and tell you whenever you have *incompatible capabilities* in a dependency graph, or ask you to choose whenever different modules in a graph provide the same *capability*.

Declaring capabilities for external modules

It's worth noting that Gradle supports declaring capabilities for components you build, but also for external components in case they didn't.

For example, if your build file contains the following dependencies:

Example 405. A build file with an implicit conflict of logging frameworks

build.gradle

```
dependencies {
    // This dependency will bring log4:log4j transitively
    implementation 'org.apache.zookeeper:zookeeper:3.4.9'

    // We use log4j over slf4j
    implementation 'org.slf4j:log4j-over-slf4j:1.7.10'
}
```

build.gradle.kts

```
dependencies {
    // This dependency will bring log4:log4j transitively
    implementation("org.apache.zookeeper:zookeeper:3.4.9")

    // We use log4j over slf4j
    implementation("org.slf4j:log4j-over-slf4j:1.7.10")
}
```

As is, it's pretty hard to figure out that you will end up with two logging frameworks on the classpath. In fact, `zookeeper` will bring in `log4j`, where what we want to use is `log4j-over-slf4j`. We can preemptively detect the conflict by adding a rule which will declare that both logging frameworks provide the same capability:

Example 406. A build file with an implicit conflict of logging frameworks

build.gradle

```
dependencies {
    // Activate the "LoggingCapability" rule
    components.all(LoggingCapability)
}

@CompileStatic
class LoggingCapability implements ComponentMetadataRule {
    final static Set<String> LOGGING_MODULES = ["log4j", "log4j-over-slf4j"]
as Set<String>

    void execute(ComponentMetadataContext context) {
        context.details.with {
            if (LOGGING_MODULES.contains(id.name)) {
                allVariants {
                    it.withCapabilities {
                        // Declare that both log4j and log4j-over-slf4j
provide the same capability
                        it.addCapability("log4j", "log4j", id.version)
                    }
                }
            }
        }
    }
}
```

build.gradle.kts

```
dependencies {
    // Activate the "LoggingCapability" rule
    components.all(LoggingCapability::class.java)
}

class LoggingCapability : ComponentMetadataRule {
    val loggingModules = setOf("log4j", "log4j-over-slf4j")

    override
    fun execute(context: ComponentMetadataContext) = context.details.run {
        if (loggingModules.contains(id.name)) {
            allVariants {
                withCapabilities {
                    // Declare that both log4j and log4j-over-slf4j provide
the same capability
                    addCapability("log4j", "log4j", id.version)
                }
            }
        }
    }
}
```

By adding this rule, we will make sure that Gradle *will* detect conflicts and properly fail:

```
> Could not resolve all files for configuration ':compileClasspath'.
   > Could not resolve org.slf4j:log4j-over-slf4j:1.7.10.
     Required by:
         project :
      > Module 'org.slf4j:log4j-over-slf4j' has been rejected:
           Cannot select module with conflict on capability 'log4j:log4j:1.7.10' also
provided by [log4j:log4j:1.2.16(compile)]
   > Could not resolve log4j:log4j:1.2.16.
     Required by:
         project : > org.apache.zookeeper:zookeeper:3.4.9
      > Module 'log4j:log4j' has been rejected:
           Cannot select module with conflict on capability 'log4j:log4j:1.2.16' also
provided by [org.slf4j:log4j-over-slf4j:1.7.10(compile)]
```

See the capabilities section of the documentation to figure out how to fix capability conflicts.

Declaring additional capabilities for a local component

All components have an *implicit capability* corresponding to the same GAV coordinates as the component. However, it is also possible to declare additional *explicit capabilities* for a component. This is convenient whenever a library published at different GAV coordinates is an *alternate*

implementation of the same API:

Example 407. Declaring capabilities of a component

build.gradle

```
configurations {
    apiElements {
        outgoing {
            capability("com.acme:my-library:1.0")
            capability("com.other:module:1.1")
        }
    }
    runtimeElements {
        outgoing {
            capability("com.acme:my-library:1.0")
            capability("com.other:module:1.1")
        }
    }
}
```

build.gradle.kts

```
configurations {
    apiElements {
        outgoing {
            capability("com.acme:my-library:1.0")
            capability("com.other:module:1.1")
        }
    }
    runtimeElements {
        outgoing {
            capability("com.acme:my-library:1.0")
            capability("com.other:module:1.1")
        }
    }
}
```

Capabilities must be attached to *outgoing configurations*, which are consumable configurations of a component.

This example shows that we declare two capabilities:

1. `com.acme:my-library:1.0`, which corresponds to the *implicit capability* of the library
2. `com.other:module:1.1`, which corresponds to another capability of this library

It's worth noting we need to do 1. because as soon as you start declaring *explicit* capabilities, then *all* capabilities need to be declared, including the *implicit* one.

The second capability can be specific to this library, or it can correspond to a capability provided by an external component. In that case, if `com.other:module` appears in the same dependency graph, the build will fail and consumers will have to choose what module to use.

Capabilities are published to Gradle Module Metadata. However, they have *no equivalent* in POM or Ivy metadata files. As a consequence, when publishing such a component, Gradle will warn you that this feature is only for Gradle consumers:

```
Maven publication 'maven' contains dependencies that cannot be represented in a
published pom file.
  - Declares capability com.acme:my-library:1.0
  - Declares capability com.other:module:1.1
```

Modeling feature variants and optional dependencies

Gradle supports the concept of *feature variants*: when building a library, it's often the case that some features should only be available when some dependencies are present, or when special artifacts are used.

Feature variants let consumers choose what *features* of a library they need: the dependency management engine will select the right artifacts and dependencies.

This allows for a number of different scenarios (list is non-exhaustive):

- a (better) substitute for Maven optional dependencies
- a *main* library is built with support for different mutually-exclusive implementations of runtime features; the user must choose one, and only one, implementation of each such feature
- a *main* library is built with support for optional runtime features, each of which requires a different set of dependencies
- a *main* library comes with secondary variants like *test fixtures*
- a *main* library comes with a main artifact, and enabling an additional feature requires additional artifacts

Selection of feature variants and capabilities

Declaring a dependency on a component is usually done by providing a set of coordinates (group, artifact, version also known as GAV coordinates). This allows the engine to determine the *component* we're looking for, but such a component may provide different *variants*. A *variant* is typically chosen based on the usage. For example, we might choose a different variant for compiling against a component (in which case we need the API of the component) or when executing code (in which case we need the runtime of the component). All variants of a component provide a number of capabilities, which are denoted similarly using GAV coordinates.

NOTE

A capability is denoted by GAV coordinates, but you must think of it as feature description:

- "I provide an SLF4J binding"
- "I provide runtime support for MySQL"
- "I provide a Groovy runtime"

And in general, having two components that provide the *same thing* in the graph is a problem (they conflict).

This is an important concept because:

- by default a variant provides a capability corresponding to the GAV coordinates of its component
- it is not allowed to have different components or different variants of a component in a dependency graph if they provide the same capability
- it is allowed to select two variants of the same component, as long as they provide *different capabilities*

A typical component will **only** provide variants with the default capability. A Java library, for example, exposes two variants (API and runtime) which provide the *same capability*. As a consequence, it is an error to have both the *API* and *runtime* of a single component in a dependency graph.

However, imagine that you need the *runtime* and the *test fixtures* of a component. Then it is allowed as long as the *runtime* and *test fixtures* variant of the library declare different capabilities.

If we do so, a consumer would then have to declare two dependencies:

- one on the "main" variant, the library
- one on the "test fixtures" variant, by *requiring its capability*

NOTE

While the engine supports feature variants independently of the ecosystem, this feature is currently only available using the Java plugins.

Declaring feature variants

Feature variants can be declared by applying the `java` or `java-library` plugins. The following code illustrates how to declare a feature named `mongodbSupport`:

Example 408. Declaring a feature variant

build.gradle

```
group = 'org.gradle.demo'
version = '1.0'

java {
    registerFeature('mongodbSupport') {
        usingSourceSet(sourceSets.main)
    }
}
```

build.gradle.kts

```
group = "org.gradle.demo"
version = "1.0"

java {
    registerFeature("mongodbSupport") {
        usingSourceSet(sourceSets["main"])
    }
}
```

Gradle will automatically setup a number of things for you, in a very similar way to how the Java Library Plugin sets up configurations:

- the configuration `mongodbSupportApi`, used to *declare API dependencies* for this feature
- the configuration `mongodbSupportImplementation`, used to *declare implementation dependencies* for this feature
- the configuration `mongodbSupportApiElements`, used by consumers to fetch the artifacts and API dependencies of this feature
- the configuration `mongodbSupportRuntimeElements`, used by consumers to fetch the artifacts and runtime dependencies of this feature

Most users will only need to care about the first two configurations, to declare the specific dependencies of this feature:

Example 409. Declaring dependencies of a feature

build.gradle

```
dependencies {
    mongodbSupportImplementation 'org.mongodb:mongodb-driver-sync:3.9.1'
}
```

build.gradle.kts

```
dependencies {
    "mongodbSupportImplementation"("org.mongodb:mongodb-driver-sync:3.9.1")
}
```

NOTE

By convention, Gradle maps the feature name to a capability whose group and version are the same as the group and version of the main component, respectively, but whose name is the main component name followed by a `-` followed by the kebab-cased feature name.

For example, if the group is `org.gradle.demo`, the name of the component is `provider`, its version is `1.0` and the feature is named `mongodbSupport`, the feature variant will be `org.gradle.demo:provider-mongodb-support:1.0`.

If you choose the capability name yourself or add more capabilities to a variant, it is recommended to follow the same convention.

Feature variant source set

In the previous example, we're declaring a feature variant which uses the *main source set*. This is a typical use case in the Java ecosystem, where it's, for whatever reason, not possible to split the sources of a project into different subprojects or different source sets. Gradle will therefore declare the configurations as described, but will also setup the compile classpath and runtime classpath of the *main source set* so that it extends from the feature configuration. Said differently, this allows you to declare the dependencies specific to a feature in their own "bucket", but everything is still compiled as a single source set. There will also be a single artifact (the component Jar) including support for all features.

However, it is often preferred to have a *separate source set* for a feature. Gradle will then perform a similar mapping, but will *not* make the compile and runtime classpath of the main component extend from the dependencies of the registered features. It will also, by convention, create a `Jar` task to bundle the classes built from this feature source set, using a classifier corresponding to the kebab-case name of the feature:

Example 410. Declaring a feature variant using a separate source set

build.gradle

```
sourceSets {
    mongodbSupport {
        java {
            srcDir 'src/mongodb/java'
        }
    }
}

java {
    registerFeature('mongodbSupport') {
        usingSourceSet(sourceSets.mongodbSupport)
    }
}
```

build.gradle.kts

```
sourceSets {
    create("mongodbSupport") {
        java {
            srcDir("src/mongodb/java")
        }
    }
}

java {
    registerFeature("mongodbSupport") {
        usingSourceSet(sourceSets["mongodbSupport"])
    }
}
```

Publishing feature variants

WARNING

Depending on the metadata file format, publishing feature variants may be lossy:

- using Gradle Module Metadata, everything is published and consumers will get the full benefit of feature variants
- using POM metadata (Maven), feature variants are published as **optional dependencies** and artifacts of feature variants are published with different *classifiers*
- using Ivy metadata, feature variants are published as extra configurations, which are *not* extended by the `default` configuration

Publishing feature variants is supported using the `maven-publish` and `ivy-publish` plugins only. The Java Plugin (or Java Library Plugin) will take care of registering the additional variants for you, so there's no additional configuration required, only the regular publications:

Example 411. Publishing a component with feature variants

build.gradle

```
plugins {
    id 'java-library'
    id 'maven-publish'
}
// ...
publishing {
    publications {
        myLibrary(MavenPublication) {
            from components.java
        }
    }
}
```

build.gradle.kts

```
plugins {
    `java-library`
    `maven-publish`
}
// ...
publishing {
    publications {
        create("myLibrary", MavenPublication::class.java) {
            from(components["java"])
        }
    }
}
```

Adding javadoc and sources JARs

Similar to the main Javadoc and sources JARs, you can configure the added feature variant so that it produces JARs for the Javadoc and sources. This however only makes sense when using a source set other than the main one.

Example 412. Producing javadoc and sources JARs for feature variants

build.gradle

```
java {
    registerFeature('mongodbSupport') {
        usingSourceSet(sourceSets.mongodbSupport)
        withJavadocJar()
        withSourcesJar()
    }
}
```

build.gradle.kts

```
java {
    registerFeature("mongodbSupport") {
        usingSourceSet(sourceSets["mongodbSupport"])
        withJavadocJar()
        withSourcesJar()
    }
}
```

Dependencies on feature variants

WARNING

As mentioned earlier, feature variants can be lossy when published. As a consequence, a consumer can depend on a feature variant only in these cases:

- with a project dependency (in a multi-project build)
- with Gradle Module Metadata available, that is the publisher MUST have published it
- within the Ivy world, by declaring a dependency on the configuration matching the feature

A consumer can specify that it needs a specific feature of a producer by declaring required capabilities. For example, if a producer declares a "MySQL support" feature like this:

Example 413. A library declaring a feature to support MySQL

build.gradle

```
group = 'org.gradle.demo'

java {
    registerFeature('mysqlSupport') {
        usingSourceSet(sourceSets.main)
    }
}

dependencies {
    mysqlSupportImplementation 'mysql:mysql-connector-java:8.0.14'
}
```

build.gradle.kts

```
group = "org.gradle.demo"

java {
    registerFeature("mysqlSupport") {
        usingSourceSet(sourceSets["main"])
    }
}

dependencies {
    "mysqlSupportImplementation"("mysql:mysql-connector-java:8.0.14")
}
```

Then the consumer can declare a dependency on the MySQL support feature by doing this:

Example 414. Consuming specific features in a multi-project build

build.gradle

```
dependencies {
    // This project requires the main producer component
    implementation(project(":producer"))

    // But we also want to use its MySQL support
    runtimeOnly(project(":producer")) {
        capabilities {
            requireCapability("org.gradle.demo:producer-mysql-support")
        }
    }
}
```

build.gradle.kts

```
dependencies {
    // This project requires the main producer component
    implementation(project(":producer"))

    // But we also want to use its MySQL support
    runtimeOnly(project(":producer")) {
        capabilities {
            requireCapability("org.gradle.demo:producer-mysql-support")
        }
    }
}
```

This will automatically bring the `mysql-connector-java` dependency on the runtime classpath. If there were more than one dependency, all of them would be brought, meaning that a feature can be used to group dependencies which contribute to a feature together.

Similarly, if an external library with feature variants was published with Gradle Module Metadata, it is possible to depend on a feature provided by that library:

Example 415. Consuming specific features from an external repository

build.gradle

```
dependencies {
    // This project requires the main producer component
    implementation('org.gradle.demo:producer:1.0')

    // But we also want to use its MongoDB support
    runtimeOnly('org.gradle.demo:producer:1.0') {
        capabilities {
            requireCapability("org.gradle.demo:producer-mongodb-support")
        }
    }
}
```

build.gradle.kts

```
dependencies {
    // This project requires the main producer component
    implementation("org.gradle.demo:producer:1.0")

    // But we also want to use its MongoDB support
    runtimeOnly("org.gradle.demo:producer:1.0") {
        capabilities {
            requireCapability("org.gradle.demo:producer-mongodb-support")
        }
    }
}
```

Handling mutually exclusive variants

The main advantage of using *capabilities* as a way to handle features is that you can precisely handle compatibility of variants. The rule is simple:

> It’s not allowed to have two variants of components that provide the same capability in a single dependency graph.

We can leverage that to ask Gradle to fail whenever the user mis-configures dependencies. Imagine, for example, that your library supports MySQL, Postgres and MongoDB, but that it’s only allowed to choose *one* of those at the same time. Not allowed should directly translate to "provide the same capability", so there must be a capability provided by all three features:

Example 416. A producer of multiple features that are mutually exclusive

build.gradle

```
java {
    registerFeature('mysqlSupport') {
        usingSourceSet(sourceSets.main)
        capability('org.gradle.demo', 'producer-db-support', '1.0')
        capability('org.gradle.demo', 'producer-mysql-support', '1.0')
    }
    registerFeature('postgresSupport') {
        usingSourceSet(sourceSets.main)
        capability('org.gradle.demo', 'producer-db-support', '1.0')
        capability('org.gradle.demo', 'producer-postgres-support', '1.0')
    }
    registerFeature('mongoSupport') {
        usingSourceSet(sourceSets.main)
        capability('org.gradle.demo', 'producer-db-support', '1.0')
        capability('org.gradle.demo', 'producer-mongo-support', '1.0')
    }
}

dependencies {
    mysqlSupportImplementation 'mysql:mysql-connector-java:8.0.14'
    postgresSupportImplementation 'org.postgresql:postgresql:42.2.5'
    mongoSupportImplementation 'org.mongodb:mongodb-driver-sync:3.9.1'
}
```

build.gradle.kts

```
java {
    registerFeature("mysqlSupport") {
        usingSourceSet(sourceSets["main"])
        capability("org.gradle.demo", "producer-db-support", "1.0")
        capability("org.gradle.demo", "producer-mysql-support", "1.0")
    }
    registerFeature("postgresSupport") {
        usingSourceSet(sourceSets["main"])
        capability("org.gradle.demo", "producer-db-support", "1.0")
        capability("org.gradle.demo", "producer-postgres-support", "1.0")
    }
    registerFeature("mongoSupport") {
        usingSourceSet(sourceSets["main"])
        capability("org.gradle.demo", "producer-db-support", "1.0")
        capability("org.gradle.demo", "producer-mongo-support", "1.0")
    }
}

dependencies {
    "mysqlSupportImplementation"("mysql:mysql-connector-java:8.0.14")
    "postgresSupportImplementation"("org.postgresql:postgresql:42.2.5")
    "mongoSupportImplementation"("org.mongodb:mongodb-driver-sync:3.9.1")
}
```

Here, the producer declares 3 variants, one for each database runtime support:

- `mysql-support` provides both the `db-support` and `mysql-support` capabilities
- `postgres-support` provides both the `db-support` and `postgres-support` capabilities
- `mongo-support` provides both the `db-support` and `mongo-support` capabilities

Then if the consumer tries to get both the `postgres-support` and `mysql-support` like this (this also works transitively):

Example 417. A consumer trying to use 2 incompatible variants at the same time

build.gradle

```
dependencies {
    implementation(project(":producer"))

    // Let's try to ask for both MySQL and Postgres support
    runtimeOnly(project(":producer")) {
        capabilities {
            requireCapability("org.gradle.demo:producer-mysql-support")
        }
    }
    runtimeOnly(project(":producer")) {
        capabilities {
            requireCapability("org.gradle.demo:producer-postgres-support")
        }
    }
}
```

build.gradle.kts

```
dependencies {
    // This project requires the main producer component
    implementation(project(":producer"))

    // Let's try to ask for both MySQL and Postgres support
    runtimeOnly(project(":producer")) {
        capabilities {
            requireCapability("org.gradle.demo:producer-mysql-support")
        }
    }
    runtimeOnly(project(":producer")) {
        capabilities {
            requireCapability("org.gradle.demo:producer-postgres-support")
        }
    }
}
```

Dependency resolution would fail with the following error:

```
Cannot choose between
   org.gradle.demo:producer:1.0 variant mysqlSupportRuntimeElements and
   org.gradle.demo:producer:1.0 variant postgresSupportRuntimeElements
   because they provide the same capability: org.gradle.demo:producer-db-support:1.0
```

Understanding variant selection

In other dependency management engines, like Apache Maven™, dependencies and artifacts are bound to a component that is published at a particular GAV (group-artifact-version) coordinates. The set of dependencies for this component are always the same, regardless of which artifact may be used from the component.

If the component does have multiple artifacts, each one is identified by a cumbersome *classifier*. There are no common semantics associated with classifiers and that makes it difficult to guarantee a globally consistent dependency graph. This means that nothing prevents multiple artifacts for a single component (e.g., `jdk7` and `jdk8` classifiers) from appearing in a classpath and causing hard to diagnose problems.

Maven component model

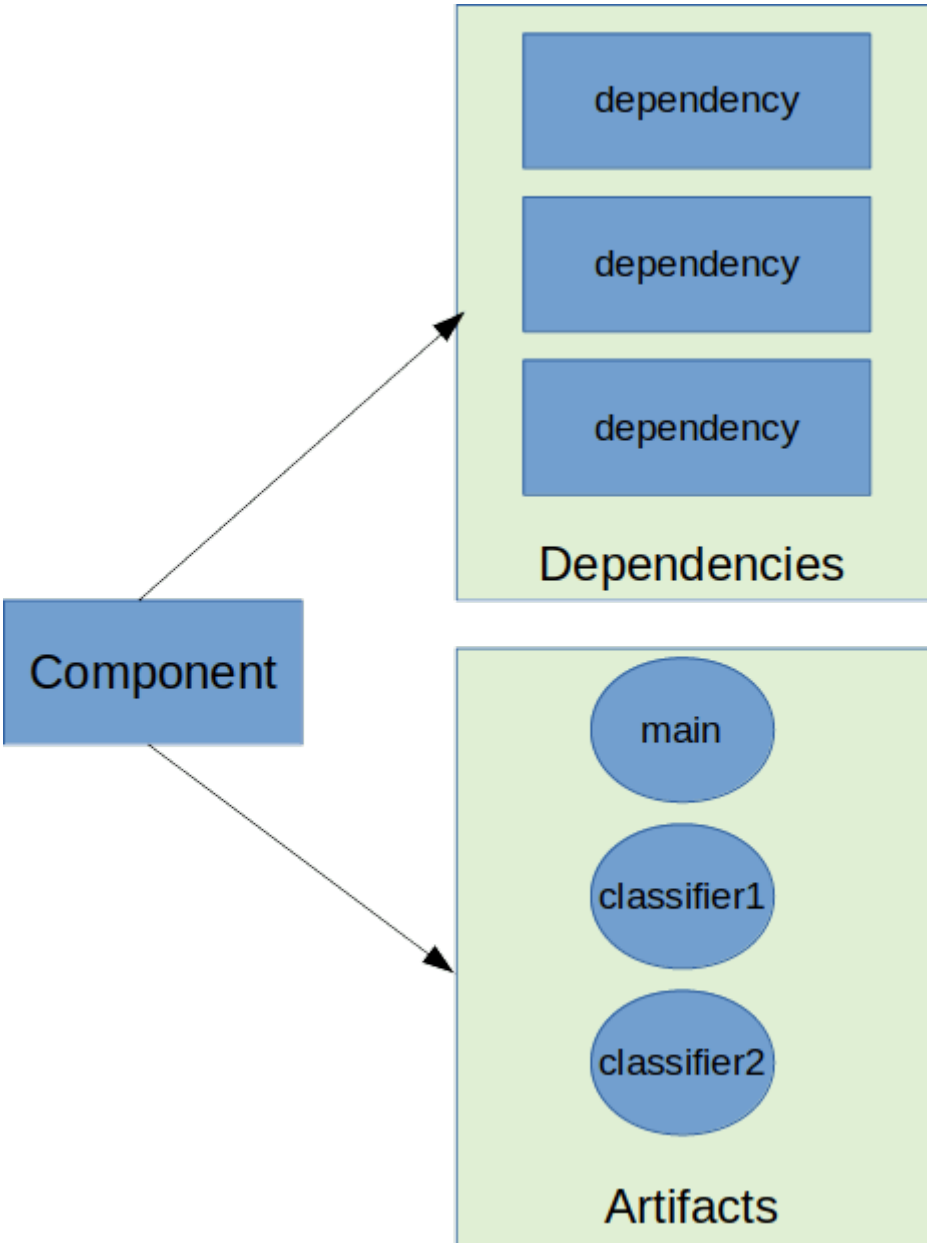


Figure 22. The Maven component model

Gradle component model

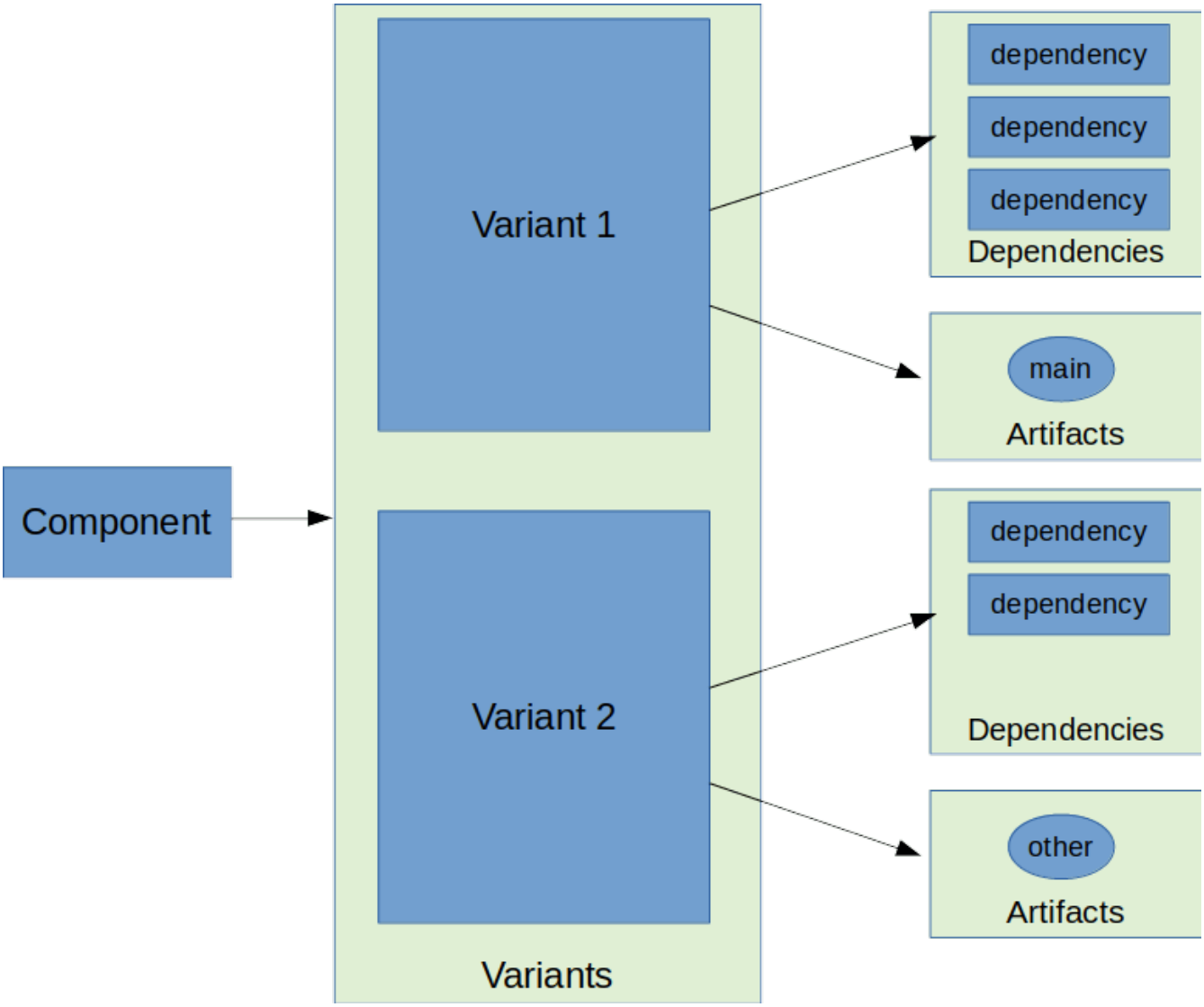


Figure 23. The Gradle component model

Gradle's dependency management engine is *variant aware*.

In addition to a component, Gradle has the concept of *variants* of a component. Variants correspond to the different ways a component can be used, such as for Java compilation or native linking or documentation. Artifacts are attached to a *variant* and each variant can have a different set of dependencies:

How does Gradle know which variant to choose when there's more than one? Variants are matched by use of attributes, which provide semantics to the variants and help the engine to produce a *consistent* resolution result.

Gradle differentiates between two kind of components: - local components (like projects), built from sources - external components, published to repositories

For local components, variants are mapped to consumable configurations. For external components, variants are defined by published Gradle Module Metadata or are derived from Ivy/Maven metadata.

NOTE	*Variants vs configurations* Variants and configurations are sometimes used interchangeably in the documentation, DSL or API for historical reasons. All components provide *variants* and those variants may be backed by a consumable configuration. Not all configurations are variants because they may be used for declaring or resolving dependencies.

Variant attributes

Attributes are type-safe key-value pairs that are defined by the consumer (for a resolvable configuration) and the producer (for each variant).

The consumer can define any number of attributes. Each attribute helps narrow the possible variants that can be selected. Attribute values do not need to be exact matches.

The variant can also define any number of attributes. The attributes should describe how the variant is intended to be used. For example, Gradle uses an attribute named `org.gradle.usage` to describe with how a component is used by the consumer (for compilation, for runtime execution, etc). It is not unusual for a variant to have more attributes than the consumer needs to provide to select it.

Variant attribute matching

NOTE	*About producer variants* The variant *name* is mostly for debugging purposes and error messages. The name does not participate variant matching—only its attributes do.

There are no restrictions on the number of variants a component can define. Usually, a component has at least an implementation variant, but it could also expose test fixtures, documentation or source code. A component may also expose *different variants* for different consumers for the same usage. For example, when compiling, a component could have different headers for Linux vs Windows vs macOS.

Gradle performs *variant aware selection* by matching the attributes requested by the consumer against attributes defined by the producer. The selection algorithm is detailed in another section.

NOTE	There are two exceptions to this rule that bypass variant aware resolution: - when a producer has no variants, a default artifact is chosen. - when a consumer *explicitly selects a configuration by name*, the artifacts of the configuration are chosen.

A simple example

Let's consider an example where a consumer is trying to use a library for compilation.

First, the consumer needs to explain how it's going to use the result of dependency resolution. This is done by setting *attributes* on the resolvable configuration of the consumer.

The consumer wants to resolve a variant that matches: - `org.gradle.usage=JAVA_API`

Second, the producer needs to expose the different variants of the component.

The producer component exposes 2 variants: - its API (named `apiElements`) with attribute `org.gradle.usage=JAVA_API` - its runtime (named `runtimeElements`) with attribute `org.gradle.usage=JAVA_RUNTIME`

Finally, Gradle selects the appropriate variant by looking at the variant attributes: - the consumer wants a variant with attributes `org.gradle.usage=JAVA_API` - the producer has a matching variant (`apiElements`) - the producer has a non-matching variant (`runtimeElements`)

Gradle provides the artifacts and dependencies from the `apiElements` variant to the consumer.

A more complicated example

In the real world, consumers and producers have more than one attribute.

A Java Library project in Gradle will involve several different attributes

- `org.gradle.usage` that describes how the variant is used
- `org.gradle.dependency.bundling` that describes how the variant handles dependencies (shadow jar vs fat jar vs regular jar)
- `org.gradle.libraryelements`, that describes the packaging of the variant (classes or jar)
- `org.gradle.jvm.version` that describes the *minimal version* of Java this variant targets
- `org.gradle.jvm.environment` that describes the type of JVM this variant targets

Let's consider an example where the consumer wants to run tests with a library on Java 8 and the producer supports two different Java versions (Java 8 and Java 11).

First, the consumer needs to explain which version of the Java it needs.

The consumer wants to resolve a variant that: - can be used at runtime (has `org.gradle.usage=JAVA_RUNTIME`) - can be run on *at least* Java 8 (`org.gradle.jvm.version=8`)

Second, the producer needs to expose the different variants of the component.

Like in the simple example, there is both a API (compilation) and runtime variant. These exist for both the Java 8 and Java 11 version of the component. - its API for Java 8 consumers (named `apiJava8Elements`) with attribute `org.gradle.usage=JAVA_API` and `org.gradle.jvm.version=8` - its runtime for Java 8 consumers (named `runtime8Elements`) with attribute `org.gradle.usage=JAVA_RUNTIME` and `org.gradle.jvm.version=8` - its API for Java 11 consumers (named `apiJava11Elements`) with attribute `org.gradle.usage=JAVA_API` and `org.gradle.jvm.version=11` - its runtime for Java 11 consumers (named `runtime11Elements`) with attribute `org.gradle.usage=JAVA_RUNTIME` and `org.gradle.jvm.version=11`

Finally, Gradle selects the best matching variant by looking at all of the attributes: - the consumer wants a variant with compatible attributes to `org.gradle.usage=JAVA_RUNTIME` and `org.gradle.jvm.version=8` - the variants `runtime8Elements` and `runtime11Elements` have `org.gradle.usage=JAVA_RUNTIME` - the variants `apiJava8Elements` and `apiJava11Elements` are incompatible - the variant `runtime8Elements` is compatible because it can run on Java 8 - the variant

`runtime11Elements` is incompatible because it cannot run on Java 8

Gradle provides the artifacts and dependencies from the `runtime8Elements` variant to the consumer.

NOTE

Compatibility of variants

What if the consumer sets `org.gradle.jvm.version` to 7?

Dependency resolution would *fail* with an error message explaining that there's no suitable variant. Gradle recognizes that the consumer wants a Java 7 compatible library and the *minimal* version of Java available on the producer is 8.

If the consumer requested `org.gradle.jvm.version=15`, then Gradle knows either the Java 8 or Java 11 variants could work. Gradle select the highest compatible Java version (11).

Variant selection errors

When selecting the most compatible variant of a component, resolution may fail: * when more than one variant from the producer matches the consumer attributes (ambiguity error) * when no variants from the producer match the consumer attributes (incompatibility error)

Dealing with ambiguity errors

An ambiguous variant selection looks like the following:

```
> Could not resolve all files for configuration ':compileClasspath'.
   > Could not resolve project :lib.
     Required by:
         project :ui
      > Cannot choose between the following variants of project :lib:
          - feature1ApiElements
          - feature2ApiElements
        All of them match the consumer attributes:
          - Variant 'feature1ApiElements' capability org.test:test-capability:1.0:
              - Unmatched attribute:
                  - Found org.gradle.category 'library' but wasn't required.
              - Compatible attributes:
                  - Provides org.gradle.dependency.bundling 'external'
                  - Provides org.gradle.jvm.version '11'
                  - Required org.gradle.libraryelements 'classes' and found value
'jar'.
                  - Provides org.gradle.usage 'java-api'
          - Variant 'feature2ApiElements' capability org.test:test-capability:1.0:
              - Unmatched attribute:
                  - Found org.gradle.category 'library' but wasn't required.
              - Compatible attributes:
                  - Provides org.gradle.dependency.bundling 'external'
                  - Provides org.gradle.jvm.version '11'
                  - Required org.gradle.libraryelements 'classes' and found value
'jar'.
                  - Provides org.gradle.usage 'java-api'
```

All *compatible* candidate variants are displayed with their attributes.

- Unmatched attributes are presented first, as they might be the missing piece in selecting the proper variant.
- Compatible attributes are presented second as they indicate what the consumer wanted and how these variants do match that request.
- There will not be any incompatible attributes as the variant would not be considered a candidate.

In the example above, the fix does not lie in attribute matching but in capability matching, which are shown next to the variant name. Because these two variants effectively provide the same attributes and capabilities, they cannot be disambiguated. So in this case, the fix is most likely to provide different capabilities on the producer side (`project :lib`) and express a capability choice on the consumer side (`project :ui`).

Dealing with no matching variant errors

A no matching variant error looks like the following:

```
> No variants of project :lib match the consumer attributes:
  - Configuration ':lib:compile':
      - Incompatible attribute:
          - Required artifactType 'dll' and found incompatible value 'jar'.
      - Other compatible attribute:
          - Provides usage 'api'
  - Configuration ':lib:compile' variant debug:
      - Incompatible attribute:
          - Required artifactType 'dll' and found incompatible value 'jar'.
      - Other compatible attributes:
          - Found buildType 'debug' but wasn't required.
          - Provides usage 'api'
  - Configuration ':lib:compile' variant release:
      - Incompatible attribute:
          - Required artifactType 'dll' and found incompatible value 'jar'.
      - Other compatible attributes:
          - Found buildType 'release' but wasn't required.
          - Provides usage 'api'
```

All *compatible* candidate variants are displayed with their attributes.

- Incompatible attributes are presented first, as they usually are the key in understanding why a variant could not be selected.
- Other attributes are presented second, this includes *required* and *compatible* ones as well as all extra *producer* attributes that are not requested by the consumer.

Similar to the ambiguous variant error, the goal is to understand which variant should be selected. In some cases, there may not be any compatible variants from the producer (e.g., trying to run on Java 8 with a library built for Java 11).

Visualizing variant information

Outgoing variants report

The report task `outgoingVariants` shows the list of variants available for selection by consumers of the project. It displays the capabilities, attributes and artifacts for each variant.

This task is similar to the `dependencyInsight` reporting task.

By default, `outgoingVariants` prints information about all variants. It offers the optional parameter `--variant <variantName>` to select a single variant to display. It also accepts the `--all` flag to include information about legacy and deprecated configurations.

Here is the output of the `outgoingVariants` task on a freshly generated `java-library` project:

```
> Task :outgoingVariants
--------------------------------------------------
Variant apiElements
--------------------------------------------------
```

```
Description = API elements for main.

Capabilities
    - new-java-library:lib:unspecified (default capability)
Attributes
    - org.gradle.category            = library
    - org.gradle.dependency.bundling = external
    - org.gradle.jvm.version         = 11
    - org.gradle.libraryelements     = jar
    - org.gradle.usage               = java-api
Artifacts
    - build/libs/lib.jar (artifactType = jar)

Secondary Variants (*)

    --------------------------------------------------
    Secondary Variant classes
    --------------------------------------------------
        Description = Directories containing compiled class files for main.

        Attributes
            - org.gradle.category            = library
            - org.gradle.dependency.bundling = external
            - org.gradle.jvm.version         = 11
            - org.gradle.libraryelements     = classes
            - org.gradle.usage               = java-api
        Artifacts
            - build/classes/java/main (artifactType = java-classes-directory)

--------------------------------------------------
Variant mainSourceElements (i)
--------------------------------------------------
Description = List of source directories contained in the Main SourceSet.

Capabilities
    - new-java-library:lib:unspecified (default capability)
Attributes
    - org.gradle.category            = verification
    - org.gradle.dependency.bundling = external
    - org.gradle.verificationtype    = main-sources
Artifacts
    - src/main/java (artifactType = directory)
    - src/main/resources (artifactType = directory)

--------------------------------------------------
Variant runtimeElements
--------------------------------------------------
Description = Elements of runtime for main.

Capabilities
    - new-java-library:lib:unspecified (default capability)
```

```
Attributes
    - org.gradle.category            = library
    - org.gradle.dependency.bundling = external
    - org.gradle.jvm.version         = 11
    - org.gradle.libraryelements     = jar
    - org.gradle.usage               = java-runtime
Artifacts
    - build/libs/lib.jar (artifactType = jar)

Secondary Variants (*)

    --------------------------------------------------
    Secondary Variant classes
    --------------------------------------------------
        Description = Directories containing compiled class files for main.

        Attributes
            - org.gradle.category            = library
            - org.gradle.dependency.bundling = external
            - org.gradle.jvm.version         = 11
            - org.gradle.libraryelements     = classes
            - org.gradle.usage               = java-runtime
        Artifacts
            - build/classes/java/main (artifactType = java-classes-directory)

    --------------------------------------------------
    Secondary Variant resources
    --------------------------------------------------
        Description = Directories containing the project's assembled resource files
for use at runtime.

        Attributes
            - org.gradle.category            = library
            - org.gradle.dependency.bundling = external
            - org.gradle.jvm.version         = 11
            - org.gradle.libraryelements     = resources
            - org.gradle.usage               = java-runtime
        Artifacts
            - build/resources/main (artifactType = java-resources-directory)

--------------------------------------------------
Variant testResultsElementsForTest (i)
--------------------------------------------------
Description = Directory containing binary results of running tests for the test Test
Suite's test target.

Capabilities
    - new-java-library:lib:unspecified (default capability)
Attributes
    - org.gradle.category              = verification
    - org.gradle.testsuite.name        = test
```

```
    - org.gradle.testsuite.target.name = test
    - org.gradle.testsuite.type        = unit-test
    - org.gradle.verificationtype      = test-results
Artifacts
    - build/test-results/test/binary (artifactType = directory)

(i) Configuration uses incubating attributes such as Category.VERIFICATION.
(*) Secondary variants are variants created via the Configuration#getOutgoing():
ConfigurationPublications API which also participate in selection, in addition to the
configuration itself.
```

From this you can see the two main variants that are exposed by a java library, `apiElements` and `runtimeElements`. Notice that the main difference is on the `org.gradle.usage` attribute, with values `java-api` and `java-runtime`. As they indicate, this is where the difference is made between what needs to be on the *compile* classpath of consumers, versus what's needed on the *runtime* classpath.

It also shows *secondary* variants, which are exclusive to Gradle projects and not published. For example, the secondary variant `classes` from `apiElements` is what allows Gradle to skip the JAR creation when compiling against a `java-library` project.

Resolvable configurations report

Gradle also offers a complimentary report task called `resolvableConfigurations` that displays the *resolvable* configurations of a project, which are those which can have dependencies added and be resolved. The report will list their attributes and any configurations that they extend. It will also list a summary of any attributes which will be affected by Compatibility Rules or Disambiguation Rules during resolution.

By default, `resolvableConfigurations` prints information about all purely resolvable configurations. These are configurations that are marked resolvable but **not** marked consumable. Though some resolvable configurations are also marked consumable, these are legacy configurations that should **not** have dependencies added in build scripts. This report offers the optional parameter `--configuration <configurationName>` to select a single configuration to display. It also accepts the `--all` flag to include information about legacy and deprecated configurations. Finally, it accepts the `--recursive` flag to list in the extended configurations section those configurations which are extended *transitively* rather than directly.

Here is the output of the `resolvableConfigurations` task on a freshly generated `java-library` project:

```
> Task :resolvableConfigurations
--------------------------------------------------
Configuration annotationProcessor
--------------------------------------------------
Description = Annotation processors and their dependencies for source set 'main'.

Attributes
    - org.gradle.category            = library
    - org.gradle.dependency.bundling = external
    - org.gradle.jvm.environment     = standard-jvm
```

```
    - org.gradle.libraryelements     = jar
    - org.gradle.usage               = java-runtime

--------------------------------------------------
Configuration compileClasspath
--------------------------------------------------
Description = Compile classpath for source set 'main'.

Attributes
    - org.gradle.category            = library
    - org.gradle.dependency.bundling = external
    - org.gradle.jvm.environment     = standard-jvm
    - org.gradle.jvm.version         = 11
    - org.gradle.libraryelements     = classes
    - org.gradle.usage               = java-api
Extended Configurations
    - compileOnly
    - implementation

--------------------------------------------------
Configuration runtimeClasspath
--------------------------------------------------
Description = Runtime classpath of source set 'main'.

Attributes
    - org.gradle.category            = library
    - org.gradle.dependency.bundling = external
    - org.gradle.jvm.environment     = standard-jvm
    - org.gradle.jvm.version         = 11
    - org.gradle.libraryelements     = jar
    - org.gradle.usage               = java-runtime
Extended Configurations
    - implementation
    - runtimeOnly

--------------------------------------------------
Configuration testAnnotationProcessor
--------------------------------------------------
Description = Annotation processors and their dependencies for source set 'test'.

Attributes
    - org.gradle.category            = library
    - org.gradle.dependency.bundling = external
    - org.gradle.jvm.environment     = standard-jvm
    - org.gradle.libraryelements     = jar
    - org.gradle.usage               = java-runtime

--------------------------------------------------
Configuration testCompileClasspath
--------------------------------------------------
Description = Compile classpath for source set 'test'.
```

```
Attributes
    - org.gradle.category            = library
    - org.gradle.dependency.bundling = external
    - org.gradle.jvm.environment     = standard-jvm
    - org.gradle.jvm.version         = 11
    - org.gradle.libraryelements     = classes
    - org.gradle.usage               = java-api
Extended Configurations
    - testCompileOnly
    - testImplementation

--------------------------------------------------
Configuration testRuntimeClasspath
--------------------------------------------------
Description = Runtime classpath of source set 'test'.

Attributes
    - org.gradle.category            = library
    - org.gradle.dependency.bundling = external
    - org.gradle.jvm.environment     = standard-jvm
    - org.gradle.jvm.version         = 11
    - org.gradle.libraryelements     = jar
    - org.gradle.usage               = java-runtime
Extended Configurations
    - testImplementation
    - testRuntimeOnly

--------------------------------------------------
Compatibility Rules
--------------------------------------------------
Description = The following Attributes have compatibility rules defined.

    - org.gradle.dependency.bundling
    - org.gradle.jvm.environment
    - org.gradle.jvm.version
    - org.gradle.libraryelements
    - org.gradle.plugin.api-version
    - org.gradle.usage

--------------------------------------------------
Disambiguation Rules
--------------------------------------------------
Description = The following Attributes have disambiguation rules defined.

    - org.gradle.category
    - org.gradle.dependency.bundling
    - org.gradle.jvm.environment
    - org.gradle.jvm.version
    - org.gradle.libraryelements
    - org.gradle.plugin.api-version
```

```
    - org.gradle.usage
```

From this you can see the two main configurations used to resolve dependencies, `compileClasspath` and `runtimeClasspath`, as well as their corresponding test configurations.

Mapping from Maven/Ivy to Gradle variants

Neither Maven nor Ivy have the concept of *variants*, which are only natively supported by Gradle Module Metadata. Gradle can still work with Maven and Ivy by using different variant derivation strategies.

> **NOTE**
>
> *Relationship with Gradle Module Metadata*
>
> Gradle Module Metadata is a metadata format for modules published on Maven, Ivy and other kinds of repositories. It is similar to the `pom.xml` or `ivy.xml` metadata file, but this format contains details about variants.
>
> See the Gradle Module Metadata specification for more information.

Mapping of Maven POM metadata to variants

Modules published on a Maven repository are automatically converted into variant-aware modules.

There is no way for Gradle to know which kind of component was published: - a BOM that represents a Gradle platform - a BOM used as a super-POM - a POM that is both a platform *and* a library

The default strategy used by Java projects in Gradle is to derive 8 different variants:

- 2 "library" variants (attribute `org.gradle.category` = `library`)
  - the `compile` variant maps the `<scope>compile</scope>` dependencies. This variant is equivalent to the `apiElements` variant of the Java Library plugin. All dependencies of this scope are considered *API dependencies*.
  - the `runtime` variant maps both the `<scope>compile</scope>` and `<scope>runtime</scope>` dependencies. This variant is equivalent to the `runtimeElements` variant of the Java Library plugin. All dependencies of those scopes are considered *runtime dependencies*.
    - in both cases, the `<dependencyManagement>` dependencies are *not converted to constraints*
- a "sources" variant that represents the sources jar for the component
- a "javadoc" variant that represents the javadoc jar for the component
- 4 "platform" variants derived from the `<dependencyManagement>` block (attribute `org.gradle.category` = `platform`):
  - the `platform-compile` variant maps the `<scope>compile</scope>` dependency management dependencies as *dependency constraints*.
  - the `platform-runtime` variant maps both the `<scope>compile</scope>` and `<scope>runtime</scope>` dependency management dependencies as *dependency constraints*.

- the `enforced-platform-compile` is similar to `platform-compile` but all the constraints are *forced*
- the `enforced-platform-runtime` is similar to `platform-runtime` but all the constraints are *forced*

You can understand more about the use of platform and enforced platforms variants by looking at the importing BOMs section of the manual. By default, whenever you declare a dependency on a Maven module, Gradle is going to look for the `library` variants. However, using the `platform` or `enforcedPlatform` keyword, Gradle is now looking for one of the "platform" variants, which allows you to import the constraints from the POM files, instead of the dependencies.

Mapping of Ivy files to variants

Gradle has no built-in derivation strategy implemented for Ivy files. Ivy is a flexible format that allows you to publish arbitrary files and can be heavily customized.

If you want to implement a derivation strategy for *compile* and *runtime* variants for Ivy, you can do so with component metadata rule. The component metadata rules API allows you to access Ivy configurations and create variants based on them. If you know that all the Ivy modules your are consuming have been published with Gradle without further customizations of the `ivy.xml` file, you can add the following rule to your build:

Example 418. Deriving compile and runtime variants for Ivy metadata

build.gradle

```
abstract class IvyVariantDerivationRule implements ComponentMetadataRule {
    final LibraryElements jarLibraryElements
    final Category libraryCategory
    final Usage javaRuntimeUsage
    final Usage javaApiUsage

    @Inject
    IvyVariantDerivationRule(ObjectFactory objectFactory) {
        jarLibraryElements = objectFactory.named(LibraryElements,
LibraryElements.JAR)
        libraryCategory = objectFactory.named(Category, Category.LIBRARY)
        javaRuntimeUsage = objectFactory.named(Usage, Usage.JAVA_RUNTIME)
        javaApiUsage = objectFactory.named(Usage, Usage.JAVA_API)
    }

    void execute(ComponentMetadataContext context) {
        // This filters out any non Ivy module
        if(context.getDescriptor(IvyModuleDescriptor) == null) {
            return
        }

        context.details.addVariant("runtimeElements", "default") {
            attributes {
                attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
jarLibraryElements)
                attribute(Category.CATEGORY_ATTRIBUTE, libraryCategory)
                attribute(Usage.USAGE_ATTRIBUTE, javaRuntimeUsage)
            }
        }
        context.details.addVariant("apiElements", "compile") {
            attributes {
                attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
jarLibraryElements)
                attribute(Category.CATEGORY_ATTRIBUTE, libraryCategory)
                attribute(Usage.USAGE_ATTRIBUTE, javaApiUsage)
            }
        }
    }
}

dependencies {
    components { all(IvyVariantDerivationRule) }
}
```

build.gradle.kts

```
abstract class IvyVariantDerivationRule @Inject internal
constructor(objectFactory: ObjectFactory) : ComponentMetadataRule {
    private val jarLibraryElements: LibraryElements
    private val libraryCategory: Category
    private val javaRuntimeUsage: Usage
    private val javaApiUsage: Usage

    init {
        jarLibraryElements = objectFactory.named(LibraryElements.JAR)
        libraryCategory = objectFactory.named(Category.LIBRARY)
        javaRuntimeUsage = objectFactory.named(Usage.JAVA_RUNTIME)
        javaApiUsage = objectFactory.named(Usage.JAVA_API)
    }

    override fun execute(context: ComponentMetadataContext) {
        // This filters out any non Ivy module
        if(context.getDescriptor(IvyModuleDescriptor::class) == null) {
            return
        }

        context.details.addVariant("runtimeElements", "default") {
            attributes {
                attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
jarLibraryElements)
                attribute(Category.CATEGORY_ATTRIBUTE, libraryCategory)
                attribute(Usage.USAGE_ATTRIBUTE, javaRuntimeUsage)
            }
        }
        context.details.addVariant("apiElements", "compile") {
            attributes {
                attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
jarLibraryElements)
                attribute(Category.CATEGORY_ATTRIBUTE, libraryCategory)
                attribute(Usage.USAGE_ATTRIBUTE, javaApiUsage)
            }
        }
    }
}

dependencies {
    components { all<IvyVariantDerivationRule>() }
}
```

The rule creates an `apiElements` variant based on the `compile` configuration and a `runtimeElements` variant based on the `default` configuration of each ivy module. For each variant, it sets the corresponding Java ecosystem attributes. Dependencies and artifacts of the variants are taken from

the underlying configurations. If not all consumed Ivy modules follow this pattern, the rule can be adjusted or only applied to a selected set of modules.

For all Ivy modules without variants, Gradle has a fallback selection method. Gradle does *not* perform variant aware resolution and instead selects either the `default` configuration or an explicitly named configuration.

Working with Variant Attributes

As explained in the section on variant aware matching, attributes give semantics to variants and are used by Gradle's dependency management engine to select the best matching variant.

As a user of Gradle, attributes are often hidden as implementation details. But it might be useful to understand the *standard attributes* defined by Gradle and its core plugins.

As a plugin author, these attributes, and the way they are defined, can serve as a basis for building your own set of attributes in your eco system plugin.

Standard attributes defined by Gradle

Gradle defines a list of standard attributes used by Gradle's core plugins.

Ecosystem-independent standard attributes

Table 17. Ecosystem-independent standard variant attributes

Attribute name	Description	Values	compatibility and disambiguation rules
`org.gradle.usage`	Indicates main purpose of variant	`Usage` values built from constants defined in Usage	Following ecosystem semantics (e.g. `java-runtime` can be used in place of `java-api` but not the opposite)
`org.gradle.category`	Indicates the category of this software component	`Category` values built from constants defined in Category	Following ecosystem semantics (e.g. `library` is default on the JVM, no compatibility otherwise)
`org.gradle.libraryelements`	Indicates the contents of a `org.gradle.category=library` variant	`LibraryElements` values built from constants defined in LibraryElements	Following ecosystem semantics(e.g. in the JVM world, `jar` is the default and is compatible with `classes`)
`org.gradle.docstype`	Indicates the contents of a `org.gradle.category=documentation` variant	`DocsType` values built from constants defined in DocsType	No default, no compatibility
`org.gradle.dependency.bundling`	Indicates how dependencies of a variant are accessed.	`Bundling` values built from constants defined in Bundling	Following ecosystem semantics (e.g. in the JVM world, `embedded` is compatible with `external`)

Attribute name	Description	Values	compatibility and disambiguation rules
`org.gradle.verificationtype`	Indicates what kind of verification task produced this output.	`VerificationType` values built from constants defined in VerificationType	No default, no compatibility

WARNING

When the `Category` attribute is present with the incubating value `org.gradle.category=verification` on a variant, that variant is considered to be a verification-time only variant.

These variants are meant to contain only the results of running verification tasks, such as test results or code coverage reports. They are **not publishable**, and will produce an error if added to a component which is published.

Table 18. Ecosystem-independent standard component attributes

Attribute name	Description	Values	compatibility and disambiguation rules
`org.gradle.status`	Component level attribute, derived	Based on a status scheme, with a default one existing based on the source repository.	Based on the scheme in use

JVM ecosystem specific attributes

In addition to the ecosystem independent attributes defined above, the JVM ecosystem adds the following attribute:

Table 19. JVM ecosystem standard component attributes

Attribute name	Description	Values	compatibility and disambiguation rules
`org.gradle.jvm.version`	Indicates the JVM version compatibility.	Integer using the version after the `1.` for Java 1.4 and before, the major version for Java 5 and beyond.	Defaults to the JVM version used by Gradle, lower is compatible with higher, prefers highest compatible.
`org.gradle.jvm.environment`	Indicates that a variant is optimized for a certain JVM environment.	Common values are `standard-jvm` and `android`. Other values are allowed.	The attribute is used to prefer one variant over another if multiple are available, but in general all values are compatible. The default is `standard-jvm`.
`org.gradle.testsuite.name`	Indicates the name of the TestSuite that produced this output.	Value is the name of the Suite.	No default, no compatibility

Attribute name	Description	Values	compatibility and disambiguation rules
`org.gradle.testsuite.target.name`	Indicates the name of the TestSuiteTarget that produced this output.	Value is the name of the Target.	No default, no compatibility
`org.gradle.testsuite.type`	Indicates the type of test suite (unit test, integration test, performance test, etc.)	`TestSuiteType` values built from constants defined in TestSuiteType or other custom values for user-defined test suite types.	No default, no compatibility

The JVM ecosystem also contains a number of compatibility and disambiguation rules over the different attributes. The reader willing to know more can take a look at the code for `org.gradle.api.internal.artifacts.JavaEcosystemSupport`.

Native ecosystem specific attributes

In addition to the ecosystem independent attributes defined above, the native ecosystem adds the following attributes:

Table 20. Native ecosystem standard component attributes

Attribute name	Description	Values	compatibility and disambiguation rules
`org.gradle.native.debuggable`	Indicates if the binary was built with debugging symbols	Boolean	N/A
`org.gradle.native.optimized`	Indicates if the binary was built with optimization flags	Boolean	N/A
`org.gradle.native.architecture`	Indicates the target architecture of the binary	`MachineArchitecture` values built from constants defined in MachineArchitecture	None
`org.gradle.native.operatingSystem`	Indicates the target operating system of the binary	`OperatingSystemFamily` values built from constants defined in OperatingSystemFamily	None

Gradle plugin ecosystem specific attributes

For Gradle plugin development, the following attribute is supported since Gradle 7.0. A Gradle plugin variant can specify compatibility with a Gradle API version through this attribute.

Table 21. Gradle plugin ecosystem standard component attributes

Attribute name	Description	Values	compatibility and disambiguation rules
`org.gradle.plugin.api-version`	Indicates the Gradle API version compatibility.	Valid Gradle version strings.	Defaults to the currently running Gradle, lower is compatible with higher, prefers highest compatible.

Declaring custom attributes

If you are extending Gradle, e.g. by writing a plugin for another ecosystem, declaring custom attributes could be an option if you want to support variant-aware dependency management features in your plugin. However, you should be cautious if you also attempt to publish libraries. Semantics of new attributes are usually defined through a plugin, which can carry compatibility and disambiguation rules. Consequently, builds that consume libraries published for a certain ecosystem, also need to apply the corresponding plugin to interpret attributes correctly. If your plugin is intended for a larger audience, i.e. if it is openly available and libraries are published to public repositories, defining new attributes effectively extends the semantics of Gradle Module Metadata and comes with responsibilities. E.g., support for attributes that are already published should not be removed again, or should be handled in some kind of compatibility layer in future versions of the plugin.

Creating attributes in a build script or plugin

Attributes are *typed*. An attribute can be created via the `Attribute<T>.of` method:

Example 419. Define attributes

build.gradle

```
// An attribute of type `String`
def myAttribute = Attribute.of("my.attribute.name", String)
// An attribute of type `Usage`
def myUsage = Attribute.of("my.usage.attribute", Usage)
```

build.gradle.kts

```
// An attribute of type `String`
val myAttribute = Attribute.of("my.attribute.name", String::class.java)
// An attribute of type `Usage`
val myUsage = Attribute.of("my.usage.attribute", Usage::class.java)
```

Attribute types support most Java primitive classes; such as `String` and `Integer`; Or anything extending `org.gradle.api.Named`. Attributes must be declared in the *attribute schema* found on the `dependencies` handler:

Example 420. Registering attributes on the attributes schema

build.gradle

```
dependencies.attributesSchema {
    // registers this attribute to the attributes schema
    attribute(myAttribute)
    attribute(myUsage)
}
```

build.gradle.kts

```
dependencies.attributesSchema {
    // registers this attribute to the attributes schema
    attribute(myAttribute)
    attribute(myUsage)
}
```

Then configurations can be configured to set values for attributes:

Example 421. Setting attributes on configurations

build.gradle

```
configurations {
    myConfiguration {
        attributes {
            attribute(myAttribute, 'my-value')
        }
    }
}
```

build.gradle.kts

```
configurations {
    create("myConfiguration") {
        attributes {
            attribute(myAttribute, "my-value")
        }
    }
}
```

For attributes which type extends `Named`, the value of the attribute **must** be created via the *object factory*:

Example 422. Named attributes

build.gradle

```
configurations {
    myConfiguration {
        attributes {
            attribute(myUsage, project.objects.named(Usage, 'my-value'))
        }
    }
}
```

build.gradle.kts

```
configurations {
    "myConfiguration" {
        attributes {
            attribute(myUsage, project.objects.named(Usage::class.java, "my-
value"))
        }
    }
}
```

Attribute matching

Attribute compatibility rules

Attributes let the engine select *compatible variants*. There are cases where a producer may not have *exactly* what the consumer requests but has a variant that can be used.

For example, if the consumer is asking for the API of a library and the producer doesn't have an exactly matching variant, the *runtime* variant could be considered compatible. This is typical of libraries published to external repositories. In this case, we know that even if we don't have an exact match (API), we can still compile against the runtime variant (it contains *more* than what we need to compile but it's still ok to use).

Gradle provides attribute compatibility rules that can be defined for each attribute. The role of a compatibility rule is to explain which attribute values are *compatible* based on what the consumer asked for.

Attribute compatibility rules have to be registered via the attribute matching strategy that you can obtain from the attributes schema.

Attribute disambiguation rules

Since multiple values for an attribute can be *compatible*, Gradle needs to choose the "best" candidate between all compatible candidates. This is called "disambiguation".

This is done by implementing an attribute disambiguation rule.

Attribute disambiguation rules have to be registered via the attribute matching strategy that you can obtain from the attributes schema.

Variant attribute matching algorithm

Finding the best variant can get complicated when there are many different variants available for a component and many different attributes. Gradle's dependency resolution engine performs the following algorithm when finding the best result (or failing):

1. Each candidate's attribute value is compared to the consumer's requested attribute value. A candidate is considered compatible if its value matches the consumer's value exactly, passes the attribute's compatibility rule or is not provided.
2. If only one candidate is considered compatible, that candidate wins.
3. If several candidates are compatible, but one of the candidates matches all of the same attributes as the other candidates, Gradle chooses that candidate. This is the candidate with the "longest" match.
4. If several candidates are compatible and are compatible with an equal number of attributes, Gradle needs to disambiguate the candidates.
    1. For each requested attribute, if a candidate does not have a value matching the disambiguation rule, it's eliminated from consideration.
    2. If the attribute has a known precedence, Gradle will stop as soon as there is a single candidate remaining.
    3. If the attribute does not have a known precedence, Gradle must consider all attributes.
5. If several candidates still remain, Gradle will start to consider "extra" attributes to disambiguate between multiple candidates. Extra attributes are attributes that were not requested by the consumer but are present on at least one candidate. These extra attributes are considered in precedence order.
    1. If the attribute has a known precedence, Gradle will stop as soon as there is a single candidate remaining.
    2. After all extra attributes with precedence are considered, the remaining candidates can be chosen if they are compatible with all of the non-ordered disambiguation rules.
6. If several candidates still remain, Gradle will consider extra attributes again. A candidate can be chosen if it has the fewest number of extra attributes.

If at any step no candidates remain compatible, resolution fails. Additionally, Gradle outputs a list of all compatible candidates from step 1 to help with debugging variant matching failures.

Plugins and ecosystems can influence the selection algorithm by implementing compatibility rules, disambiguation rules and telling Gradle the precedence of attributes. Attributes with a higher

precedence are used to eliminate compatible matches in order.

For example, in the Java ecosystem, the `org.gradle.usage` attribute has a higher precedence than `org.gradle.libraryelements`. This means that if two candidates were available with compatible values for both `org.gradle.usage` and `org.gradle.libraryelements`, Gradle will choose the candidate that passes the disambiguation rule for `org.gradle.usage`.

Sharing outputs between projects

A common pattern, in multi-project builds, is that one project consumes the artifacts of another project. In general, the simplest consumption form in the Java ecosystem is that when `A` depends on `B`, then `A` would depend on the `jar` produced by project `B`. As previously described in this chapter, this is modeled by `A` depending on a *variant of* `B`, where the variant is selected based on the needs of `A`. For compilation, we need the API dependencies of `B`, provided by the `apiElements` variant. For runtime, we need the runtime dependencies of `B`, provided by the `runtimeElements` variant.

However, what if you need a *different* artifact than the main one? Gradle provides, for example, built-in support for depending on the test fixtures of another project, but sometimes the artifact you need to depend on simply isn't exposed as a variant.

In order to be *safe to share* between projects and allow maximum performance (parallelism), such artifacts must be exposed via *outgoing configurations*.

WARNING

Don't reference other project tasks directly

A frequent anti-pattern to declare cross-project dependencies is:

```
dependencies {
   // this is unsafe!
   implementation project(":other").tasks.someOtherJar
}
```

This publication model is *unsafe* and can lead to non-reproducible and hard to parallelize builds. This section explains how to *properly create cross-project boundaries* by defining "exchanges" between projects by using *variants*.

There are two, complementary, options to share artifacts between projects. The simplified version is only suitable if what you need to share is a simple artifact that doesn't depend on the consumer. The simple solution is also limited to cases where this artifact is not published to a repository. This also implies that the consumer does not publish a dependency to this artifact. In cases where the consumer resolves to different artifacts in different contexts (e.g., different target platforms) or that publication is required, you need to use the advanced version.

Simple sharing of artifacts between projects

First, a producer needs to declare a configuration which is going to be *exposed* to consumers. As explained in the configurations chapter, this corresponds to a *consumable configuration*.

Let's imagine that the consumer requires *instrumented classes* from the producer, but that this

artifact is *not* the main one. The producer can expose its instrumented classes by creating a configuration that will "carry" this artifact:

Example 423. Declaring an outgoing variant

producer/build.gradle

```
configurations {
    instrumentedJars {
        canBeConsumed = true
        canBeResolved = false
        // If you want this configuration to share the same dependencies,
otherwise omit this line
        extendsFrom implementation, runtimeOnly
    }
}
```

producer/build.gradle.kts

```
val instrumentedJars by configurations.creating {
    isCanBeConsumed = true
    isCanBeResolved = false
    // If you want this configuration to share the same dependencies,
otherwise omit this line
    extendsFrom(configurations["implementation"],
configurations["runtimeOnly"])
}
```

This configuration is *consumable*, which means it’s an "exchange" meant for consumers. We’re now going to add artifacts to this configuration, that consumers would get when they consume it:

Example 424. Attaching an artifact to an outgoing configuration

producer/build.gradle

```
artifacts {
    instrumentedJars(instrumentedJar)
}
```

producer/build.gradle.kts

```
artifacts {
    add("instrumentedJars", instrumentedJar)
}
```

Here the "artifact" we're attaching is a *task* that actually generates a Jar. Doing so, Gradle can automatically track dependencies of this task and build them as needed. This is possible because the `Jar` task extends `AbstractArchiveTask`. If it's not the case, you will need to explicitly declare how the artifact is generated.

Example 425. Explicitly declaring the task dependency of an artifact

producer/build.gradle

```
artifacts {
    instrumentedJars(someTask.outputFile) {
        builtBy(someTask)
    }
}
```

producer/build.gradle.kts

```
artifacts {
    add("instrumentedJars", someTask.outputFile) {
        builtBy(someTask)
    }
}
```

Now the *consumer* needs to depend on this configuration in order to get the right artifact:

Example 426. An explicit configuration dependency

consumer/build.gradle

```
dependencies {
    instrumentedClasspath(project(path: ":producer", configuration:
'instrumentedJars'))
}
```

consumer/build.gradle.kts

```
dependencies {
    instrumentedClasspath(project(mapOf(
        "path" to ":producer",
        "configuration" to "instrumentedJars")))
}
```

WARNING Declaring a dependency on an explicit target configuration is *not recommended* if you plan to publish the component which has this dependency: this would likely lead to broken metadata. If you need to publish the component on a remote repository, follow the instructions of the variant-aware cross publication documentation.

In this case, we're adding the dependency to the *instrumentedClasspath* configuration, which is a *consumer specific configuration*. In Gradle terminology, this is called a resolvable configuration, which is defined this way:

Example 427. Declaring a resolvable configuration on the consumer

consumer/build.gradle

```
configurations {
    instrumentedClasspath {
        canBeConsumed = false
        canBeResolved = true
    }
}
```

consumer/build.gradle.kts

```
val instrumentedClasspath by configurations.creating {
    isCanBeConsumed = false
    isCanBeResolved = true
}
```

Variant-aware sharing of artifacts between projects

In the simple sharing solution, we defined a configuration on the producer side which serves as an exchange of artifacts between the producer and the consumer. However, the consumer has to explicitly tell which configuration it depends on, which is something we want to avoid in *variant aware resolution*. In fact, we also have explained that it is possible for a consumer to express requirements using *attributes* and that the producer should provide the appropriate outgoing variants using attributes too. This allows for smarter selection, because using a single dependency declaration, without any explicit target configuration, the consumer may resolve different things. The typical example is that using a single dependency declaration `project(":myLib")`, we would either choose the `arm64` or `i386` version of `myLib` depending on the architecture.

To do this, we will add attributes to both the consumer and the producer.

WARNING

It is important to understand that once configurations have attributes, they participate in *variant aware resolution*, which means that they are candidates considered whenever *any* notation like `project(":myLib")` is used. In other words, the attributes set on the producer *must be consistent with the other variants produced on the same project*. They must not, in particular, introduce ambiguity for the existing selection.

In practice, it means that the attribute set used on the configuration you create are likely to be dependent on the *ecosystem* in use (Java, C++, ...) because the relevant plugins for those ecosystems often use different attributes.

Let's enhance our previous example which happens to be a Java Library project. Java libraries

expose a couple of variants to their consumers, `apiElements` and `runtimeElements`. Now, we're adding a 3rd one, `instrumentedJars`.

Therefore, we need to understand what our new variant is used for in order to set the proper attributes on it. Let's look at the attributes we find on the `runtimeElements` configuration on the producer:

gradle outgoingVariants --variant runtimeElements

```
Attributes
    - org.gradle.category            = library
    - org.gradle.dependency.bundling = external
    - org.gradle.jvm.version         = 11
    - org.gradle.libraryelements     = jar
    - org.gradle.usage               = java-runtime
```

What it tells us is that the Java Library plugin produces variants with 5 attributes:

- `org.gradle.category` tells us that this variant represents a *library*
- `org.gradle.dependency.bundling` tells us that the dependencies of this variant are found as jars (they are not, for example, repackaged inside the jar)
- `org.gradle.jvm.version` tells us that the minimum Java version this library supports is Java 11
- `org.gradle.libraryelements` tells us this variant contains all elements found in a jar (classes and resources)
- `org.gradle.usage` says that this variant is a Java runtime, therefore suitable for a Java compiler but also at runtime

As a consequence, if we want our instrumented classes to be used in place of this variant when executing tests, we need to attach similar attributes to our variant. In fact, the attribute we care about is `org.gradle.libraryelements` which explains *what the variant contains*, so we can setup the variant this way:

Example 428. Declaring the variant attributes

producer/build.gradle

```
configurations {
    instrumentedJars {
        canBeConsumed = true
        canBeResolved = false
        attributes {
            attribute(Category.CATEGORY_ATTRIBUTE, objects.named(Category,
Category.LIBRARY))
            attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage, Usage
.JAVA_RUNTIME))
            attribute(Bundling.BUNDLING_ATTRIBUTE, objects.named(Bundling,
Bundling.EXTERNAL))
            attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
JavaVersion.current().majorVersion.toInteger())
            attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE, objects
.named(LibraryElements, 'instrumented-jar'))
        }
    }
}
```

producer/build.gradle.kts

```
val instrumentedJars by configurations.creating {
    isCanBeConsumed = true
    isCanBeResolved = false
    attributes {
        attribute(Category.CATEGORY_ATTRIBUTE,
objects.named(Category.LIBRARY))
        attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage.JAVA_RUNTIME))
        attribute(Bundling.BUNDLING_ATTRIBUTE,
objects.named(Bundling.EXTERNAL))
        attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
JavaVersion.current().majorVersion.toInt())
        attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
objects.named("instrumented-jar"))
    }
}
```

NOTE	Choosing the right attributes to set is the hardest thing in this process, because they carry the semantics of the variant. Therefore, before adding *new attributes*, you should always ask yourself if there isn't an attribute which carries the semantics you need. If there isn't, then you may add a new attribute. When adding new attributes, you must also be careful because it's possible that it creates ambiguity during selection. Often adding an attribute means adding it to *all* existing variants.

What we have done here is that we have added a *new* variant, which can be used *at runtime*, but contains instrumented classes instead of the normal classes. However, it now means that for runtime, the consumer has to choose between two variants:

- `runtimeElements`, the regular variant offered by the `java-library` plugin
- `instrumentedJars`, the variant we have created

In particular, say we want the instrumented classes on the test runtime classpath. We can now, on the consumer, declare our dependency as a regular project dependency:

Example 429. Declaring the project dependency

consumer/build.gradle

```
dependencies {
    testImplementation 'junit:junit:4.13'
    testImplementation project(':producer')
}
```

consumer/build.gradle.kts

```
dependencies {
    testImplementation("junit:junit:4.13")
    testImplementation(project(":producer"))
}
```

If we stop here, Gradle will still select the `runtimeElements` variant in place of our `instrumentedJars` variant. This is because the `testRuntimeClasspath` configuration asks for a configuration which `libraryelements` attribute is `jar`, and our new `instrumented-jars` value is *not compatible*.

So we need to change the requested attributes so that we now look for instrumented jars:

Example 430. Changing the consumer attributes

consumer/build.gradle

```
configurations {
    testRuntimeClasspath {
        attributes {
            attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE, objects
.named(LibraryElements, 'instrumented-jar'))
        }
    }
}
```

consumer/build.gradle.kts

```
configurations {
    testRuntimeClasspath {
        attributes {
            attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
objects.named(LibraryElements::class.java, "instrumented-jar"))
        }
    }
}
```

We can look at another report *on the consumer side* to view exactly what attributes of each dependency will be requested:

gradle resolvableConfigurations --configuration testRuntimeClasspath

```
Attributes
    - org.gradle.category            = library
    - org.gradle.dependency.bundling = external
    - org.gradle.jvm.version         = 11
    - org.gradle.libraryelements     = instrumented-jar
    - org.gradle.usage               = java-runtime
```

The `resolvableConfigurations` report is the complement of the `outgoingVariants` report. By running both of these reports on the consumer and producer sides of a relationship, respectively, you can see exactly what attributes are involved in matching during dependency resolution and better predict the outcome when configurations are resolved.

Now, we're saying that whenever we're going to resolve the test runtime classpath, what we are looking for is *instrumented classes*. There is a problem though: in our dependencies list, we have JUnit, which, obviously, is *not* instrumented. So if we stop here, Gradle is going to fail, explaining that there's no variant of JUnit which provide instrumented classes. This is because we didn't

explain that it's fine to use the regular jar, if no instrumented version is available. To do this, we need to write a *compatibility rule*:

Example 431. A compatibility rule

consumer/build.gradle

```
abstract class InstrumentedJarsRule implements AttributeCompatibilityRule
<LibraryElements> {

    @Override
    void execute(CompatibilityCheckDetails<LibraryElements> details) {
        if (details.consumerValue.name == 'instrumented-jar' && details
.producerValue.name == 'jar') {
            details.compatible()
        }
    }
}
```

consumer/build.gradle.kts

```
abstract class InstrumentedJarsRule:
AttributeCompatibilityRule<LibraryElements> {

    override fun execute(details: CompatibilityCheckDetails<LibraryElements>)
= details.run {
        if (consumerValue?.name == "instrumented-jar" && producerValue?.name
== "jar") {
            compatible()
        }
    }
}
```

which we need to declare on the attributes schema:

Example 432. Making use of the compatibility rule

consumer/build.gradle

```
dependencies {
    attributesSchema {
        attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE) {
            compatibilityRules.add(InstrumentedJarsRule)
        }
    }
}
```

consumer/build.gradle.kts

```
dependencies {
    attributesSchema {
        attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE) {
            compatibilityRules.add(InstrumentedJarsRule::class.java)
        }
    }
}
```

And that's it! Now we have:

- added a variant which provides instrumented jars
- explained that this variant is a substitute for the runtime
- explained that the consumer needs this variant *only for test runtime*

Gradle therefore offers a powerful mechanism to select the right variants based on preferences and compatibility. More details can be found in the variant aware plugins section of the documentation.

WARNING

By adding a value to an existing attribute like we have done, or by defining new attributes, we are extending the model. This means that *all consumers* have to know about this extended model. For local consumers, this is usually not a problem because all projects understand and share the same schema, but if you had to publish this new variant to an external repository, it means that external consumers would have to add the same rules to their builds for them to pass. This is in general not a problem for *ecosystem plugins* (e.g: the Kotlin plugin) where consumption is in any case not possible without applying the plugin, but it is a problem if you add custom values or attributes.

So, **avoid publishing custom variants** if they are for internal use only.

Targeting different platforms

It is common for a library to target different platforms. In the Java ecosystem, we often see different artifacts for the same library, distinguished by a different *classifier*. A typical example is Guava, which is published as this:

- `guava-jre` for JDK 8 and above
- `guava-android` for JDK 7

The problem with this approach is that there's no semantics associated with the classifier. The dependency resolution engine, in particular, cannot determine automatically which version to use based on the consumer requirements. For example, it would be better to express that you have a dependency on Guava, and let the engine choose between `jre` and `android` based on what is compatible.

Gradle provides an improved model for this, which doesn't have the weakness of classifiers: attributes.

In particular, in the Java ecosystem, Gradle provides a built-in attribute that library authors can use to express compatibility with the Java ecosystem: `org.gradle.jvm.version`. This attribute expresses the *minimal version that a consumer must have in order to work properly*.

When you apply the `java` or `java-library` plugins, Gradle will automatically associate this attribute to the outgoing variants. This means that all libraries published with Gradle automatically tell which target platform they use.

By default, the `org.gradle.jvm.version` is set to the value of the `release` property (or as fallback to the `targetCompatibility` value) of the main compilation task of the source set.

While this attribute is automatically set, Gradle *will not*, by default, let you build a project for different JVMs. If you need to do this, then you will need to create additional variants following the instructions on variant-aware matching.

NOTE	Future versions of Gradle will provide ways to automatically build for different Java platforms.

Transforming dependency artifacts on resolution

As described in different kinds of configurations, there may be different variants for the same dependency. For example, an external Maven dependency has a variant which should be used when compiling against the dependency (`java-api`), and a variant for running an application which uses the dependency (`java-runtime`). A project dependency has even more variants, for example the classes of the project which are used for compilation are available as classes directories (`org.gradle.usage=java-api, org.gradle.libraryelements=classes`) or as JARs (`org.gradle.usage=java-api, org.gradle.libraryelements=jar`).

The variants of a dependency may differ in its transitive dependencies or in the artifact itself. For example, the `java-api` and `java-runtime` variants of a Maven dependency only differ in the transitive dependencies and both use the same artifact — the JAR file. For a project dependency, the

`java-api,classes` and the `java-api,jars` variants have the same transitive dependencies and different artifacts — the classes directories and the JAR files respectively.

Gradle identifies a variant of a dependency uniquely by its set of attributes. The `java-api` variant of a dependency is the variant identified by the `org.gradle.usage` attribute with value `java-api`.

When Gradle resolves a configuration, the attributes on the resolved configuration determine the *requested attributes*. For all dependencies in the configuration, the variant with the requested attributes is selected when resolving the configuration. For example, when the configuration requests `org.gradle.usage=java-api, org.gradle.libraryelements=classes` on a project dependency, then the classes directory is selected as the artifact.

When the dependency does not have a variant with the requested attributes, resolving the configuration fails. Sometimes it is possible to transform the artifact of the dependency into the requested variant without changing the transitive dependencies. For example, unzipping a JAR transforms the artifact of the `java-api,jars` variant into the `java-api,classes` variant. Such a transformation is called *Artifact Transform*. Gradle allows registering artifact transforms, and when the dependency does not have the requested variant, then Gradle will try to find a chain of artifact transforms for creating the variant.

Artifact transform selection and execution

As described above, when Gradle resolves a configuration and a dependency in the configuration does not have a variant with the requested attributes, Gradle tries to find a chain of artifact transforms to create the variant. The process of finding a matching chain of artifact transforms is called *artifact transform selection*. Each registered transform converts from a set of attributes to a set of attributes. For example, the unzip transform can convert from `org.gradle.usage=java-api, org.gradle.libraryelements=jars` to `org.gradle.usage=java-api, org.gradle.libraryelements=classes`.

In order to find a chain, Gradle starts with the requested attributes and then considers all transforms which modify some of the requested attributes as possible paths leading there. Going backwards, Gradle tries to obtain a path to some existing variant using transforms.

For example, consider a `minified` attribute with two values: `true` and `false`. The minified attribute represents a variant of a dependency with unnecessary class files removed. There is an artifact transform registered, which can transform `minified` from `false` to `true`. When `minified=true` is requested for a dependency, and there are only variants with `minified=false`, then Gradle selects the registered minify transform. The minify transform is able to transform the artifact of the dependency with `minified=false` to the artifact with `minified=true`.

Of all the found transform chains, Gradle tries to select the best one:

- If there is only one transform chain, it is selected.
- If there are two transform chains, and one is a suffix of the other one, it is selected.
- If there is a shortest transform chain, then it is selected.
- In all other cases, the selection fails and an error is reported.

NOTE	Gradle does not try to select artifact transforms when there is already a variant of the dependency matching the requested attributes.
NOTE	The `artifactType` attribute is special, since it is only present on resolved artifacts and not on dependencies. As a consequence, any transform which is only mutating `artifactType` will never be selected when resolving a configuration with only the `artifactType` as requested attribute. It will only be considered when using an ArtifactView.

After selecting the required artifact transforms, Gradle resolves the variants of the dependencies which are necessary for the initial transform in the chain. As soon as Gradle finishes resolving the artifacts for the variant, either by downloading an external dependency or executing a task producing the artifact, Gradle starts transforming the artifacts of the variant with the selected chain of artifact transforms. Gradle executes the transform chains in parallel when possible.

Picking up the minify example above, consider a configuration with two dependencies, the external `guava` dependency and a project dependency on the `producer` project. The configuration has the attributes `org.gradle.usage=java-runtime,org.gradle.libraryelements=jar,minified=true`. The external `guava` dependency has two variants:

- `org.gradle.usage=java-runtime,org.gradle.libraryelements=jar,minified=false` and
- `org.gradle.usage=java-api,org.gradle.libraryelements=jar,minified=false`.

Using the minify transform, Gradle can convert the variant `org.gradle.usage=java-runtime,org.gradle.libraryelements=jar,minified=false` of `guava` to `org.gradle.usage=java-runtime,org.gradle.libraryelements=jar,minified=true`, which are the requested attributes. The project dependency also has variants:

- `org.gradle.usage=java-runtime,org.gradle.libraryelements=jar,minified=false`,
- `org.gradle.usage=java-runtime,org.gradle.libraryelements=classes,minified=false`,
- `org.gradle.usage=java-api,org.gradle.libraryelements=jar,minified=false`,
- `org.gradle.usage=java-api,org.gradle.libraryelements=classes,minified=false`
- and a few more.

Again, using the minify transform, Gradle can convert the variant `org.gradle.usage=java-runtime,org.gradle.libraryelements=jar,minified=false` of the project `producer` to `org.gradle.usage=java-runtime,org.gradle.libraryelements=jar,minified=true`, which are the requested attributes.

When the configuration is resolved, Gradle needs to download the `guava` JAR and minify it. Gradle also needs to execute the `producer:jar` task to generate the JAR artifact of the project and then minify it. The downloading and the minification of the `guava.jar` happens in parallel to the execution of the `producer:jar` task and the minification of the resulting JAR.

Here is how to setup the `minified` attribute so that the above works. You need to register the new attribute in the schema, add it to all JAR artifacts and request it on all resolvable configurations.

Example 433. Artifact transform attribute setup

build.gradle

```
def artifactType = Attribute.of('artifactType', String)
def minified = Attribute.of('minified', Boolean)
dependencies {
    attributesSchema {
        attribute(minified)                     ①
    }
    artifactTypes.getByName("jar") {
        attributes.attribute(minified, false)    ②
    }
}

configurations.all {
    afterEvaluate {
        if (canBeResolved) {
            attributes.attribute(minified, true) ③
        }
    }
}

dependencies {
    registerTransform(Minify) {
        from.attribute(minified, false).attribute(artifactType, "jar")
        to.attribute(minified, true).attribute(artifactType, "jar")
    }
}

dependencies {                                  ④
    implementation('com.google.guava:guava:27.1-jre')
    implementation(project(':producer'))
}
```

build.gradle.kts

```
val artifactType = Attribute.of("artifactType", String::class.java)
val minified = Attribute.of("minified", Boolean::class.javaObjectType)
dependencies {
    attributesSchema {
        attribute(minified)                     ①
    }
    artifactTypes.getByName("jar") {
        attributes.attribute(minified, false)    ②
    }
}

configurations.all {
    afterEvaluate {
        if (isCanBeResolved) {
            attributes.attribute(minified, true) ③
        }
    }
}

dependencies {
    registerTransform(Minify::class) {
        from.attribute(minified, false).attribute(artifactType, "jar")
        to.attribute(minified, true).attribute(artifactType, "jar")
    }
}

dependencies {                                  ④
    implementation("com.google.guava:guava:27.1-jre")
    implementation(project(":producer"))
}
```

① Add the attribute to the schema

② All JAR files are not minified

③ Request `minified=true` on all resolvable configurations

④ Add the dependencies which will be transformed

You can now see what happens when we run the `resolveRuntimeClasspath` task which resolves the `runtimeClasspath` configuration. Observe that Gradle transforms the project dependency before the `resolveRuntimeClasspath` task starts. Gradle transforms the binary dependencies when it executes the `resolveRuntimeClasspath` task.

Output when resolving the runtimeClasspath configuration

```
> gradle resolveRuntimeClasspath

> Task :producer:compileJava
> Task :producer:processResources NO-SOURCE
> Task :producer:classes
> Task :producer:jar

> Transform producer.jar (project :producer) with Minify
Nothing to minify - using producer.jar unchanged

> Task :resolveRuntimeClasspath
Minifying guava-27.1-jre.jar
Nothing to minify - using listenablefuture-9999.0-empty-to-avoid-conflict-with-
guava.jar unchanged
Nothing to minify - using jsr305-3.0.2.jar unchanged
Nothing to minify - using checker-qual-2.5.2.jar unchanged
Nothing to minify - using error_prone_annotations-2.2.0.jar unchanged
Nothing to minify - using j2objc-annotations-1.1.jar unchanged
Nothing to minify - using animal-sniffer-annotations-1.17.jar unchanged
Nothing to minify - using failureaccess-1.0.1.jar unchanged

BUILD SUCCESSFUL in 0s
3 actionable tasks: 3 executed
```

Implementing artifact transforms

Similar to task types, an artifact transform consists of an action and some parameters. The major difference to custom task types is that the action and the parameters are implemented as two separate classes.

The implementation of the artifact transform action is a class implementing TransformAction. You need to implement the `transform()` method on the action, which converts an input artifact into zero, one or multiple of output artifacts. Most artifact transforms will be one-to-one, so the transform method will transform the input artifact to exactly one output artifact.

The implementation of the artifact transform action needs to register each output artifact by calling TransformOutputs.dir() or TransformOutputs.file().

You can only supply two types of paths to the `dir` or `file` methods:

- An absolute path to the input artifact or in the input artifact (for an input directory).
- A relative path.

Gradle uses the absolute path as the location of the output artifact. For example, if the input artifact is an exploded WAR, then the transform action can call `TransformOutputs.file()` for all jar files in the `WEB-INF/lib` directory. The output of the transform would then be the library JARs of the web application.

For a relative path, the `dir()` or `file()` method returns a workspace to the transform action. The implementation of the transform action needs to create the transformed artifact at the location of the provided workspace.

The output artifacts replace the input artifact in the transformed variant in the order they were registered. For example, if the configuration consists of the artifacts `lib1.jar`, `lib2.jar`, `lib3.jar`, and the transform action registers a minified output artifact `<artifact-name>-min.jar` for the input artifact, then the transformed configuration consists of the artifacts `lib1-min.jar`, `lib2-min.jar` and `lib3-min.jar`.

Here is the implementation of an `Unzip` transform which transforms a JAR file into a classes directory by unzipping it. The `Unzip` transform does not require any parameters. Note how the implementation uses `@InputArtifact` to inject the artifact to transform into the action. It requests a directory for the unzipped classes by using `TransformOutputs.dir()` and then unzips the JAR file into this directory.

Example 434. Artifact transform without parameters

build.gradle

```
abstract class Unzip implements TransformAction<TransformParameters.None> {
①
    @InputArtifact
②
    abstract Provider<FileSystemLocation> getInputArtifact()

    @Override
    void transform(TransformOutputs outputs) {
        def input = inputArtifact.get().asFile
        def unzipDir = outputs.dir(input.name)
③
        unzipTo(input, unzipDir)
④
    }

    private static void unzipTo(File zipFile, File unzipDir) {
        // implementation...
    }
}
```

build.gradle.kts

```
abstract class Unzip : TransformAction<TransformParameters.None> {
①
    @get:InputArtifact
②
    abstract val inputArtifact: Provider<FileSystemLocation>

    override
    fun transform(outputs: TransformOutputs) {
        val input = inputArtifact.get().asFile
        val unzipDir = outputs.dir(input.name)
③
        unzipTo(input, unzipDir)
④
    }

    private fun unzipTo(zipFile: File, unzipDir: File) {
        // implementation...
    }
}
```

① Use `TransformParameters.None` if the transform does not use parameters

② Inject the input artifact

③ Request an output location for the unzipped files

④ Do the actual work of the transform

An artifact transform may require parameters, like a `String` determining some filter, or some file collection which is used for supporting the transformation of the input artifact. In order to pass those parameters to the transform action, you need to define a new type with the desired parameters. The type needs to implement the marker interface TransformParameters. The parameters must be represented using managed properties and the parameters type must be a managed type. You can use an interface or abstract class declaring the getters and Gradle will generate the implementation. All getters need to have proper input annotations, see the table in the section on incremental build.

You can find out more about implementing artifact transform parameters in Developing Custom Gradle Types.

Here is the implementation of a `Minify` transform that makes JARs smaller by only keeping certain classes in them. The `Minify` transform requires the classes to keep as parameters. Observe how you can obtain the parameters by `TransformAction.getParameters()` in the `transform()` method. The implementation of the `transform()` method requests a location for the minified JAR by using `TransformOutputs.file()` and then creates the minified JAR at this location.

Example 435. Minify transform implementation

build.gradle

```
abstract class Minify implements TransformAction<Parameters> { ①
    interface Parameters extends TransformParameters {         ②
        @Input
        Map<String, Set<String>> getKeepClassesByArtifact()
        void setKeepClassesByArtifact(Map<String, Set<String>> keepClasses)
    }

    @PathSensitive(PathSensitivity.NAME_ONLY)
    @InputArtifact
    abstract Provider<FileSystemLocation> getInputArtifact()

    @Override
    void transform(TransformOutputs outputs) {
        def fileName = inputArtifact.get().asFile.name
        for (entry in parameters.keepClassesByArtifact) {      ③
            if (fileName.startsWith(entry.key)) {
                def nameWithoutExtension = fileName.substring(0, fileName
.length() - 4)
                minify(inputArtifact.get().asFile, entry.value, outputs.file
("${nameWithoutExtension}-min.jar"))
                return
            }
        }
        println "Nothing to minify - using ${fileName} unchanged"
        outputs.file(inputArtifact)                            ④
    }

    private void minify(File artifact, Set<String> keepClasses, File jarFile)
{
        println "Minifying ${artifact.name}"
        // Implementation ...
    }
}
```

build.gradle.kts

```
abstract class Minify : TransformAction<Minify.Parameters> {   ①
    interface Parameters : TransformParameters {               ②
        @get:Input
        var keepClassesByArtifact: Map<String, Set<String>>

    }

    @get:PathSensitive(PathSensitivity.NAME_ONLY)
    @get:InputArtifact
    abstract val inputArtifact: Provider<FileSystemLocation>

    override
    fun transform(outputs: TransformOutputs) {
        val fileName = inputArtifact.get().asFile.name
        for (entry in parameters.keepClassesByArtifact) {      ③
            if (fileName.startsWith(entry.key)) {
                val nameWithoutExtension = fileName.substring(0,
fileName.length - 4)
                minify(inputArtifact.get().asFile, entry.value,
outputs.file("${nameWithoutExtension}-min.jar"))
                return
            }
        }
        println("Nothing to minify - using ${fileName} unchanged")
        outputs.file(inputArtifact)                            ④
    }

    private fun minify(artifact: File, keepClasses: Set<String>, jarFile:
File) {
        println("Minifying ${artifact.name}")
        // Implementation ...
    }
}
```

① Declare the parameter type

② Interface for the transform parameters

③ Use the parameters

④ Use the unchanged input artifact when no minification is required

Remember that the input artifact is a dependency, which may have its own dependencies. If your artifact transform needs access to those transitive dependencies, it can declare an abstract getter returning a `FileCollection` and annotate it with @InputArtifactDependencies. When your transform runs, Gradle will inject the transitive dependencies into that `FileCollection` property by implementing the getter. Note that using input artifact dependencies in a transform has

performance implications, only inject them when you really need them.

Moreover, artifact transforms can make use of the build cache for their outputs. To enable the build cache for an artifact transform, add the `@CacheableTransform` annotation on the action class. For cacheable transforms, you must annotate its @InputArtifact property — and any property marked with @InputArtifactDependencies — with normalization annotations such as @PathSensitive.

The following example shows a more complicated transform. It moves some selected classes of a JAR to a different package, rewriting the byte code of the moved classes and all classes using the moved classes (class relocation). In order to determine the classes to relocate, it looks at the packages of the input artifact and the dependencies of the input artifact. It also does not relocate packages contained in JAR files in an external classpath.

Example 436. Artifact transform for class relocation

build.gradle

```
@CacheableTransform
①
abstract class ClassRelocator implements TransformAction<Parameters> {
    interface Parameters extends TransformParameters {
②
        @CompileClasspath
③
        ConfigurableFileCollection getExternalClasspath()
        @Input
        Property<String> getExcludedPackage()
    }

    @Classpath
④
    @InputArtifact
    abstract Provider<FileSystemLocation> getPrimaryInput()

    @CompileClasspath
    @InputArtifactDependencies
⑤
    abstract FileCollection getDependencies()

    @Override
    void transform(TransformOutputs outputs) {
        def primaryInputFile = primaryInput.get().asFile
        if (parameters.externalClasspath.contains(primaryInput)) {
⑥
            outputs.file(primaryInput)
        } else {
            def baseName = primaryInputFile.name.substring(0,
primaryInputFile.name.length - 4)
            relocateJar(outputs.file("$baseName-relocated.jar"))
        }
    }

    private relocateJar(File output) {
        // implementation...
        def relocatedPackages = (dependencies.collectMany { readPackages(it)
} + readPackages(primaryInput.get().asFile)) as Set
        def nonRelocatedPackages = parameters.externalClasspath.collectMany {
readPackages(it) }
        def relocations = (relocatedPackages - nonRelocatedPackages).collect
{ packageName ->
            def toPackage = "relocated.$packageName"
            println("$packageName -> $toPackage")
            new Relocation(packageName, toPackage)
```

```
        }
        new JarRelocator(primaryInput.get().asFile, output, relocations).run
()
    }
}
```

build.gradle.kts

```
@CacheableTransform
①
abstract class ClassRelocator : TransformAction<ClassRelocator.Parameters> {
    interface Parameters : TransformParameters {
②
        @get:CompileClasspath
③
        val externalClasspath: ConfigurableFileCollection
        @get:Input
        val excludedPackage: Property<String>
    }

    @get:Classpath
④
    @get:InputArtifact
    abstract val primaryInput: Provider<FileSystemLocation>

    @get:CompileClasspath
    @get:InputArtifactDependencies
⑤
    abstract val dependencies: FileCollection

    override
    fun transform(outputs: TransformOutputs) {
        val primaryInputFile = primaryInput.get().asFile
        if (parameters.externalClasspath.contains(primaryInputFile)) {
⑥
            outputs.file(primaryInput)
        } else {
            val baseName = primaryInputFile.name.substring(0,
primaryInputFile.name.length - 4)
            relocateJar(outputs.file("$baseName-relocated.jar"))
        }
    }

    private fun relocateJar(output: File) {
        // implementation...
        val relocatedPackages = (dependencies.flatMap { it.readPackages() } +
primaryInput.get().asFile.readPackages()).toSet()
        val nonRelocatedPackages = parameters.externalClasspath.flatMap {
it.readPackages() }
        val relocations = (relocatedPackages - nonRelocatedPackages).map {
packageName ->
            val toPackage = "relocated.$packageName"
            println("$packageName -> $toPackage")
            Relocation(packageName, toPackage)
        }
        JarRelocator(primaryInput.get().asFile, output, relocations).run()
```

```
    }
}
```

① Declare the transform cacheable

② Interface for the transform parameters

③ Declare input type for each parameter

④ Declare a normalization for the input artifact

⑤ Inject the input artifact dependencies

⑥ Use the parameters

Registering artifact transforms

You need to register the artifact transform actions, providing parameters if necessary, so that they can be selected when resolving dependencies.

In order to register an artifact transform, you must use registerTransform() within the `dependencies {}` block.

There are a few points to consider when using `registerTransform()`:

- The `from` and `to` attributes are required.
- The transform action itself can have configuration options. You can configure them with the `parameters {}` block.
- You must register the transform on the project that has the configuration that will be resolved.
- You can supply any type implementing TransformAction to the `registerTransform()` method.

For example, imagine you want to unpack some dependencies and put the unpacked directories and files on the classpath. You can do so by registering an artifact transform action of type `Unzip`, as shown here:

Example 437. Artifact transform registration without parameters

build.gradle

```gradle
def artifactType = Attribute.of('artifactType', String)

dependencies {
    registerTransform(Unzip) {
        from.attribute(artifactType, 'jar')
        to.attribute(artifactType, 'java-classes-directory')
    }
}
```

build.gradle.kts

```gradle.kts
val artifactType = Attribute.of("artifactType", String::class.java)

dependencies {
    registerTransform(Unzip::class) {
        from.attribute(artifactType, "jar")
        to.attribute(artifactType, "java-classes-directory")
    }
}
```

Another example is that you want to minify JARs by only keeping some `class` files from them. Note the use of the `parameters {}` block to provide the classes to keep in the minified JARs to the `Minify` transform.

Example 438. Artifact transform registration with parameters

build.gradle

```
def artifactType = Attribute.of('artifactType', String)
def minified = Attribute.of('minified', Boolean)
def keepPatterns = [
    "guava": [
        "com.google.common.base.Optional",
        "com.google.common.base.AbstractIterator"
    ] as Set
]


dependencies {
    registerTransform(Minify) {
        from.attribute(minified, false).attribute(artifactType, "jar")
        to.attribute(minified, true).attribute(artifactType, "jar")

        parameters {
            keepClassesByArtifact = keepPatterns
        }
    }
}
```

build.gradle.kts

```
val artifactType = Attribute.of("artifactType", String::class.java)
val minified = Attribute.of("minified", Boolean::class.javaObjectType)
val keepPatterns = mapOf(
    "guava" to setOf(
        "com.google.common.base.Optional",
        "com.google.common.base.AbstractIterator"
    )
)


dependencies {
    registerTransform(Minify::class) {
        from.attribute(minified, false).attribute(artifactType, "jar")
        to.attribute(minified, true).attribute(artifactType, "jar")

        parameters {
            keepClassesByArtifact = keepPatterns
        }
    }
}
```

Implementing incremental artifact transforms

Similar to incremental tasks, artifact transforms can avoid work by only processing changed files from the last execution. This is done by using the InputChanges interface. For artifact transforms, only the input artifact is an incremental input, and therefore the transform can only query for changes there. In order to use InputChanges in the transform action, inject it into the action. For more information on how to use InputChanges, see the corresponding documentation for incremental tasks.

Here is an example of an incremental transform that counts the lines of code in Java source files:

Example 439. Artifact transform for lines of code counting

build.gradle

```
abstract class CountLoc implements TransformAction<TransformParameters.None>
{

    @Inject                                                                 ①
    abstract InputChanges getInputChanges()

    @PathSensitive(PathSensitivity.RELATIVE)
    @InputArtifact
    abstract Provider<FileSystemLocation> getInput()

    @Override
    void transform(TransformOutputs outputs) {
        def outputDir = outputs.dir("${input.get().asFile.name}.loc")
        println("Running transform on ${input.get().asFile.name},
incremental: ${inputChanges.incremental}")
        inputChanges.getFileChanges(input).forEach { change ->          ②
            def changedFile = change.file
            if (change.fileType != FileType.FILE) {
                return
            }
            def outputLocation = new File(outputDir, "${change.
normalizedPath}.loc")
            switch (change.changeType) {
                case ADDED:
                case MODIFIED:
                    println("Processing file ${changedFile.name}")
                    outputLocation.parentFile.mkdirs()

                    outputLocation.text = changedFile.readLines().size()

                case REMOVED:
                    println("Removing leftover output file ${outputLocation
.name}")
                    outputLocation.delete()

            }
        }
    }
}
```

build.gradle.kts

```
abstract class CountLoc : TransformAction<TransformParameters.None> {

    @get:Inject                                                          ①
    abstract val inputChanges: InputChanges

    @get:PathSensitive(PathSensitivity.RELATIVE)
    @get:InputArtifact
    abstract val input: Provider<FileSystemLocation>

    override
    fun transform(outputs: TransformOutputs) {
        val outputDir = outputs.dir("${input.get().asFile.name}.loc")
        println("Running transform on ${input.get().asFile.name},
incremental: ${inputChanges.isIncremental}")
        inputChanges.getFileChanges(input).forEach { change ->          ②
            val changedFile = change.file
            if (change.fileType != FileType.FILE) {
                return@forEach
            }
            val outputLocation =
outputDir.resolve("${change.normalizedPath}.loc")
            when (change.changeType) {
                ChangeType.ADDED, ChangeType.MODIFIED -> {

                    println("Processing file ${changedFile.name}")
                    outputLocation.parentFile.mkdirs()


outputLocation.writeText(changedFile.readLines().size.toString())
                }
                ChangeType.REMOVED -> {
                    println("Removing leftover output file
${outputLocation.name}")
                    outputLocation.delete()
                }
            }
        }
    }
}
```

① Inject `InputChanges`

② Query for changes in the input artifact

Working in a Multi-repo Environment

Composing builds

What is a composite build?

A composite build is simply a build that includes other builds. In many ways a composite build is similar to a Gradle multi-project build, except that instead of including single `projects`, complete `builds` are included.

Composite builds allow you to:

- combine builds that are usually developed independently, for instance when trying out a bug fix in a library that your application uses
- decompose a large multi-project build into smaller, more isolated chunks that can be worked in independently or together as needed

A build that is included in a composite build is referred to, naturally enough, as an "included build". Included builds do not share any configuration with the composite build, or the other included builds. Each included build is configured and executed in isolation.

Included builds interact with other builds via *dependency substitution*. If any build in the composite has a dependency that can be satisfied by the included build, then that dependency will be replaced by a project dependency on the included build. *Because of the reliance on dependency substitution, composite builds may force configurations to be resolved earlier, when composing the task execution graph. This can have a negative impact on overall build performance, because these configurations are not resolved in parallel.*

By default, Gradle will attempt to determine the dependencies that can be substituted by an included build. However for more flexibility, it is possible to explicitly declare these substitutions if the default ones determined by Gradle are not correct for the composite. See Declaring substitutions.

As well as consuming outputs via project dependencies, a composite build can directly declare task dependencies on included builds. Included builds are isolated, and are not able to declare task dependencies on the composite build or on other included builds. See Depending on tasks in an included build.

Defining a composite build

The following examples demonstrate the various ways that 2 Gradle builds that are normally developed separately can be combined into a composite build. For these examples, the `my-utils` multi-project build produces 2 different java libraries (`number-utils` and `string-utils`), and the `my-app` build produces an executable using functions from those libraries.

The `my-app` build does not have direct dependencies on `my-utils`. Instead, it declares binary dependencies on the libraries produced by `my-utils`.

Example 440. Dependencies of my-app

my-app/app/build.gradle

```
plugins {
    id 'application'
}

application {
    mainClass = 'org.sample.myapp.Main'
}

dependencies {
    implementation 'org.sample:number-utils:1.0'
    implementation 'org.sample:string-utils:1.0'
}
```

my-app/app/build.gradle.kts

```
plugins {
    id("application")
}

application {
    mainClass.set("org.sample.myapp.Main")
}

dependencies {
    implementation("org.sample:number-utils:1.0")
    implementation("org.sample:string-utils:1.0")
}
```

Defining a composite build via `--include-build`

The `--include-build` command-line argument turns the executed build into a composite, substituting dependencies from the included build into the executed build.

Output of `gradle --include-build ../my-utils run`

```
> gradle --include-build ../my-utils run
link:https://docs.gradle.org/7.6.6-20260912032715+0000/samples/build-
organization/composite-builds/basic/tests/basicCli.out[]
```

Defining a composite build via the settings file

It's possible to make the above arrangement persistent, by using Settings.includeBuild(java.lang.Object) to declare the included build in the `settings.gradle` (or `settings.gradle.kts` in Kotlin) file. The settings file can be used to add subprojects and included builds at the same time. Included builds are added by location. See the examples below for more details.

Defining a separate composite build

One downside of the above approach is that it requires you to modify an existing build, rendering it less useful as a standalone build. One way to avoid this is to define a separate composite build, whose only purpose is to combine otherwise separate builds.

Example 441. Declaring a separate composite

settings.gradle

```
rootProject.name = 'my-composite'

includeBuild 'my-app'
includeBuild 'my-utils'
```

settings.gradle.kts

```
rootProject.name = "my-composite"

includeBuild("my-app")
includeBuild("my-utils")
```

In this scenario, the 'main' build that is executed is the composite, and it doesn't define any useful tasks to execute itself. In order to execute the 'run' task in the 'my-app' build, the composite build must define a delegating task.

Example 442. Depending on task from included build

build.gradle

```
tasks.register('run') {
    dependsOn gradle.includedBuild('my-app').task(':app:run')
}
```

build.gradle.kts

```
tasks.register("run") {
    dependsOn(gradle.includedBuild("my-app").task(":app:run"))
}
```

More details about tasks that depend on included build tasks are below.

Including builds that define Gradle plugins

A special case of included builds are builds that define Gradle plugins. These builds should be included using the `includeBuild` statement inside the `pluginManagement {}` block of the settings file. Using this mechanism, the included build may also contribute a settings plugin that can be applied in the settings file itself.

Example 443. Including a plugin build

settings.gradle

```
pluginManagement {
    includeBuild '../url-verifier-plugin'
}
```

settings.gradle.kts

```
pluginManagement {
    includeBuild("../url-verifier-plugin")
}
```

NOTE	Including plugin builds via the plugin management block is an incubating feature. You may also use the stable `includeBuild` mechanism outside `pluginManagement` to include plugin builds. However, this does not support all use cases and including plugin builds like that will be deprecated once the new mechanism is stable.

Restrictions on included builds

Most builds can be included into a composite, including other composite builds. However there are some restrictions.

Every included build:

- must not have a `rootProject.name` the same as another included build.
- must not have a `rootProject.name` the same as a top-level project of the composite build.
- must not have a `rootProject.name` the same as the composite build `rootProject.name`.

Interacting with a composite build

In general, interacting with a composite build is much the same as a regular multi-project build. Tasks can be executed, tests can be run, and builds can be imported into the IDE.

Executing tasks

Tasks from an included build can be executed from the command-line or from your IDE in the same way as tasks from a regular multi-project build. Executing a task will result in task dependencies being executed, as well as those tasks required to build dependency artifacts from other included builds.

You call a task in an included build using a fully qualified path, for example `:included-build-name:project-name:taskName`. Project and task names can be abbreviated.

```
$ ./gradlew :included-build:subproject-a:compileJava
> Task :included-build:subproject-a:compileJava

$ ./gradlew :i-b:sA:cJ
> Task :included-build:subproject-a:compileJava
```

To exclude a task from the command line, you also need to provide the fully qualified path to the task.

NOTE	Included build tasks are automatically executed in order to generate required dependency artifacts, or the including build can declare a dependency on a task from an included build.

Importing into the IDE

One of the most useful features of composite builds is IDE integration. By applying the idea or eclipse plugin to your build, it is possible to generate a single IDEA or Eclipse project that permits

all builds in the composite to be developed together.

In addition to these Gradle plugins, recent versions of IntelliJ IDEA and Eclipse Buildship support direct import of a composite build.

Importing a composite build permits sources from separate Gradle builds to be easily developed together. For every included build, each sub-project is included as an IDEA Module or Eclipse Project. Source dependencies are configured, providing cross-build navigation and refactoring.

Declaring the dependencies substituted by an included build

By default, Gradle will configure each included build in order to determine the dependencies it can provide. The algorithm for doing this is very simple: Gradle will inspect the group and name for the projects in the included build, and substitute project dependencies for any external dependency matching `${project.group}:${project.name}`.

NOTE By default, substitutions are not registered for the *main* build. To make the (sub)projects of the main build addressable by `${project.group}:${project.name}`, you can tell Gradle to treat the main build like an included build by self-including it: `includeBuild(".")`.

There are cases when the default substitutions determined by Gradle are not sufficient, or they are not correct for a particular composite. For these cases it is possible to explicitly declare the substitutions for an included build. Take for example a single-project build 'anonymous-library', that produces a java utility library but does not declare a value for the group attribute:

Example 444. Build that does not declare group attribute

build.gradle

```
plugins {
    id 'java'
}
```

build.gradle.kts

```
plugins {
    java
}
```

When this build is included in a composite, it will attempt to substitute for the dependency module "undefined:anonymous-library" ("undefined" being the default value for `project.group`, and "anonymous-library" being the root project name). Clearly this isn't going to be very useful in a composite build. To use the unpublished library unmodified in a composite build, the composing build can explicitly declare the substitutions that it provides:

Example 445. Declaring the substitutions for an included build

settings.gradle

```
rootProject.name = 'declared-substitution'

include 'app'

// tag::composite_substitution[]
includeBuild('anonymous-library') {
    dependencySubstitution {
        substitute module('org.sample:number-utils') using project(':')
    }
}
// end::composite_substitution[]
```

settings.gradle.kts

```
rootProject.name = "declared-substitution"

include("app")

// tag::composite_substitution[]
includeBuild("anonymous-library") {
    dependencySubstitution {
        substitute(module("org.sample:number-utils")).using(project(":"))
    }
}
// end::composite_substitution[]
```

With this configuration, the "my-app" composite build will substitute any dependency on `org.sample:number-utils` with a dependency on the root project of "anonymous-library".

Deactivate included build substitutions for a Configuration

If you need to resolve a published version of a module that is also available as part of an included build, you can deactivate the included build substitution rules on the ResolutionStrategy of the Configuration that is resolved. This is necessary, because the rules are globally applied in the build and Gradle does not consider published versions during resolution by default.

Example 446. Deactivate global dependency substitution rules

build.gradle

```
configurations.create('publishedRuntimeClasspath') {
    resolutionStrategy.useGlobalDependencySubstitutionRules = false

    extendsFrom(configurations.runtimeClasspath)
    canBeConsumed = false
    canBeResolved = true
    attributes.attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage, Usage
.JAVA_RUNTIME))
}
```

build.gradle.kts

```
configurations.create("publishedRuntimeClasspath") {
    resolutionStrategy.useGlobalDependencySubstitutionRules.set(false)

    extendsFrom(configurations.runtimeClasspath.get())
    isCanBeConsumed = false
    isCanBeResolved = true
    attributes.attribute(Usage.USAGE_ATTRIBUTE,
objects.named(Usage.JAVA_RUNTIME))
}
```

In this example, we create a separate `publishedRuntimeClasspath` configuration that gets resolve to the published versions of modules that also exist in one of the local builds. This can be used, for example, to compare published and locally built Jar files.

Cases where included build substitutions must be declared

Many builds will function automatically as an included build, without declared substitutions. Here are some common cases where declared substitutions are required:

- When the `archivesBaseName` property is used to set the name of the published artifact.
- When a configuration other than `default` is published.
- When the `MavenPom.addFilter()` is used to publish artifacts that don't match the project name.
- When the `maven-publish` or `ivy-publish` plugins are used for publishing, and the publication coordinates don't match `${project.group}:${project.name}`.

Cases where composite build substitutions won't work

Some builds won't function correctly when included in a composite, even when dependency

substitutions are explicitly declared. This limitation is due to the fact that a project dependency that is substituted will always point to the `default` configuration of the target project. Any time that the artifacts and dependencies specified for the default configuration of a project don't match what is actually published to a repository, then the composite build may exhibit different behaviour.

Here are some cases where the publish module metadata may be different from the project default configuration:

- When a configuration other than `default` is published.
- When the `maven-publish` or `ivy-publish` plugins are used.
- When the `POM` or `ivy.xml` file is tweaked as part of publication.

Builds using these features function incorrectly when included in a composite build. We plan to improve this in the future.

Depending on tasks in an included build

While included builds are isolated from one another and cannot declare direct dependencies, a composite build is able to declare task dependencies on its included builds. The included builds are accessed using Gradle.getIncludedBuilds() or Gradle.includedBuild(java.lang.String), and a task reference is obtained via the IncludedBuild.task(java.lang.String) method.

Using these APIs, it is possible to declare a dependency on a task in a particular included build, or tasks with a certain path in all or some of the included builds.

Example 447. Depending on a single task from an included build

build.gradle

```
tasks.register('run') {
    dependsOn gradle.includedBuild('my-app').task(':app:run')
}
```

build.gradle.kts

```
tasks.register("run") {
    dependsOn(gradle.includedBuild("my-app").task(":app:run"))
}
```

Example 448. Depending on a task with path in all included builds

build.gradle

```
tasks.register('publishDeps') {
    dependsOn gradle.includedBuilds*.task(
':publishMavenPublicationToMavenRepository')
}
```

build.gradle.kts

```
tasks.register("publishDeps") {
    dependsOn(gradle.includedBuilds.map {
it.task(":publishMavenPublicationToMavenRepository") })
}
```

Current limitations and future plans for composite builds

Limitations of the current implementation include:

- No support for included builds that have publications that don't mirror the project default configuration. See Cases where composite builds won't work.
- Software model based native builds are not supported. (Binary dependencies are not yet supported for native builds).
- Multiple composite builds may conflict when run in parallel, if more than one includes the same build. Gradle does not share the project lock of a shared composite build to between Gradle invocation to prevent concurrent execution.

Improvements we have planned for upcoming releases include:

- Making the implicit `buildSrc` project an included build.

Publishing Libraries

Publishing a project as module

The vast majority of software projects build something that aims to be consumed in some way. It could be a library that other software projects use or it could be an application for end users. *Publishing* is the process by which the thing being built is made available to consumers.

In Gradle, that process looks like this:

1. Define what to publish

2. Define where to publish it to
3. Do the publishing

Each of the these steps is dependent on the type of repository to which you want to publish artifacts. The two most common types are Maven-compatible and Ivy-compatible repositories, or Maven and Ivy repositories for short.

As of Gradle 6.0, the Gradle Module Metadata will always be published alongside the Ivy XML or Maven POM metadata file.

Gradle makes it easy to publish to these types of repository by providing some prepackaged infrastructure in the form of the Maven Publish Plugin and the Ivy Publish Plugin. These plugins allow you to configure what to publish and perform the publishing with a minimum of effort.

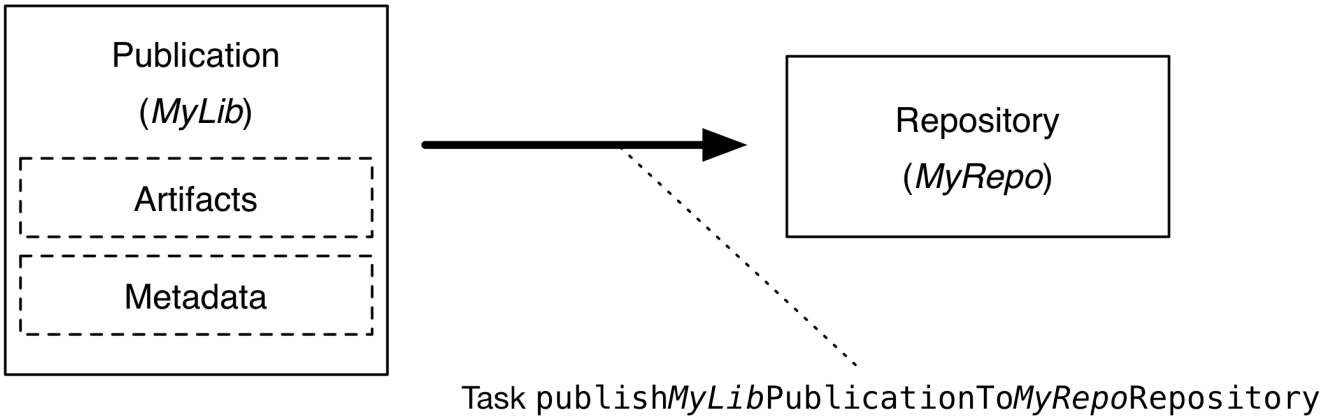


Figure 24. The publishing process

Let's take a look at those steps in more detail:

What to publish

Gradle needs to know what files and information to publish so that consumers can use your project. This is typically a combination of artifacts and metadata that Gradle calls a publication. Exactly what a publication contains depends on the type of repository it's being published to.

For example, a publication destined for a Maven repository includes:

- One or more artifacts — typically built by the project,
- The Gradle Module Metadata file which will describe the variants of the published component,
- The Maven POM file will identify the primary artifact and its dependencies. The primary artifact is typically the project's production JAR and secondary artifacts might consist of "-sources" and "-javadoc" JARs.

In addition, Gradle will publish checksums for all of the above, and signatures when configured to do so. From Gradle 6.0 onwards, this includes `SHA256` and `SHA512` checksums.

Where to publish

Gradle needs to know where to publish artifacts so that consumers can get hold of them. This is done via repositories, which store and make available all sorts of artifact. Gradle also needs to

interact with the repository, which is why you must provide the type of the repository and its location.

How to publish

Gradle automatically generates publishing tasks for all possible combinations of publication and repository, allowing you to publish any artifact to any repository. If you're publishing to a Maven repository, the tasks are of type PublishToMavenRepository, while for Ivy repositories the tasks are of type PublishToIvyRepository.

What follows is a practical example that demonstrates the entire publishing process.

Setting up basic publishing

The first step in publishing, irrespective of your project type, is to apply the appropriate publishing plugin. As mentioned in the introduction, Gradle supports both Maven and Ivy repositories via the following plugins:

- Maven Publish Plugin
- Ivy Publish Plugin

These provide the specific publication and repository classes needed to configure publishing for the corresponding repository type. Since Maven repositories are the most commonly used ones, they will be the basis for this example and for the other samples in the chapter. Don't worry, we will explain how to adjust individual samples for Ivy repositories.

Let's assume we're working with a simple Java library project, so only the following plugins are applied:

Example 449. Applying the necessary plugins

build.gradle

```
plugins {
    id 'java-library'
    id 'maven-publish'
}
```

build.gradle.kts

```
plugins {
    `java-library`
    `maven-publish`
}
```

Once the appropriate plugin has been applied, you can configure the publications and repositories.

For this example, we want to publish the project's production JAR file — the one produced by the `jar` task — to a custom, Maven repository. We do that with the following `publishing {}` block, which is backed by PublishingExtension:

Example 450. Configuring a Java library for publishing

build.gradle

```
group = 'org.example'
version = '1.0'

publishing {
    publications {
        myLibrary(MavenPublication) {
            from components.java
        }
    }

    repositories {
        maven {
            name = 'myRepo'
            url = layout.buildDirectory.dir("repo")
        }
    }
}
```

build.gradle.kts

```
group = "org.example"
version = "1.0"

publishing {
    publications {
        create<MavenPublication>("myLibrary") {
            from(components["java"])
        }
    }

    repositories {
        maven {
            name = "myRepo"
            url = uri(layout.buildDirectory.dir("repo"))
        }
    }
}
```

This defines a publication called "myLibrary" that can be published to a Maven repository by virtue of its type: MavenPublication. This publication consists of just the production JAR artifact and its metadata, which combined are represented by the `java` component of the project.

NOTE	Components are the standard way of defining a publication. They are provided by plugins, usually of the language or platform variety. For example, the Java Plugin defines the `components.java` SoftwareComponent, while the War Plugin defines `components.web`.

The example also defines a file-based Maven repository with the name "myRepo". Such a file-based repository is convenient for a sample, but real-world builds typically work with HTTPS-based repository servers, such as Maven Central or an internal company server.

NOTE	You may define one, and only one, repository without a name. This translates to an implicit name of "Maven" for Maven repositories and "Ivy" for Ivy repositories. All other repository definitions must be given an explicit name.

In combination with the project's `group` and `version`, the publication and repository definitions provide everything that Gradle needs to publish the project's production JAR. Gradle will then create a dedicated `publishMyLibraryPublicationToMyRepoRepository` task that does just that. Its name is based on the template `publishPubNamePublicationToRepoNameRepository`. See the appropriate publishing plugin's documentation for more details on the nature of this task and any other tasks that may be available to you.

You can either execute the individual publishing tasks directly, or you can execute `publish`, which will run all the available publishing tasks. In this example, `publish` will just run `publishMyLibraryPublicationToMavenRepository`.

NOTE	Basic publishing to an Ivy repository is very similar: you simply use the Ivy Publish Plugin, replace `MavenPublication` with IvyPublication, and use `ivy` instead of `maven` in the repository definition. There are differences between the two types of repository, particularly around the extra metadata that each support — for example, Maven repositories require a POM file while Ivy ones have their own metadata format — so see the plugin chapters for comprehensive information on how to configure both publications and repositories for whichever repository type you're working with.

That's everything for the basic use case. However, many projects need more control over what gets published, so we look at several common scenarios in the following sections.

Suppressing validation errors

Gradle performs validation of generated module metadata. In some cases, validation can fail, indicating that you most likely have an error to fix, but you may have done something intentionally. If this is the case, Gradle will indicate the name of the validation error you can disable on the `GenerateModuleMetadata` tasks:

Example 451. Disabling some validation errors

build.gradle

```
tasks.withType(GenerateModuleMetadata).configureEach {
    // The value 'enforced-platform' is provided in the validation
    // error message you got
    suppressedValidationErrors.add('enforced-platform')
}
```

build.gradle.kts

```
tasks.withType<GenerateModuleMetadata> {
    // The value 'enforced-platform' is provided in the validation
    // error message you got
    suppressedValidationErrors.add("enforced-platform")
}
```

Understanding Gradle Module Metadata

Gradle Module Metadata is a format used to serialize the Gradle component model. It is similar to Apache Maven™'s POM file or Apache Ivy™ ivy.xml files. The goal of metadata files is to provide *to consumers* a reasonable model of what is published on a repository.

Gradle Module Metadata is a unique format aimed at improving dependency resolution by making it multi-platform and variant-aware.

In particular, Gradle Module Metadata supports:

- rich version constraints
- dependency constraints
- component capabilities
- variant-aware resolution

Publication of Gradle Module Metadata will enable better dependency management for your consumers:

- early discovery of problems by detecting incompatible modules
- consistent selection of platform-specific dependencies
- native dependency version alignment
- automatically getting dependencies for specific features of your library

Gradle Module Metadata is automatically published when using the Maven Publish plugin or the

Ivy Publish plugin.

The specification for Gradle Module Metadata can be found here.

Mapping with other formats

Gradle Module Metadata is automatically published on Maven or Ivy repositories. However, it doesn't replace the *pom.xml* or *ivy.xml* files: it is published alongside those files. This is done to maximize compatibility with third-party build tools.

Gradle does its best to map Gradle-specific concepts to Maven or Ivy. When a build file uses features that can only be represented in Gradle Module Metadata, Gradle will warn you at publication time. The table below summarizes how some Gradle specific features are mapped to Maven and Ivy:

Table 22. Mapping of Gradle specific concepts to Maven and Ivy

Gradle	Maven	Ivy	Description
dependency constraints	`<dependencyManagement>` dependencies	Not published	Gradle dependency constraints are *transitive*, while Maven's dependency management block *isn't*
rich version constraints	Publishes the *requires* version	Published the *requires* version	
component capabilities	Not published	Not published	Component capabilities are unique to Gradle
Feature variants	Variant artifacts are uploaded, dependencies are published as *optional* dependencies	Variant artifacts are uploaded, dependencies are not published	Feature variants are a good replacement for optional dependencies
Custom component types	Artifacts are uploaded, dependencies are those described by the mapping	Artifacts are uploaded, dependencies are ignored	Custom component types are probably not consumable from Maven or Ivy in any case. They usually exist in the context of a custom ecosystem.

Disabling metadata compatibility publication warnings

If you want to suppress warnings, you can use the following APIs to do so:

- For Maven, see the `suppress*` methods in MavenPublication
- For Ivy, see the `suppress*` methods in IvyPublication

Example 452. Disabling publication warnings

build.gradle

```
publications {
    maven(MavenPublication) {
        from components.java
        suppressPomMetadataWarningsFor('runtimeElements')
    }
}
```

build.gradle.kts

```
publications {
    register<MavenPublication>("maven") {
        from(components["java"])
        suppressPomMetadataWarningsFor("runtimeElements")
    }
}
```

Interactions with other build tools

Because Gradle Module Metadata is not widely spread and because it aims at maximizing compatibility with other tools, Gradle does a couple of things:

- Gradle Module Metadata is systematically published alongside the normal descriptor for a given repository (Maven or Ivy)
- the `pom.xml` or `ivy.xml` file will contain a *marker comment* which tells Gradle that Gradle Module Metadata exists for this module

The goal of the marker is *not* for other tools to parse module metadata: it's for Gradle users only. It explains to Gradle that a *better* module metadata file exists and that it should use it instead. It doesn't mean that consumption from Maven or Ivy would be broken either, only that it works in degraded mode.

NOTE	This must be seen as a *performance optimization*: instead of having to do 2 network requests, one to get Gradle Module Metadata, then one to get the POM/Ivy file in case of a miss, Gradle will first look at the file which is most likely to be present, then only perform a 2nd request if the module was actually published with Gradle Module Metadata.

If you know that the modules you depend on are always published with Gradle Module Metadata, you can optimize the network calls by configuring the metadata sources for a repository:

Example 453. Resolving Gradle Module Metadata only

build.gradle

```
repositories {
    maven {
        url "http://repo.mycompany.com/repo"
        metadataSources {
            gradleMetadata()
        }
    }
}
```

build.gradle.kts

```
repositories {
    maven {
        setUrl("http://repo.mycompany.com/repo")
        metadataSources {
            gradleMetadata()
        }
    }
}
```

Gradle Module Metadata validation

Gradle Module Metadata is validated before being published.

The following rules are enforced:

- Variant names must be unique,
- Each variant must have at least one attribute,
- Two variants cannot have the exact same attributes and capabilities,
- If there are dependencies, at least one, across all variants, must carry version information.

These rules ensure the quality of the metadata produced, and help confirm that consumption will not be problematic.

Gradle Module Metadata reproducibility

The task generating the module metadata files is currently never marked `UP-TO-DATE` by Gradle due to the way it is implemented. However, if neither build inputs nor build scripts changed, the task result is effectively up-to-date: it always produces the same output.

If users desire to have a unique `module` file per build invocation, it is possible to link an identifier in the produced metadata to the build that created it. Users can choose to enable this unique identifier in their `publication`:

Example 454. Configuring the build identifier of a publication

build.gradle

```
publishing {
    publications {
        myLibrary(MavenPublication) {
            from components.java
            withBuildIdentifier()
        }
    }
}
```

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("myLibrary") {
            from(components["java"])
            withBuildIdentifier()
        }
    }
}
```

With the changes above, the generated Gradle Module Metadata file will always be different, forcing downstream tasks to consider it out-of-date.

Disabling Gradle Module Metadata publication

There are situations where you might want to disable publication of Gradle Module Metadata:

- the repository you are uploading to rejects the metadata file (unknown format)
- you are using Maven or Ivy specific concepts which are not properly mapped to Gradle Module Metadata

In this case, disabling the publication of Gradle Module Metadata is done simply by disabling the task which generates the metadata file:

Example 455. Disabling publication of Gradle Module Metadata

build.gradle

```
tasks.withType(GenerateModuleMetadata) {
    enabled = false
}
```

build.gradle.kts

```
tasks.withType<GenerateModuleMetadata> {
    enabled = false
}
```

Signing artifacts

The Signing Plugin can be used to sign all artifacts and metadata files that make up a publication, including Maven POM files and Ivy module descriptors. In order to use it:

1. Apply the Signing Plugin
2. Configure the signatory credentials — follow the link to see how
3. Specify the publications you want signed

Here's an example that configures the plugin to sign the `mavenJava` publication:

Example 456. Signing a publication

build.gradle

```
signing {
    sign publishing.publications.mavenJava
}
```

build.gradle.kts

```
signing {
    sign(publishing.publications["mavenJava"])
}
```

This will create a `Sign` task for each publication you specify and wire all `publishPubNamePublicationToRepoNameRepository` tasks to depend on it. Thus, publishing any publication will automatically create and publish the signatures for its artifacts and metadata, as you can see from this output:

Example: Sign and publish a project

Output of `gradle publish`

```
> gradle publish
> Task :compileJava
> Task :processResources
> Task :classes
> Task :jar
> Task :javadoc
> Task :javadocJar
> Task :sourcesJar
> Task :generateMetadataFileForMavenJavaPublication
> Task :generatePomFileForMavenJavaPublication
> Task :signMavenJavaPublication
> Task :publishMavenJavaPublicationToMavenRepository
> Task :publish

BUILD SUCCESSFUL in 0s
10 actionable tasks: 10 executed
```

Customizing publishing

Modifying and adding variants to existing components for publishing

Gradle's publication model is based on the notion of *components*, which are defined by plugins. For example, the Java Library plugin defines a `java` component which corresponds to a library, but the Java Platform plugin defines another kind of component, named `javaPlatform`, which is effectively a different kind of software component (a *platform*).

Sometimes we want to add *more variants* to or modify *existing variants* of an existing component. For example, if you added a variant of a Java library for a different platform, you may just want to declare this additional variant on the `java` component itself. In general, declaring additional variants is often the best solution to publish *additional artifacts*.

To perform such additions or modifications, the `AdhocComponentWithVariants` interface declares two methods called `addVariantsFromConfiguration` and `withVariantsFromConfiguration` which accept two parameters:

- the outgoing configuration that is used as a variant source
- a customization action which allows you to *filter* which variants are going to be published

To utilise these methods, you must make sure that the `SoftwareComponent` you work with is itself an `AdhocComponentWithVariants`, which is the case for the components created by the Java plugins (Java,

Java Library, Java Platform). Adding a variant is then very simple:

Example 457. Adding a variant to an existing software component

InstrumentedJarsPlugin.groovy

```
        AdhocComponentWithVariants javaComponent =
(AdhocComponentWithVariants) project.components.findByName("java")
        javaComponent.addVariantsFromConfiguration(outgoing) {
            // dependencies for this variant are considered runtime
dependencies
            it.mapToMavenScope("runtime")
            // and also optional dependencies, because we don't want them to
leak
            it.mapToOptional()
        }
```

InstrumentedJarsPlugin.kt

```
        val javaComponent = components.findByName("java") as
AdhocComponentWithVariants
        javaComponent.addVariantsFromConfiguration(outgoing) {
            // dependencies for this variant are considered runtime
dependencies
            mapToMavenScope("runtime")
            // and also optional dependencies, because we don't want them to
leak
            mapToOptional()
        }
```

In other cases, you might want to modify a variant that was added by one of the Java plugins already. For example, if you activate publishing of Javadoc and sources, these become additional variants of the `java` component. If you only want to publish one of them, e.g. only Javadoc but no sources, you can modify the `sources` variant to not being published:

Example 458. Publish a java library with Javadoc but without sources

build.gradle

```
java {
    withJavadocJar()
    withSourcesJar()
}

components.java.withVariantsFromConfiguration(configurations.sourcesElements)
{
    skip()
}

publishing {
    publications {
        mavenJava(MavenPublication) {
            from components.java
        }
    }
}
```

build.gradle.kts

```
java {
    withJavadocJar()
    withSourcesJar()
}

val javaComponent = components["java"] as AdhocComponentWithVariants
javaComponent.withVariantsFromConfiguration(configurations["sourcesElements"]
) {
    skip()
}

publishing {
    publications {
        create<MavenPublication>("mavenJava") {
            from(components["java"])
        }
    }
}
```

Creating and publishing custom components

In the previous example, we have demonstrated how to extend or modify an existing component, like the components provided by the Java plugins. But Gradle also allows you to build a custom component (not a Java Library, not a Java Platform, not something supported natively by Gradle).

To create a custom component, you first need to create an empty *adhoc* component. At the moment, this is only possible via a plugin because you need to get a handle on the SoftwareComponentFactory :

Example 459. Injecting the software component factory

InstrumentedJarsPlugin.groovy

```
    private final SoftwareComponentFactory softwareComponentFactory

    @Inject
    InstrumentedJarsPlugin(SoftwareComponentFactory softwareComponentFactory)
{
        this.softwareComponentFactory = softwareComponentFactory
    }
```

InstrumentedJarsPlugin.kt

```
class InstrumentedJarsPlugin @Inject constructor(
    private val softwareComponentFactory: SoftwareComponentFactory) :
Plugin<Project> {
```

Declaring *what* a custom component publishes is still done via the AdhocComponentWithVariants API. For a custom component, the first step is to create custom outgoing variants, following the instructions in this chapter. At this stage, what you should have is variants which can be used in cross-project dependencies, but that we are now going to publish to external repositories.

Example 460. Creating a custom, adhoc component

InstrumentedJarsPlugin.groovy

```
        // create an adhoc component
        def adhocComponent = softwareComponentFactory.adhoc("
myAdhocComponent")
        // add it to the list of components that this project declares
        project.components.add(adhocComponent)
        // and register a variant for publication
        adhocComponent.addVariantsFromConfiguration(outgoing) {
            it.mapToMavenScope("runtime")
        }
```

InstrumentedJarsPlugin.kt

```
        // create an adhoc component
        val adhocComponent =
softwareComponentFactory.adhoc("myAdhocComponent")
        // add it to the list of components that this project declares
        components.add(adhocComponent)
        // and register a variant for publication
        adhocComponent.addVariantsFromConfiguration(outgoing) {
            mapToMavenScope("runtime")
        }
```

First we use the factory to create a new adhoc component. Then we add a variant through the `addVariantsFromConfiguration` method, which is described in more detail in the previous section.

In simple cases, there's a one-to-one mapping between a `Configuration` and a variant, in which case you can publish all variants issued from a single `Configuration` because they are effectively the same thing. However, there are cases where a `Configuration` is associated with additional configuration publications that we also call *secondary variants*. Such configurations make sense in the cross-project publications use case, but not when publishing externally. This is for example the case when between projects you share a *directory of files*, but there's no way you can publish a *directory* directly on a Maven repository (only packaged things like jars or zips). Look at the ConfigurationVariantDetails class for details about how to skip publication of a particular variant. If `addVariantsFromConfiguration` has already been called for a configuration, further modification of the resulting variants can be performed using `withVariantsFromConfiguration`.

When publishing an adhoc component like this:

- Gradle Module Metadata will *exactly* represent the published variants. In particular, all outgoing variants will inherit dependencies, artifacts and attributes of the published configuration.

- Maven and Ivy metadata files will be generated, but you need to declare how the dependencies are mapped to Maven scopes via the ConfigurationVariantDetails class.

In practice, it means that components created this way can be consumed by Gradle the same way as if they were "local components".

Adding custom artifacts to a publication

WARNING

Instead of thinking in terms of artifacts, you should embrace the variant aware model of Gradle. It is expected that a single module may need multiple artifacts. However this rarely stops there, if the additional artifacts represent an optional feature, they might also have different dependencies and more.

Gradle, via *Gradle Module Metadata*, supports the publication of *additional variants* which make those artifacts known to the dependency resolution engine. Please refer to the variant-aware sharing section of the documentation to see how to declare such variants and check out how to publish custom components.

If you attach extra artifacts to a publication directly, they are published "out of context". That means, they are not referenced in the metadata at all and can then only be addressed directly through a classifier on a dependency. In contrast to Gradle Module Metadata, Maven pom metadata will not contain information on additional artifacts regardless of whether they are added through a variant or directly, as variants cannot be represented in the pom format.

The following section describes how you publish artifacts directly if you are sure that metadata, for example Gradle or POM metadata, is irrelevant for your use case. For example, if your project doesn't need to be consumed by other projects and the only thing required as result of the publishing are the artifacts themselves.

In general, there are two options:

- Create a publication only with artifacts
- Add artifacts to a publication based on a component with metadata (not recommended, instead adjust a component or use a adhoc component publication which will both also produce metadata fitting your artifacts)

To create a publication based on artifacts, start by defining a custom artifact and attaching it to a Gradle configuration of your choice. The following sample defines an RPM artifact that is produced by an `rpm` task (not shown) and attaches that artifact to the `archives` configuration:

Example 461. Defining a custom artifact for a configuration

build.gradle

```
def rpmFile = layout.buildDirectory.file('rpms/my-package.rpm')
def rpmArtifact = artifacts.add('archives', rpmFile.get().asFile) {
    type 'rpm'
    builtBy 'rpm'
}
```

build.gradle.kts

```
val rpmFile = layout.buildDirectory.file("rpms/my-package.rpm")
val rpmArtifact = artifacts.add("archives", rpmFile.get().asFile) {
    type = "rpm"
    builtBy("rpm")
}
```

The `artifacts.add()` method — from ArtifactHandler — returns an artifact object of type PublishArtifact that can then be used in defining a publication, as shown in the following sample:

Example 462. Attaching a custom PublishArtifact to a publication

build.gradle

```
publishing {
    publications {
        maven(MavenPublication) {
            artifact rpmArtifact
        }
    }
}
```

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("maven") {
            artifact(rpmArtifact)
        }
    }
}
```

- The `artifact()` method accepts *publish artifacts* as argument — like `rpmArtifact` in the sample — as well as any type of argument accepted by Project.file(java.lang.Object), such as a `File` instance, a string file path or a archive task.
- Publishing plugins support different artifact configuration properties, so always check the plugin documentation for more details. The `classifier` and `extension` properties are supported by both the Maven Publish Plugin and the Ivy Publish Plugin.
- Custom artifacts need to be distinct within a publication, typically via a unique combination of `classifier` and `extension`. See the documentation for the plugin you're using for the precise requirements.
- If you use `artifact()` with an archive task, Gradle automatically populates the artifact's metadata with the `classifier` and `extension` properties from that task.

Now you can publish the RPM.

If you really want to add an artifact to a publication based on a component, instead of adjusting the component itself, you can combine the `from components.someComponent` and `artifact someArtifact` notations.

Restricting publications to specific repositories

When you have defined multiple publications or repositories, you often want to control which publications are published to which repositories. For instance, consider the following sample that

defines two publications — one that consists of just a binary and another that contains the binary and associated sources — and two repositories — one for internal use and one for external consumers:

Example 463. Adding multiple publications and repositories

build.gradle

```
publishing {
    publications {
        binary(MavenPublication) {
            from components.java
        }
        binaryAndSources(MavenPublication) {
            from components.java
            artifact sourcesJar
        }
    }
    repositories {
        // change URLs to point to your repos, e.g. http://my.org/repo
        maven {
            name = 'external'
            url = layout.buildDirectory.dir('repos/external')
        }
        maven {
            name = 'internal'
            url = layout.buildDirectory.dir('repos/internal')
        }
    }
}
```

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("binary") {
            from(components["java"])
        }
        create<MavenPublication>("binaryAndSources") {
            from(components["java"])
            artifact(tasks["sourcesJar"])
        }
    }
    repositories {
        // change URLs to point to your repos, e.g. http://my.org/repo
        maven {
            name = "external"
            url = uri(layout.buildDirectory.dir("repos/external"))
        }
        maven {
            name = "internal"
            url = uri(layout.buildDirectory.dir("repos/internal"))
        }
    }
}
```

The publishing plugins will create tasks that allow you to publish either of the publications to either repository. They also attach those tasks to the `publish` aggregate task. But let's say you want to restrict the binary-only publication to the external repository and the binary-with-sources publication to the internal one. To do that, you need to make the publishing *conditional*.

Gradle allows you to skip any task you want based on a condition via the Task.onlyIf(String, org.gradle.api.specs.Spec) method. The following sample demonstrates how to implement the constraints we just mentioned:

Example 464. Configuring which artifacts should be published to which repositories

build.gradle

```
tasks.withType(PublishToMavenRepository) {
    onlyIf("publishing binary to the external repository, or binary and
sources to the internal one") {
        (repository == publishing.repositories.external &&
            publication == publishing.publications.binary) ||
        (repository == publishing.repositories.internal &&
            publication == publishing.publications.binaryAndSources)
    }
}
tasks.withType(PublishToMavenLocal) {
    onlyIf("publishing binary and sources") {
        publication == publishing.publications.binaryAndSources
    }
}
```

build.gradle.kts

```
tasks.withType<PublishToMavenRepository>().configureEach {
    onlyIf("publishing binary to the external repository, or binary and
sources to the internal one") {
        (repository == publishing.repositories["external"] &&
            publication == publishing.publications["binary"]) ||
        (repository == publishing.repositories["internal"] &&
            publication == publishing.publications["binaryAndSources"])
    }
}
tasks.withType<PublishToMavenLocal>().configureEach {
    onlyIf("publishing binary and sources") {
        publication == publishing.publications["binaryAndSources"]
    }
}
```

Output of `gradle publish`

```
> gradle publish
> Task :compileJava
> Task :processResources
> Task :classes
> Task :jar
> Task :generateMetadataFileForBinaryAndSourcesPublication
> Task :generatePomFileForBinaryAndSourcesPublication
> Task :sourcesJar
> Task :publishBinaryAndSourcesPublicationToExternalRepository SKIPPED
> Task :publishBinaryAndSourcesPublicationToInternalRepository
> Task :generateMetadataFileForBinaryPublication
> Task :generatePomFileForBinaryPublication
> Task :publishBinaryPublicationToExternalRepository
> Task :publishBinaryPublicationToInternalRepository SKIPPED
> Task :publish

BUILD SUCCESSFUL in 0s
10 actionable tasks: 10 executed
```

You may also want to define your own aggregate tasks to help with your workflow. For example, imagine that you have several publications that should be published to the external repository. It could be very useful to publish all of them in one go without publishing the internal ones.

The following sample demonstrates how you can do this by defining an aggregate task — `publishToExternalRepository` — that depends on all the relevant publish tasks:

Example 465. Defining your own shorthand tasks for publishing

build.gradle

```
tasks.register('publishToExternalRepository') {
    group = 'publishing'
    description = 'Publishes all Maven publications to the external Maven
repository.'
    dependsOn tasks.withType(PublishToMavenRepository).matching {
        it.repository == publishing.repositories.external
    }
}
```

build.gradle.kts

```
tasks.register("publishToExternalRepository") {
    group = "publishing"
    description = "Publishes all Maven publications to the external Maven
repository."
    dependsOn(tasks.withType<PublishToMavenRepository>().matching {
        it.repository == publishing.repositories["external"]
    })
}
```

This particular sample automatically handles the introduction or removal of the relevant publishing tasks by using TaskCollection.withType(java.lang.Class) with the PublishToMavenRepository task type. You can do the same with PublishToIvyRepository if you're publishing to Ivy-compatible repositories.

Configuring publishing tasks

The publishing plugins create their non-aggregate tasks after the project has been evaluated, which means you cannot directly reference them from your build script. If you would like to configure any of these tasks, you should use deferred task configuration. This can be done in a number of ways via the project's `tasks` collection.

For example, imagine you want to change where the `generatePomFileForPubNamePublication` tasks write their POM files. You can do this by using the TaskCollection.withType(java.lang.Class) method, as demonstrated by this sample:

Example 466. Configuring a dynamically named task created by the publishing plugins

build.gradle

```
tasks.withType(GenerateMavenPom).all {
    def matcher = name =~ /generatePomFileFor(\w+)Publication/
    def publicationName = matcher[0][1]
    destination = layout.buildDirectory.file("poms/${publicationName}-
pom.xml").get().asFile
}
```

build.gradle.kts

```
tasks.withType<GenerateMavenPom>().configureEach {
    val matcher =
Regex("""generatePomFileFor(\w+)Publication""").matchEntire(name)
    val publicationName = matcher?.let { it.groupValues[1] }
    destination = layout.buildDirectory.file("poms/${publicationName}-
pom.xml").get().asFile
}
```

The above sample uses a regular expression to extract the name of the publication from the name of the task. This is so that there is no conflict between the file paths of all the POM files that might be generated. If you only have one publication, then you don't have to worry about such conflicts since there will only be one POM file.

Dependency Management Terminology

Dependency management comes with a wealth of terminology. Here you can find the most commonly-used terms including references to the user guide to learn about their practical application.

Artifact

A file or directory produced by a build, such as a JAR, a ZIP distribution, or a native executable.

Artifacts are typically designed to be used or consumed by users or other projects, or deployed to hosting systems. In such cases, the artifact is a single file. Directories are common in the case of inter-project dependencies to avoid the cost of producing the publishable artifact.

Capability

A capability identifies a feature offered by one or multiple components. A capability is identified by coordinates similar to the coordinates used for module versions. By default, each module version offers a capability that matches its coordinates, for example `com.google:guava:18.0`. Capabilities can

be used to express that a component provides multiple feature variants or that two different components implement the same feature (and thus cannot be used together). For more details, see the section on capabilities.

Component

Any single version of a module.

For external libraries, the term component refers to one published version of the library.

In a build, components are defined by plugins (e.g. the Java Library plugin) and provide a simple way to define a publication for publishing. They comprise artifacts as well as the appropriate metadata that describes a component's variants in detail. For example, the `java` component in its default setup consists of a JAR — produced by the `jar` task — and the dependency information of the Java *api* and *runtime* variants. It may also define additional variants, for example *sources* and *Javadoc*, with the corresponding artifacts.

Configuration

A configuration is a named set of dependencies grouped together for a specific goal. Configurations provide access to the underlying, resolved modules and their artifacts. For more information, see the sections on dependency configurations as well as resolvable and consumable configurations.

NOTE	The word "configuration" is an overloaded term and has a different meaning outside of the context of dependency management.

Dependency

A dependency is a pointer to another piece of software required to build, test or run a module. For more information, see the section on declaring dependencies.

Dependency constraint

A dependency constraint defines requirements that need to be met by a module to make it a valid resolution result for the dependency. For example, a dependency constraint can narrow down the set of supported module versions. Dependency constraints can be used to express such requirements for transitive dependencies. For more information, see the sections on upgrading and downgrading transitive dependencies.

Feature Variant

A feature variant is a variant representing a feature of a component that can be individually selected or not. A feature variant is identified by one or more capabilities. For more information, see the sections on modeling feature variants and optional dependencies.

Module

A piece of software that evolves over time e.g. Google Guava. Every module has a name. Each release of a module is optimally represented by a module version. For convenient consumption, modules can be hosted in a repository.

Module metadata

Releases of a module provide metadata. Metadata is the data that describes the module in more detail e.g. information about the location of artifacts or required transitive dependencies. Gradle offers its own metadata format called Gradle Module Metadata (`.module` file) but also supports Maven (`.pom`) and Ivy (`ivy.xml`) metadata. See the section on understanding Gradle Module Metadata for more information on the supported metadata formats.

Component metadata rule

A component metadata rule is a rule that modifies a component's metadata after it was fetched from a repository, e.g. to add missing information or to correct wrong information. In contrast to resolution rules, component metadata rules are applied **before** resolution starts. Component metadata rules are defined as part of the build logic and can be shared through plugins. For more information, see the section on fixing metadata with component metadata rules.

Module version

A module version represents a distinct set of changes of a released module. For example `18.0` represents the version of the module with the coordinates `com.google:guava:18.0`. In practice there's no limitation to the scheme of the module version. Timestamps, numbers, special suffixes like `-GA` are all allowed identifiers. The most widely-used versioning strategy is semantic versioning.

Platform

A platform is a set of modules aimed to be used together. There are different categories of platforms, corresponding to different use cases:

- module set: often a set of modules published together as a whole. Using one module of the set often means we want to use the same version for all modules of the set. For example, if using `groovy` 1.2, also use `groovy-json` 1.2.
- runtime environment: a set of libraries known to work well together. e.g., the Spring Platform, recommending versions for both Spring and components that work well with Spring.
- deployment environment: Java runtime, application server, ...

In addition Gradle defines virtual platforms.

NOTE	Maven's BOM (bill-of-material) is a popular kind of platform that Gradle supports.

Publication

A description of the files and metadata that should be published to a repository as a single entity for use by consumers.

A publication has a name and consists of one or more artifacts plus information about those artifacts (the metadata).

Repository

A repository hosts a set of modules, each of which may provide one or many releases (components) indicated by a module version. The repository can be based on a binary repository product (e.g. Artifactory or Nexus) or a directory structure in the filesystem. For more information, see Declaring Repositories.

Resolution rule

A resolution rule influences the behavior of how a dependency is resolved directly. Resolution rules are defined as part of the build logic. For more information, see the section on customizing resolution of a dependency directly.

Transitive dependency

A variant of a component can have dependencies on other modules to work properly, so-called transitive dependencies. Releases of a module hosted on a repository can provide metadata to declare those transitive dependencies. By default, Gradle resolves transitive dependencies automatically. The version selection for transitive dependencies can be influenced by declaring dependency constraints.

Variant (of a component)

Each component consists of one or more variants. A variant consists of a set of artifacts and defines a set of dependencies. It is identified by a set of attributes and capabilities.

Gradle's dependency resolution is variant-aware and selects one or more variants of each component after a component (i.e. one version of a module) has been selected. It may also fail if the variant selection result is ambiguous, meaning that Gradle does not have enough information to select one of multiple mutual exclusive variants. In that case, more information can be provided through variant attributes. Examples of variants each Java components typically offers are *api* and *runtime* variants. Others examples are JDK8 and JDK11 variants. For more information, see the section on variant selection.

Variant Attribute

Attributes are used to identify and select variants. A variant has one or more attributes defined, for example `org.gradle.usage=java-api`, `org.gradle.jvm.version=11`. When dependencies are resolved, a set of attributes are requested and Gradle finds the best fitting variant(s) for each component in the dependency graph. Compatibility and disambiguation rules can be implemented for an attribute to express compatibility between values (e.g. Java 8 is compatible with Java 11, but Java 11 should be preferred if the requested version is 11 or higher). Such rules are typically provided by plugins. For more information, see the sections on variant selection and declaring attributes.

Java & Other JVM Projects

Building Java & JVM projects

Gradle uses a convention-over-configuration approach to building JVM-based projects that borrows several conventions from Apache Maven. In particular, it uses the same default directory structure for source files and resources, and it works with Maven-compatible repositories.

We will look at Java projects in detail in this chapter, but most of the topics apply to other supported JVM languages as well, such as Kotlin, Groovy and Scala. If you don't have much experience with building JVM-based projects with Gradle, take a look at the Java samples for step-by-step instructions on how to build various types of basic Java projects.

NOTE	The example in this section use the Java Library Plugin. However the described features are shared by all JVM plugins. Specifics of the different plugins are available in their dedicated documentation.

NOTE	There are a number of hands-on samples that you can explore for Java, Groovy, Scala and Kotlin

Introduction

The simplest build script for a Java project applies the Java Library Plugin and optionally sets the project version and selects the Java toolchain to use:

Example 467. Applying the Java Library Plugin

build.gradle

```
plugins {
    id 'java-library'
}

java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(11)
    }
}

version = '1.2.1'
```

build.gradle.kts

```
plugins {
    `java-library`
}

java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(11))
    }
}

version = "1.2.1"
```

By applying the Java Library Plugin, you get a whole host of features:

- A `compileJava` task that compiles all the Java source files under *src/main/java*
- A `compileTestJava` task for source files under *src/test/java*
- A `test` task that runs the tests from *src/test/java*
- A `jar` task that packages the `main` compiled classes and resources from *src/main/resources* into a single JAR named *<project>-<version>.jar*
- A `javadoc` task that generates Javadoc for the `main` classes

This isn't sufficient to build any non-trivial Java project — at the very least, you'll probably have some file dependencies. But it means that your build script only needs the information that is specific to *your* project.

NOTE	Although the properties in the example are optional, we recommend that you specify them in your projects. The toolchain options protects against problems with the project being built with different Java versions. The version string is important for tracking the progression of the project. The project version is also used in archive names by default.

The Java Library Plugin also integrates the above tasks into the standard Base Plugin lifecycle tasks:

- `jar` is attached to `assemble` [7]
- `test` is attached to `check`

The rest of the chapter explains the different avenues for customizing the build to your requirements. You will also see later how to adjust the build for libraries, applications, web apps and enterprise apps.

Declaring your source files via source sets

Gradle's Java support was the first to introduce a new concept for building source-based projects: *source sets*. The main idea is that source files and resources are often logically grouped by type, such as application code, unit tests and integration tests. Each logical group typically has its own sets of file dependencies, classpaths, and more. Significantly, the files that form a source set *don't have to be located in the same directory*!

Source sets are a powerful concept that tie together several aspects of compilation:

- the source files and where they're located
- the compilation classpath, including any required dependencies (via Gradle configurations)
- where the compiled class files are placed

You can see how these relate to one another in this diagram:

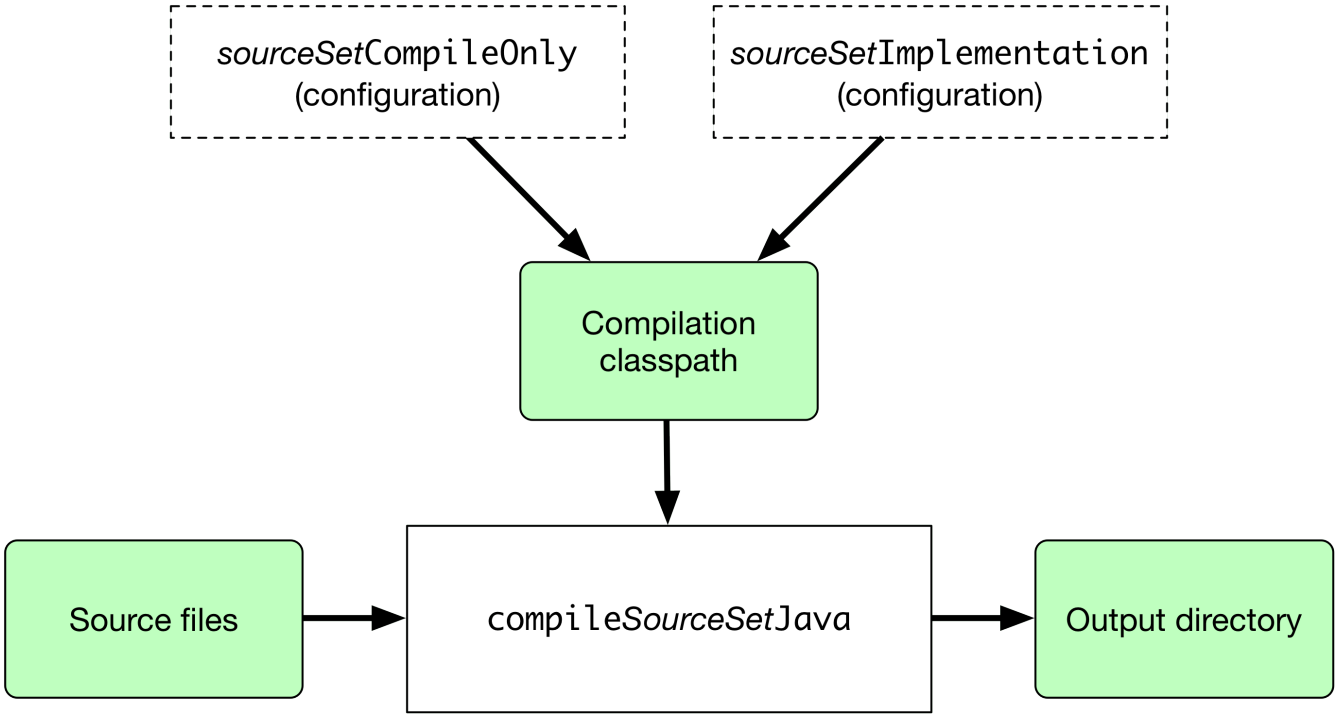


Figure 25. Source sets and Java compilation

The shaded boxes represent properties of the source set itself. On top of that, the Java Library Plugin automatically creates a compilation task for every source set you or a plugin defines — named `compileSourceSetJava` — and several dependency configurations.

NOTE

The `main` source set

Most language plugins, Java included, automatically create a source set called `main`, which is used for the project's production code. This source set is special in that its name is not included in the names of the configurations and tasks, hence why you have just a `compileJava` task and `compileOnly` and `implementation` configurations rather than `compileMainJava`, `mainCompileOnly` and `mainImplementation` respectively.

Java projects typically include resources other than source files, such as properties files, that may need processing — for example by replacing tokens within the files — and packaging within the final JAR. The Java Library Plugin handles this by automatically creating a dedicated task for each defined source set called `processSourceSetResources` (or `processResources` for the `main` source set). The following diagram shows how the source set fits in with this task:

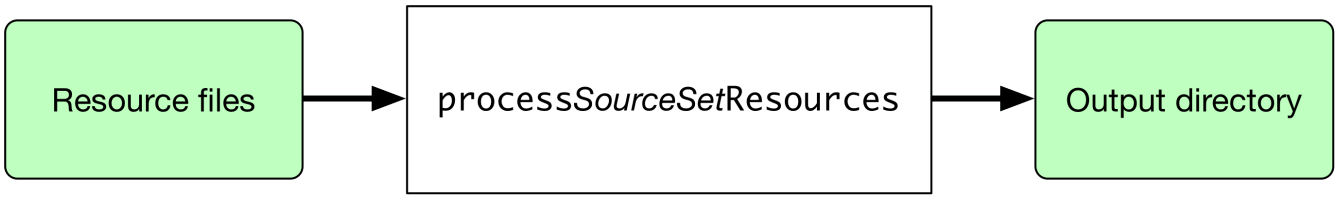


Figure 26. Processing non-source files for a source set

As before, the shaded boxes represent properties of the source set, which in this case comprises the locations of the resource files and where they are copied to.

In addition to the `main` source set, the Java Library Plugin defines a `test` source set that represents the project's tests. This source set is used by the `test` task, which runs the tests. You can learn more about this task and related topics in the Java testing chapter.

Projects typically use this source set for unit tests, but you can also use it for integration, acceptance and other types of test if you wish. The alternative approach is to define a new source set for each of your other test types, which is typically done for one or both of the following reasons:

- You want to keep the tests separate from one another for aesthetics and manageability
- The different test types require different compilation or runtime classpaths or some other difference in setup

You can see an example of this approach in the Java testing chapter, which shows you how to set up integration tests in a project.

You'll learn more about source sets and the features they provide in:

- Customizing file and directory locations
- Configuring Java integration tests

Managing your dependencies

The vast majority of Java projects rely on libraries, so managing a project's dependencies is an important part of building a Java project. Dependency management is a big topic, so we will focus on the basics for Java projects here. If you'd like to dive into the detail, check out the introduction to dependency management.

Specifying the dependencies for your Java project requires just three pieces of information:

- Which dependency you need, such as a name and version
- What it's needed for, e.g. compilation or running
- Where to look for it

The first two are specified in a `dependencies {}` block and the third in a `repositories {}` block. For example, to tell Gradle that your project requires version 3.6.7 of Hibernate Core to compile and run your production code, and that you want to download the library from the Maven Central repository, you can use the following fragment:

Example 468. Declaring dependencies

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.hibernate:hibernate-core:3.6.7.Final'
}
```

build.gradle.kts

```
repositories {
    mavenCentral()
}

dependencies {
    implementation("org.hibernate:hibernate-core:3.6.7.Final")
}
```

The Gradle terminology for the three elements is as follows:

- *Repository* (ex: `mavenCentral()`) — where to look for the modules you declare as dependencies
- *Configuration* (ex: `implementation`) — a named collection of dependencies, grouped together for a specific goal such as compiling or running a module — a more flexible form of Maven scopes
- *Module coordinate* (ex: `org.hibernate:hibernate-core-3.6.7.Final`) — the ID of the dependency, usually in the form '<*group*>:<*module*>:<*version*>' (or '<*groupId*>:<*artifactId*>:<*version*>' in Maven terminology)

You can find a more comprehensive glossary of dependency management terms here.

As far as configurations go, the main ones of interest are:

- `compileOnly` — for dependencies that are necessary to compile your production code but shouldn't be part of the runtime classpath
- `implementation` (supersedes `compile`) — used for compilation and runtime
- `runtimeOnly` (supersedes `runtime`) — only used at runtime, not for compilation
- `testCompileOnly` — same as `compileOnly` except it's for the tests
- `testImplementation` — test equivalent of `implementation`
- `testRuntimeOnly` — test equivalent of `runtimeOnly`

You can learn more about these and how they relate to one another in the plugin reference chapter.

Be aware that the Java Library Plugin offers two additional configurations — `api` and `compileOnlyApi` — for dependencies that are required for compiling both the module and any modules that depend on it.

NOTE

Why no `compile` configuration?

The Java Library Plugin has historically used the `compile` configuration for dependencies that are required to both compile and run a project's production code. It is now deprecated, and will issue warnings when used, because it doesn't distinguish between dependencies that impact the public API of a Java library project and those that don't. You can learn more about the importance of this distinction in Building Java libraries.

We have only scratched the surface here, so we recommend that you read the dedicated dependency management chapters once you're comfortable with the basics of building Java projects with Gradle. Some common scenarios that require further reading include:

- Defining a custom Maven- or Ivy-compatible repository
- Using dependencies from a local filesystem directory
- Declaring dependencies with changing (e.g. SNAPSHOT) and dynamic (range) versions
- Declaring a sibling project as a dependency
- Controlling transitive dependencies and their versions
- Testing your fixes to a 3rd-party dependency via composite builds (a better alternative to publishing to and consuming from Maven Local)

You'll discover that Gradle has a rich API for working with dependencies — one that takes time to master, but is straightforward to use for common scenarios.

Compiling your code

Compiling both your production and test code can be trivially easy if you follow the conventions:

1. Put your production source code under the *src/main/java* directory
2. Put your test source code under *src/test/java*
3. Declare your production compile dependencies in the `compileOnly` or `implementation` configurations (see previous section)
4. Declare your test compile dependencies in the `testCompileOnly` or `testImplementation` configurations
5. Run the `compileJava` task for the production code and `compileTestJava` for the tests

Other JVM language plugins, such as the one for Groovy, follow the same pattern of conventions. We recommend that you follow these conventions wherever possible, but you don't have to. There are several options for customization, as you'll see next.

Customizing file and directory locations

Imagine you have a legacy project that uses an *src* directory for the production code and *test* for the test code. The conventional directory structure won't work, so you need to tell Gradle where to find the source files. You do that via source set configuration.

Each source set defines where its source code resides, along with the resources and the output directory for the class files. You can override the convention values by using the following syntax:

Example 469. Declaring custom source directories

build.gradle

```
sourceSets {
    main {
         java {
            srcDirs = ['src']
         }
    }

    test {
        java {
            srcDirs = ['test']
        }
    }
}
```

build.gradle.kts

```
sourceSets {
    main {
        java {
            setSrcDirs(listOf("src"))
        }
    }

    test {
        java {
            setSrcDirs(listOf("test"))
        }
    }
}
```

Now Gradle will only search directly in *src* and *test* for the respective source code. What if you don't want to override the convention, but simply want to *add* an extra source directory, perhaps one that contains some third-party source code you want to keep separate? The syntax is similar:

Example 470. Declaring custom source directories additively

build.gradle

```
sourceSets {
    main {
        java {
            srcDir 'thirdParty/src/main/java'
        }
    }
}
```

build.gradle.kts

```
sourceSets {
    main {
        java {
            srcDir("thirdParty/src/main/java")
        }
    }
}
```

Crucially, we're using the *method* `srcDir()` here to append a directory path, whereas setting the `srcDirs` property replaces any existing values. This is a common convention in Gradle: setting a property replaces values, while the corresponding method appends values.

You can see all the properties and methods available on source sets in the DSL reference for SourceSet and SourceDirectorySet. Note that `srcDirs` and `srcDir()` are both on `SourceDirectorySet`.

Changing compiler options

Most of the compiler options are accessible through the corresponding task, such as `compileJava` and `compileTestJava`. These tasks are of type JavaCompile, so read the task reference for an up-to-date and comprehensive list of the options.

For example, if you want to use a separate JVM process for the compiler and prevent compilation failures from failing the build, you can use this configuration:

Example 471. Setting Java compiler options

build.gradle

```
compileJava {
    options.incremental = true
    options.fork = true
    options.failOnError = false
}
```

build.gradle.kts

```
tasks.compileJava {
    options.isIncremental = true
    options.isFork = true
    options.isFailOnError = false
}
```

That's also how you can change the verbosity of the compiler, disable debug output in the byte code and configure where the compiler can find annotation processors.

Targeting a specific Java version

By default, Gradle will compile Java code to the language level of the JVM running Gradle. With the usage of Java toolchains, you can break that link by making sure a given Java version, defined by the build, is used for compilation, execution and documentation. It is however possible to override some compiler and execution options at the task level.

Since version 9, the Java compiler can be configured to produce bytecode for an older Java version while making sure the code does not use any APIs from a more recent version. Gradle now supports this release flag on `CompileOptions` directly for Java compilation. This option takes precedence over the properties described below.

WARNING	Due to a bug in Java 9 that was fixed in Java 10, Gradle cannot leverage the `release` flag when compiling with Java 9.

Example 472. Setting Java release flag

build.gradle

```
compileJava {
    options.release = 7
}
```

build.gradle.kts

```
tasks.compileJava {
    options.release.set(7)
}
```

Historical options for the Java compiler remain available:

`sourceCompatibility`
: Defines which language version of Java your source files should be treated as.

`targetCompatibility`
: Defines the minimum JVM version your code should run on, i.e. it determines the version of byte code the compiler generates.

These options can be set per JavaCompile task, or on the `java { }` extension for all compile tasks, using properties with the same names.

NOTE	Using a toolchain makes it illegal to configure the `sourceCompatibility` or `targetCompatibility` at the `java { }` extension level.

However, these options do not protect against the use of APIs introduced in later Java versions.

Compiling and testing Java 6/7

Gradle can only run on Java version 8 or higher. Gradle still supports compiling, testing, generating Javadoc and executing applications for Java 6 and Java 7. Java 5 and below are not supported.

NOTE	If using Java 10+, leveraging the `release` flag might be an easier solution, see above.

To use Java 6 or Java 7, the following tasks need to be configured:

- `JavaCompile` task to fork and use the correct Java home
- `Javadoc` task to use the correct `javadoc` executable
- `Test` and the `JavaExec` task to use the correct `java` executable.

With the usage of Java toolchains, this can be done as follows:

Example: Configure Java 7 build

build.gradle

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(7)
    }
}
```

build.gradle.kts

```
java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(7))
    }
}
```

The only requirement is that Java 7 is installed and has to be either in a location Gradle can detect automatically or explicitly configured.

Compiling independent sources separately

Most projects have at least two independent sets of sources: the production code and the test code. Gradle already makes this scenario part of its Java convention, but what if you have other sets of sources? One of the most common scenarios is when you have separate integration tests of some form or other. In that case, a custom source set may be just what you need.

You can see a complete example for setting up integration tests in the Java testing chapter. You can set up other source sets that fulfil different roles in the same way. The question then becomes: when should you define a custom source set?

To answer that question, consider whether the sources:

1. Need to be compiled with a unique classpath
2. Generate classes that are handled differently from the `main` and `test` ones
3. Form a natural part of the project

If your answer to both 3 and either one of the others is yes, then a custom source set is probably the right approach. For example, integration tests are typically part of the project because they test the code in `main`. In addition, they often have either their own dependencies independent of the `test` source set or they need to be run with a custom `Test` task.

Other common scenarios are less clear cut and may have better solutions. For example:

- Separate API and implementation JARs — it may make sense to have these as separate projects, particularly if you already have a multi-project build
- Generated sources — if the resulting sources should be compiled with the production code, add their path(s) to the `main` source set and make sure that the `compileJava` task depends on the task that generates the sources

If you're unsure whether to create a custom source set or not, then go ahead and do so. It should be straightforward and if it's not, then it's probably not the right tool for the job.

Managing resources

Many Java projects make use of resources beyond source files, such as images, configuration files and localization data. Sometimes these files simply need to be packaged unchanged and sometimes they need to be processed as template files or in some other way. Either way, the Java Library Plugin adds a specific Copy task for each source set that handles the processing of its associated resources.

The task's name follows the convention of `processSourceSetResources` — or `processResources` for the `main` source set — and it will automatically copy any files in *src/[sourceSet]/resources* to a directory that will be included in the production JAR. This target directory will also be included in the runtime classpath of the tests.

Since `processResources` is an instance of the `ProcessResources` task, you can perform any of the processing described in the Working With Files chapter.

Java properties files and reproducible builds

You can easily create Java properties files via the WriteProperties task, which fixes a well-known problem with `Properties.store()` that can reduce the usefulness of incremental builds.

The standard Java API for writing properties files produces a unique file every time, even when the same properties and values are used, because it includes a timestamp in the comments. Gradle's `WriteProperties` task generates exactly the same output byte-for-byte if none of the properties have changed. This is achieved by a few tweaks to how a properties file is generated:

- no timestamp comment is added to the output
- the line separator is system independent, but can be configured explicitly (it defaults to `'\n'`)
- the properties are sorted alphabetically

Sometimes it can be desirable to recreate archives in a byte for byte way on different machines. You want to be sure that building an artifact from source code produces the same result, byte for byte, no matter when and where it is built. This is necessary for projects like reproducible-builds.org.

These tweaks not only lead to better incremental build integration, but they also help with reproducible builds. In essence, reproducible builds guarantee that you will see the same results from a build execution — including test results and production binaries — no matter when or on what system you run it.

Running tests

Alongside providing automatic compilation of unit tests in *src/test/java*, the Java Library Plugin has native support for running tests that use JUnit 3, 4 & 5 (JUnit 5 support came in Gradle 4.6) and TestNG. You get:

- An automatic `test` task of type Test, using the `test` source set
- An HTML test report that includes the results from *all* `Test` tasks that run
- Easy filtering of which tests to run
- Fine-grained control over how the tests are run
- The opportunity to create your own test execution and test reporting tasks

You do *not* get a `Test` task for every source set you declare, since not every source set represents tests! That's why you typically need to create your own `Test` tasks for things like integration and acceptance tests if they can't be included with the `test` source set.

As there is a lot to cover when it comes to testing, the topic has its own chapter in which we look at:

- How tests are run
- How to run a subset of tests via filtering
- How Gradle discovers tests
- How to configure test reporting and add your own reporting tasks
- How to make use of specific JUnit and TestNG features

You can also learn more about configuring tests in the DSL reference for Test.

Packaging and publishing

How you package and potentially publish your Java project depends on what type of project it is. Libraries, applications, web applications and enterprise applications all have differing requirements. In this section, we will focus on the bare bones provided by the Java Library Plugin.

By default, the Java Library Plugin provides the `jar` task that packages all the compiled production classes and resources into a single JAR. This JAR is also automatically built by the `assemble` task. Furthermore, the plugin can be configured to provide the `javadocJar` and `sourcesJar` tasks to package Javadoc and source code if so desired. If a publishing plugin is used, these tasks will automatically run during publishing or can be called directly.

Example 473. Configure a project to publish Javadoc and sources

build.gradle

```
java {
    withJavadocJar()
    withSourcesJar()
}
```

build.gradle.kts

```
java {
    withJavadocJar()
    withSourcesJar()
}
```

If you want to create an 'uber' (AKA 'fat') JAR, then you can use a task definition like this:

Example 474. Creating a Java uber or fat JAR

build.gradle

```
plugins {
    id 'java'
}

version = '1.0.0'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'commons-io:commons-io:2.6'
}

tasks.register('uberJar', Jar) {
    archiveClassifier = 'uber'

    from sourceSets.main.output

    dependsOn configurations.runtimeClasspath
    from {
        configurations.runtimeClasspath.findAll { it.name.endsWith('jar') }
.collect { zipTree(it) }
    }
}
```

build.gradle.kts

```
plugins {
    java
}

version = "1.0.0"

repositories {
    mavenCentral()
}

dependencies {
    implementation("commons-io:commons-io:2.6")
}

tasks.register<Jar>("uberJar") {
    archiveClassifier.set("uber")

    from(sourceSets.main.get().output)

    dependsOn(configurations.runtimeClasspath)
    from({
        configurations.runtimeClasspath.get().filter {
it.name.endsWith("jar") }.map { zipTree(it) }
    })
}
```

See Jar for more details on the configuration options available to you. And note that you need to use `archiveClassifier` rather than `archiveAppendix` here for correct publication of the JAR.

You can use one of the publishing plugins to publish the JARs created by a Java project:

- Maven Publish Plugin
- Ivy Publish Plugin

Modifying the JAR manifest

Each instance of the `Jar`, `War` and `Ear` tasks has a `manifest` property that allows you to customize the *MANIFEST.MF* file that goes into the corresponding archive. The following example demonstrates how to set attributes in the JAR's manifest:

Example 475. Customization of MANIFEST.MF

build.gradle

```
jar {
    manifest {
        attributes("Implementation-Title": "Gradle",
                   "Implementation-Version": archiveVersion)
    }
}
```

build.gradle.kts

```
tasks.jar {
    manifest {
        attributes(
            "Implementation-Title" to "Gradle",
            "Implementation-Version" to archiveVersion
        )
    }
}
```

See Manifest for the configuration options it provides.

You can also create standalone instances of `Manifest`. One reason for doing so is to share manifest information between JARs. The following example demonstrates how to share common attributes between JARs:

Example 476. Creating a manifest object.

build.gradle

```
ext.sharedManifest = manifest {
    attributes("Implementation-Title": "Gradle",
               "Implementation-Version": version)
}
tasks.register('fooJar', Jar) {
    manifest = project.manifest {
        from sharedManifest
    }
}
```

build.gradle.kts

```
val sharedManifest = the<JavaPluginConvention>().manifest {
    attributes (
        "Implementation-Title" to "Gradle",
        "Implementation-Version" to version
    )
}

tasks.register<Jar>("fooJar") {
    manifest = project.the<JavaPluginConvention>().manifest {
        from(sharedManifest)
    }
}
```

Another option available to you is to merge manifests into a single `Manifest` object. Those source manifests can take the form of a text for or another `Manifest` object. In the following example, the source manifests are all text files except for `sharedManifest`, which is the `Manifest` object from the previous example:

Example 477. Separate MANIFEST.MF for a particular archive

build.gradle

```
tasks.register('barJar', Jar) {
    manifest {
        attributes key1: 'value1'
        from sharedManifest, 'src/config/basemanifest.txt'
        from(['src/config/javabasemanifest.txt',
'src/config/libbasemanifest.txt']) {
            eachEntry { details ->
                if (details.baseValue != details.mergeValue) {
                    details.value = baseValue
                }
                if (details.key == 'foo') {
                    details.exclude()
                }
            }
        }
    }
}
```

build.gradle.kts

```
tasks.register<Jar>("barJar") {
    manifest {
        attributes("key1" to "value1")
        from(sharedManifest, "src/config/basemanifest.txt")
        from(listOf("src/config/javabasemanifest.txt",
"src/config/libbasemanifest.txt")) {
            eachEntry(Action<ManifestMergeDetails> {
                if (baseValue != mergeValue) {
                    value = baseValue
                }
                if (key == "foo") {
                    exclude()
                }
            })
        }
    }
}
```

Manifests are merged in the order they are declared in the `from` statement. If the base manifest and the merged manifest both define values for the same key, the merged manifest wins by default. You can fully customize the merge behavior by adding `eachEntry` actions in which you have access to a

ManifestMergeDetails instance for each entry of the resulting manifest. Note that the merge is done lazily, either when generating the JAR or when `Manifest.writeTo()` or `Manifest.getEffectiveManifest()` are called.

Speaking of `writeTo()`, you can use that to easily write a manifest to disk at any time, like so:

Example 478. Saving a MANIFEST.MF to disk

build.gradle

```
tasks.named('jar') { manifest.writeTo(layout.buildDirectory.file(
'mymanifest.mf')) }
```

build.gradle.kts

```
tasks.jar { manifest.writeTo(layout.buildDirectory.file("mymanifest.mf")) }
```

Generating API documentation

The Java Library Plugin provides a `javadoc` task of type Javadoc, that will generate standard Javadocs for all your production code, i.e. whatever source is in the `main` source set. The task supports the core Javadoc and standard doclet options described in the Javadoc reference documentation. See CoreJavadocOptions and StandardJavadocDocletOptions for a complete list of those options.

As an example of what you can do, imagine you want to use Asciidoc syntax in your Javadoc comments. To do this, you need to add Asciidoclet to Javadoc's doclet path. Here's an example that does just that:

Example 479. Using a custom doclet with Javadoc

build.gradle

```
configurations {
    asciidoclet
}

dependencies {
    asciidoclet 'org.asciidoctor:asciidoclet:1.+'
}

tasks.register('configureJavadoc') {
    doLast {
        javadoc {
            options.doclet = 'org.asciidoctor.Asciidoclet'
            options.docletpath = configurations.asciidoclet.files.toList()
        }
    }
}

javadoc {
    dependsOn configureJavadoc
}
```

build.gradle.kts

```
val asciidoclet by configurations.creating

dependencies {
    asciidoclet("org.asciidoctor:asciidoclet:1.+")
}

tasks.register("configureJavadoc") {
    doLast {
        tasks.javadoc {
            options.doclet = "org.asciidoctor.Asciidoclet"
            options.docletpath = asciidoclet.files.toList()
        }
    }
}

tasks.javadoc {
    dependsOn("configureJavadoc")
}
```

You don't have to create a configuration for this, but it's an elegant way to handle dependencies that are required for a unique purpose.

You might also want to create your own Javadoc tasks, for example to generate API docs for the tests:

Example 480. Defining a custom Javadoc task

build.gradle

```
tasks.register('testJavadoc', Javadoc) {
    source = sourceSets.test.allJava
}
```

build.gradle.kts

```
tasks.register<Javadoc>("testJavadoc") {
    source = sourceSets.test.get().allJava
}
```

These are just two non-trivial but common customizations that you might come across.

Cleaning the build

The Java Library Plugin adds a `clean` task to your project by virtue of applying the Base Plugin. This task simply deletes everything in the `$buildDir` directory, hence why you should always put files generated by the build in there. The task is an instance of Delete and you can change what directory it deletes by setting its `dir` property.

Building JVM components

All of the specific JVM plugins are built on top of the Java Plugin. The examples above only illustrated concepts provided by this base plugin and shared with all JVM plugins.

Read on to understand which plugins fits which project type, as it is recommended to pick a specific plugin instead of applying the Java Plugin directly.

Building Java libraries

The unique aspect of library projects is that they are used (or "consumed") by other Java projects. That means the dependency metadata published with the JAR file — usually in the form of a Maven POM — is crucial. In particular, consumers of your library should be able to distinguish between two different types of dependencies: those that are only required to compile your library and those that are also required to compile the consumer.

Gradle manages this distinction via the Java Library Plugin, which introduces an *api* configuration in addition to the *implementation* one covered in this chapter. If the types from a dependency appear in public fields or methods of your library's public classes, then that dependency is exposed via your library's public API and should therefore be added to the *api* configuration. Otherwise, the dependency is an internal implementation detail and should be added to *implementation*.

If you're unsure of the difference between an API and implementation dependency, the Java Library Plugin chapter has a detailed explanation. In addition, you can explore a basic, practical sample of building a Java library.

Building Java applications

Java applications packaged as a JAR aren't set up for easy launching from the command line or a desktop environment. The Application Plugin solves the command line aspect by creating a distribution that includes the production JAR, its dependencies and launch scripts Unix-like and Windows systems.

See the plugin's chapter for more details, but here's a quick summary of what you get:

- `assemble` creates ZIP and TAR distributions of the application containing everything needed to run it
- A `run` task that starts the application from the build (for easy testing)
- Shell and Windows Batch scripts to start the application

You can see a basic example of building a Java application in the corresponding sample.

Building Java web applications

Java web applications can be packaged and deployed in a number of ways depending on the technology you use. For example, you might use Spring Boot with a fat JAR or a Reactive-based system running on Netty. Whatever technology you use, Gradle and its large community of plugins will satisfy your needs. Core Gradle, though, only directly supports traditional Servlet-based web applications deployed as WAR files.

That support comes via the War Plugin, which automatically applies the Java Plugin and adds an extra packaging step that does the following:

- Copies static resources from *src/main/webapp* into the root of the WAR
- Copies the compiled production classes into a *WEB-INF/classes* subdirectory of the WAR
- Copies the library dependencies into a *WEB-INF/lib* subdirectory of the WAR

This is done by the `war` task, which effectively replaces the `jar` task — although that task remains — and is attached to the `assemble` lifecycle task. See the plugin's chapter for more details and configuration options.

There is no core support for running your web application directly from the build, but we do recommend that you try the Gretty community plugin, which provides an embedded Servlet container.

Building Java EE applications

Java enterprise systems have changed a lot over the years, but if you're still deploying to JEE application servers, you can make use of the Ear Plugin. This adds conventions and a task for building EAR files. The plugin's chapter has more details.

Building Java Platforms

A Java platform represents a set of dependency declarations and constraints that form a cohesive unit to be applied on consuming projects. The platform has no source and no artifact of its own. It maps in the Maven world to a BOM.

The support comes via the Java Platform plugin, which sets up the different configurations and publication components.

NOTE	This plugin is the exception as it does not apply the Java Plugin.

Enabling Java preview features

WARNING	Using a Java preview feature is very likely to make your code incompatible with that compiled without a feature preview. As a consequence, we strongly recommend you not to publish libraries compiled with preview features and restrict the use of feature previews to toy projects.

To enable Java preview features for compilation, test execution and runtime, you can use the following DSL snippet:

Example 481. Enabling Java feature preview

build.gradle

```
tasks.withType(JavaCompile) {
    options.compilerArgs += "--enable-preview"
}
tasks.withType(Test) {
    jvmArgs += "--enable-preview"
}
tasks.withType(JavaExec) {
    jvmArgs += "--enable-preview"
}
```

build.gradle.kts

```
tasks.withType<JavaCompile> {
    options.compilerArgs.add("--enable-preview")
}
tasks.withType<Test> {
    jvmArgs("--enable-preview")
}
tasks.withType<JavaExec> {
    jvmArgs("--enable-preview")
}
```

Building other JVM language projects

If you want to leverage the multi language aspect of the JVM, most of what was described here will still apply.

Gradle itself provides Groovy and Scala plugins. The plugins automatically apply support for compiling Java code and can be further enhanced by combining them with the `java-library` plugin.

Compilation dependency between languages

These plugins create a dependency between Groovy/Scala compilation and Java compilation (of source code in the `java` folder of a source set). You can change this default behavior by adjusting the classpath of the involved compile tasks as shown in the following example:

Example 482. Changing the classpath of compile tasks

build.gradle

```
tasks.named('compileGroovy') {
    // Groovy only needs the declared dependencies
    // (and not longer the output of compileJava)
    classpath = sourceSets.main.compileClasspath
}
tasks.named('compileJava') {
    // Java also depends on the result of Groovy compilation
    // (which automatically makes it depend of compileGroovy)
    classpath += files(sourceSets.main.groovy.classesDirectory)
}
```

build.gradle.kts

```
tasks.named<AbstractCompile>("compileGroovy") {
    // Groovy only needs the declared dependencies
    // (and not longer the output of compileJava)
    classpath = sourceSets.main.get().compileClasspath
}
tasks.named<AbstractCompile>("compileJava") {
    // Java also depends on the result of Groovy compilation
    // (which automatically makes it depend of compileGroovy)
    classpath += files(sourceSets.main.get().groovy.classesDirectory)
}
```

1. By setting the `compileGroovy` classpath to be only `sourceSets.main.compileClasspath`, we effectively remove the previous dependency on `compileJava` that was declared by having the classpath also take into consideration `sourceSets.main.java.classesDirectory`
2. By adding `sourceSets.main.groovy.classesDirectory` to the `compileJava classpath`, we effectively declare a dependency on the `compileGroovy` task

All of this is possible through the use of directory properties.

Extra language support

Beyond core Gradle, there are other great plugins for more JVM languages!

Testing in Java & JVM projects

Testing on the JVM is a rich subject matter. There are many different testing libraries and frameworks, as well as many different types of test. All need to be part of the build, whether they

are executed frequently or infrequently. This chapter is dedicated to explaining how Gradle handles differing requirements between and within builds, with significant coverage of how it integrates with the two most common testing frameworks: JUnit and TestNG.

It explains:

- Ways to control how the tests are run (Test execution)
- How to select specific tests to run (Test filtering)
- What test reports are generated and how to influence the process (Test reporting)
- How Gradle finds tests to run (Test detection)
- How to make use of the major frameworks' mechanisms for grouping tests together (Test grouping)

But first, let's look at the basics of JVM testing in Gradle.

NOTE	A new configuration DSL for modeling test execution phases is available via the incubating JVM Test Suite plugin.

The basics

All JVM testing revolves around a single task type: Test. This runs a collection of test cases using any supported test library — JUnit, JUnit Platform or TestNG — and collates the results. You can then turn those results into a report via an instance of the TestReport task type.

In order to operate, the `Test` task type requires just two pieces of information:

- Where to find the compiled test classes (property: Test.getTestClassesDirs())
- The execution classpath, which should include the classes under test as well as the test library that you're using (property: Test.getClasspath())

When you're using a JVM language plugin — such as the Java Plugin — you will automatically get the following:

- A dedicated `test` source set for unit tests
- A `test` task of type `Test` that runs those unit tests

The JVM language plugins use the source set to configure the task with the appropriate execution classpath and the directory containing the compiled test classes. In addition, they attach the `test` task to the `check` lifecycle task.

It's also worth bearing in mind that the `test` source set automatically creates corresponding dependency configurations — of which the most useful are `testImplementation` and `testRuntimeOnly` — that the plugins tie into the `test` task's classpath.

All you need to do in most cases is configure the appropriate compilation and runtime dependencies and add any necessary configuration to the `test` task. The following example shows a simple setup that uses JUnit Platform and changes the maximum heap size for the tests' JVM to 1 gigabyte:

Example 483. A basic configuration for the 'test' task

build.gradle

```
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.7.1'
}

test {
    useJUnitPlatform()

    maxHeapSize = '1G'

    testLogging {
        events "passed"
    }
}
```

build.gradle.kts

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:5.7.1")
}

tasks.test {
    useJUnitPlatform()

    maxHeapSize = "1G"

    testLogging {
        events("passed")
    }
}
```

The Test task has many generic configuration options as well as several framework-specific ones that you can find described in JUnitOptions, JUnitPlatformOptions and TestNGOptions. We cover a significant number of them in the rest of the chapter.

If you want to set up your own `Test` task with its own set of test classes, then the easiest approach is to create your own source set and `Test` task instance, as shown in Configuring integration tests.

Test execution

Gradle executes tests in a separate ('forked') JVM, isolated from the main build process. This prevents classpath pollution and excessive memory consumption for the build process. It also

allows you to run the tests with different JVM arguments than the build is using.

You can control how the test process is launched via several properties on the `Test` task, including the following:

`maxParallelForks` — **default: 1**

You can run your tests in parallel by setting this property to a value greater than 1. This may make your test suites complete faster, particularly if you run them on a multi-core CPU. When using parallel test execution, make sure your tests are properly isolated from one another. Tests that interact with the filesystem are particularly prone to conflict, causing intermittent test failures.

Your tests can distinguish between parallel test processes by using the value of the `org.gradle.test.worker` property, which is unique for each process. You can use this for anything you want, but it's particularly useful for filenames and other resource identifiers to prevent the kind of conflict we just mentioned.

`forkEvery` — **default: 0 (no maximum)**

This property specifies the maximum number of test classes that Gradle should run on a test process before its disposed of and a fresh one created. This is mainly used as a way to manage leaky tests or frameworks that have static state that can't be cleared or reset between tests.

Warning: a low value (other than 0) can severely hurt the performance of the tests

`ignoreFailures` — **default: false**

If this property is `true`, Gradle will continue with the project's build once the tests have completed, even if some of them have failed. Note that, by default, the `Test` task always executes every test that it detects, irrespective of this setting.

`failFast` — **(since Gradle 4.6) default: false**

Set this to `true` if you want the build to fail and finish as soon as one of your tests fails. This can save a lot of time when you have a long-running test suite and is particularly useful when running the build on continuous integration servers. When a build fails before all tests have run, the test reports only include the results of the tests that have completed, successfully or not.

You can also enable this behavior by using the `--fail-fast` command line option.

`testLogging` — **default: *not set***

This property represents a set of options that control which test events are logged and at what level. You can also configure other logging behavior via this property. See TestLoggingContainer for more detail.

See Test for details on all the available configuration options.

NOTE

The test process can exit unexpectedly if configured incorrectly. For instance, if the Java executable does not exist or an invalid JVM argument is provided, the test process will fail to start. Similarly, if a test makes programmatic changes to the test process, this can also cause unexpected failures.

For example, issues may occur if a `SecurityManager` is modified in a test because Gradle's internal messaging depends on reflection and socket communication, which may be disrupted if the permissions on the security manager change. In this particular case, you should restore the original `SecurityManager` after the test so that the gradle test worker process can continue to function.

Test filtering

It's a common requirement to run subsets of a test suite, such as when you're fixing a bug or developing a new test case. Gradle provides two mechanisms to do this:

- Filtering (the preferred option)
- Test inclusion/exclusion

Filtering supersedes the inclusion/exclusion mechanism, but you may still come across the latter in the wild.

With Gradle's test filtering you can select tests to run based on:

- A fully-qualified class name or fully qualified method name, e.g. `org.gradle.SomeTest`, `org.gradle.SomeTest.someMethod`
- A simple class name or method name if the pattern starts with an upper-case letter, e.g. `SomeTest`, `SomeTest.someMethod` (since Gradle 4.7)
- '*' wildcard matching

You can enable filtering either in the build script or via the `--tests` command-line option. Here's an example of some filters that are applied every time the build runs:

Example 484. Filtering tests in the build script

build.gradle

```
test {
    filter {
        //include specific method in any of the tests
        includeTestsMatching "*UiCheck"

        //include all tests from package
        includeTestsMatching "org.gradle.internal.*"

        //include all integration tests
        includeTestsMatching "*IntegTest"
    }
}
```

build.gradle.kts

```
tasks.test {
    filter {
        //include specific method in any of the tests
        includeTestsMatching("*UiCheck")

        //include all tests from package
        includeTestsMatching("org.gradle.internal.*")

        //include all integration tests
        includeTestsMatching("*IntegTest")
    }
}
```

For more details and examples of declaring filters in the build script, please see the TestFilter reference.

The command-line option is especially useful to execute a single test method. When you use `--tests`, be aware that the inclusions declared in the build script are still honored. It is also possible to supply multiple `--tests` options, all of whose patterns will take effect. The following sections have several examples of using the command-line option.

NOTE	Not all test frameworks play well with filtering. Some advanced, synthetic tests may not be fully compatible. However, the vast majority of tests and use cases work perfectly well with Gradle's filtering mechanism.

The following two sections look at the specific cases of simple class/method names and fully-

qualified names.

Simple name pattern

Since 4.7, Gradle has treated a pattern starting with an uppercase letter as a simple class name, or a class name + method name. For example, the following command lines run either all or exactly one of the tests in the `SomeTestClass` test case, regardless of what package it's in:

```
# Executes all tests in SomeTestClass
gradle test --tests SomeTestClass

# Executes a single specified test in SomeTestClass
gradle test --tests SomeTestClass.someSpecificMethod

gradle test --tests SomeTestClass.*someMethod*
```

Fully-qualified name pattern

Prior to 4.7 or if the pattern doesn't start with an uppercase letter, Gradle treats the pattern as fully-qualified. So if you want to use the test class name irrespective of its package, you would use `--tests *.SomeTestClass`. Here are some more examples:

```
# specific class
gradle test --tests org.gradle.SomeTestClass

# specific class and method
gradle test --tests org.gradle.SomeTestClass.someSpecificMethod

# method name containing spaces
gradle test --tests "org.gradle.SomeTestClass.some method containing spaces"

# all classes at specific package (recursively)
gradle test --tests 'all.in.specific.package*'

# specific method at specific package (recursively)
gradle test --tests 'all.in.specific.package*.someSpecificMethod'

gradle test --tests '*IntegTest'

gradle test --tests '*IntegTest*ui*'

gradle test --tests '*ParameterizedTest.foo*'

# the second iteration of a parameterized test
gradle test --tests '*ParameterizedTest.*[2]'
```

Note that the wildcard '*' has no special understanding of the '.' package separator. It's purely text based. So `--tests *.SomeTestClass` will match any package, regardless of its 'depth'.

You can also combine filters defined at the command line with continuous build to re-execute a subset of tests immediately after every change to a production or test source file. The following executes all tests in the 'com.mypackage.foo' package or subpackages whenever a change triggers the tests to run:

```
gradle test --continuous --tests "com.mypackage.foo.*"
```

Test reporting

The `Test` task generates the following results by default:

- An HTML test report
- XML test results in a format compatible with the Ant JUnit report task — one that is supported by many other tools, such as CI servers
- An efficient binary format of the results used by the `Test` task to generate the other formats

In most cases, you'll work with the standard HTML report, which automatically includes the results from *all* your `Test` tasks, even the ones you explicitly add to the build yourself. For example, if you add a `Test` task for integration tests, the report will include the results of both the unit tests and the integration tests if both tasks are run.

NOTE	To aggregate test results across multiple subprojects, see the Test Report Aggregation Plugin.

Unlike with many of the testing configuration options, there are several project-level convention properties that affect the test reports. For example, you can change the destination of the test results and reports like so:

Example 485. Changing the default test report and results directories

build.gradle

```
reporting.baseDir = "my-reports"
testResultsDirName = "$buildDir/my-test-results"

tasks.register('showDirs') {
    doLast {
        logger.quiet(rootDir.toPath().relativize(project.reportsDir.toPath()
).toString())
        logger.quiet(rootDir.toPath().relativize(project.testResultsDir
.toPath()).toString())
    }
}
```

build.gradle.kts

```
reporting.baseDir = file("my-reports")
project.setProperty("testResultsDirName", "$buildDir/my-test-results")

tasks.register("showDirs") {
    doLast {

logger.quiet(rootDir.toPath().relativize((project.property("reportsDir") as
File).toPath()).toString())

logger.quiet(rootDir.toPath().relativize((project.property("testResultsDir")
as File).toPath()).toString())
    }
}
```

Output of `gradle -q showDirs`

```
> gradle -q showDirs
my-reports
build/my-test-results
```

Follow the link to the convention properties for more details.

There is also a standalone TestReport task type that you can use to generate a custom HTML test report. All it requires are a value for `destinationDir` and the test results you want included in the report. Here is a sample which generates a combined report for the unit tests from all subprojects:

Example 486. Creating a unit test report for subprojects

buildSrc/src/main/groovy/myproject.java-conventions.gradle

```
plugins {
    id 'java'
}

// Disable the test report for the individual test task
test {
    reports.html.required = false
}

// Share the test report data to be aggregated for the whole project
configurations {
    binaryTestResultsElements {
        canBeResolved = false
        canBeConsumed = true
        attributes {
            attribute(Category.CATEGORY_ATTRIBUTE, objects.named(Category,
Category.DOCUMENTATION))
            attribute(DocsType.DOCS_TYPE_ATTRIBUTE, objects.named(DocsType,
'test-report-data'))
        }
        outgoing.artifact(test.binaryResultsDirectory)
    }
}
```

build.gradle

```
// A resolvable configuration to collect test reports data
configurations {
    testReportData {
        canBeResolved = true
        canBeConsumed = false
        attributes {
            attribute(Category.CATEGORY_ATTRIBUTE, objects.named(Category,
Category.DOCUMENTATION))
            attribute(DocsType.DOCS_TYPE_ATTRIBUTE, objects.named(DocsType,
'test-report-data'))
        }
    }
}

dependencies {
    testReportData project(':core')
    testReportData project(':util')
}

tasks.register('testReport', TestReport) {
    destinationDirectory = reporting.baseDirectory.dir('allTests')
    // Use test results from testReportData configuration
    testResults.from(configurations.testReportData)
}
```

buildSrc/src/main/kotlin/myproject.java-conventions.gradle.kts

```
plugins {
    id("java")
}

// Disable the test report for the individual test task
tasks.named<Test>("test") {
    reports.html.required.set(false)
}

// Share the test report data to be aggregated for the whole project
configurations.create("binaryTestResultsElements") {
    isCanBeResolved = false
    isCanBeConsumed = true
    attributes {
        attribute(Category.CATEGORY_ATTRIBUTE,
objects.named(Category.DOCUMENTATION))
        attribute(DocsType.DOCS_TYPE_ATTRIBUTE, objects.named("test-report-
data"))
    }
    outgoing.artifact(tasks.test.map { task ->
task.getBinaryResultsDirectory().get() })
}
```

build.gradle.kts

```
val testReportData by configurations.creating {
    isCanBeResolved = true
    isCanBeConsumed = false
    attributes {
        attribute(Category.CATEGORY_ATTRIBUTE,
objects.named(Category.DOCUMENTATION))
        attribute(DocsType.DOCS_TYPE_ATTRIBUTE, objects.named("test-report-
data"))
    }
}

dependencies {
    testReportData(project(":core"))
    testReportData(project(":util"))
}

tasks.register<TestReport>("testReport") {
    destinationDirectory.set(reporting.baseDirectory.dir("allTests"))
    // Use test results from testReportData configuration
    testResults.from(testReportData)
}
```

In this example, we use a convention plugin `myproject.java-conventions` to expose the test results from a project to Gradle's variant aware dependency management engine.

The plugin declares a consumable `binaryTestResultsElements` configuration that represents the binary test results of the `test` task. In the aggregation project's build file, we declare the `testReportData` configuration and depend on all of the projects that we want to aggregate the results from. Gradle will automatically select the binary test result variant from each of the subprojects instead of the project's jar file. Lastly, we add a `testReport` task that aggregates the test results from the `testResultsDirs` property, which contains all of the binary test results resolved from the `testReportData` configuration.

You should note that the `TestReport` type combines the results from multiple test tasks and needs to aggregate the results of individual test classes. This means that if a given test class is executed by multiple test tasks, then the test report will include executions of that class, but it can be hard to distinguish individual executions of that class and their output.

Communicating test results to CI servers and other tools via XML files

The Test tasks creates XML files describing the test results, in the "JUnit XML" pseudo standard. It is common for CI servers and other tooling to observe test results via these XML files.

By default, the files are written to `$buildDir/test-results/$testTaskName` with a file per test class. The location can be changed for all test tasks of a project, or individually per test task.

Example 487. Changing JUnit XML results location for all test tasks

build.gradle

```
testResultsDirName = "$buildDir/junit-xml"
```

build.gradle.kts

```
project.setProperty("testResultsDirName", "$buildDir/junit-xml")
```

With the above configuration, the XML files will be written to `$buildDir/junit-xml/$testTaskName`.

Example 488. Changing JUnit XML results location for a particular test task

build.gradle

```
test {
    reports {
        junitXml.outputLocation.set(layout.buildDirectory.dir("test-junit-
xml"))
    }
}
```

build.gradle.kts

```
tasks.test {
    reports {
        junitXml.outputLocation.set(layout.buildDirectory.dir("test-junit-
xml"))
    }
}
```

With the above configuration, the XML files for the `test` task will be written to `$buildDir/test-results/test-junit-xml`. The location of the XML files for other test tasks will be unchanged.

Configuration options

The content of the XML files can also be configured to convey the results differently, by configuring the JUnitXmlReport options.

Example 489. Configuring how the results are conveyed

build.gradle

```
test {
    reports {
        junitXml {
            outputPerTestCase = true // defaults to false
            mergeReruns = true // defaults to false
        }
    }
}
```

build.gradle.kts

```
tasks.test {
    reports {
        junitXml.apply {
            isOutputPerTestCase = true // defaults to false
            mergeReruns.set(true) // defaults to false
        }
    }
}
```

outputPerTestCase

The `outputPerTestCase` option, when enabled, associates any output logging generated during a test case to that test case in the results. When disabled (the default) output is associated with the test class as whole and not the individual test cases (e.g. test methods) that produced the logging output. Most modern tools that observe JUnit XML files support the “output per test case” format.

If you are using the XML files to communicate test results, it is recommended to enable this option as it provides more useful reporting.

mergeReruns

When `mergeReruns` is enabled, if a test fails but is then retried and succeeds, its failures will be recorded as `<flakyFailure>` instead of `<failure>`, within one `<testcase>`. This is effectively the reporting produced by the surefire plugin of Apache Maven™ when enabling reruns. If your CI server understands this format, it will indicate that the test was flaky. If it does not, it will indicate that the test succeeded as it will ignore the `<flakyFailure>` information. If the test does not succeed (i.e. it fails for every retry), it will be indicated as having failed whether your tool understands this format or not.

When `mergeReruns` is disabled (the default), each execution of a test will be listed as a separate test

case.

If you are using build scans or Gradle Enterprise, flaky tests will be detected regardless of this setting.

Enabling this option is especially useful when using a CI tool that uses the XML test results to determine build failure instead of relying on Gradle's determination of whether the build failed or not, and you wish to not consider the build failed if all failed tests passed when retried. This is the case for the Jenkins CI server and its JUnit plugin. With `mergeReruns` enabled, tests that pass-on-retry will no longer cause this Jenkins plugin to consider the build to have failed. However, failed test executions will be omitted from the Jenkins test result visualizations as it does not consider `<flakyFailure>` information. The separate Flaky Test Handler Jenkins plugin can be used in addition to the JUnit Jenkins plugin to have such "flaky failures" also be visualized.

Tests are grouped and merged based on their reported name. When using any kind of test parameterization that affects the reported test name, or any other kind of mechanism that produces a potentially dynamic test name, care should be taken to ensure that the test name is stable and does not unnecessarily change.

Enabling the `mergeReruns` option does not add any retry/rerun functionality to test execution. Rerunning can be enabled by the test execution framework (e.g. JUnit's @RepeatedTest), or via the separate Test Retry Gradle plugin.

Test detection

By default, Gradle will run all tests that it detects, which it does by inspecting the compiled test classes. This detection uses different criteria depending on the test framework used.

For *JUnit*, Gradle scans for both JUnit 3 and 4 test classes. A class is considered to be a JUnit test if it:

- Ultimately inherits from `TestCase` or `GroovyTestCase`
- Is annotated with `@RunWith`
- Contains a method annotated with `@Test` or a super class does

For *TestNG*, Gradle scans for methods annotated with `@Test`.

Note that abstract classes are not executed. In addition, be aware that Gradle scans up the inheritance tree into jar files on the test classpath. So if those JARs contain test classes, they will also be run.

If you don't want to use test class detection, you can disable it by setting the `scanForTestClasses` property on Test to `false`. When you do that, the test task uses only the `includes` and `excludes` properties to find test classes.

If `scanForTestClasses` is false and no include or exclude patterns are specified, Gradle defaults to running any class that matches the patterns `**/*Tests.class` and `**/*Test.class`, excluding those that match `**/Abstract*.class`.

NOTE	With JUnit Platform, only `includes` and `excludes` are used to filter test classes — `scanForTestClasses` has no effect.

Test grouping

JUnit, JUnit Platform and TestNG allow sophisticated groupings of test methods.

NOTE	This section applies to grouping individual test classes or methods within a collection of tests that serve the same testing purpose (unit tests, integration tests, acceptance tests, etc.). For dividing test classes based upon their purpose, see the incubating JVM Test Suite plugin.

JUnit 4.8 introduced the concept of categories for grouping JUnit 4 tests classes and methods.[8] Test.useJUnit(org.gradle.api.Action) allows you to specify the JUnit categories you want to include and exclude. For example, the following configuration includes tests in `CategoryA` and excludes those in `CategoryB` for the `test` task:

Example 490. JUnit Categories

build.gradle

```
test {
    useJUnit {
        includeCategories 'org.gradle.junit.CategoryA'
        excludeCategories 'org.gradle.junit.CategoryB'
    }
}
```

build.gradle.kts

```
tasks.test {
    useJUnit {
        includeCategories("org.gradle.junit.CategoryA")
        excludeCategories("org.gradle.junit.CategoryB")
    }
}
```

JUnit Platform introduced tagging to replace categories. You can specify the included/excluded tags via Test.useJUnitPlatform(org.gradle.api.Action), as follows:

Example 491. JUnit Platform Tags

build.gradle

```
test {
    useJUnitPlatform {
        includeTags 'fast'
        excludeTags 'slow'
    }
}
```

build.gradle.kts

```
tasks.test {
    useJUnitPlatform {
        includeTags("fast")
        excludeTags("slow")
    }
}
```

The TestNG framework uses the concept of test groups for a similar effect.[9] You can configure which test groups to include or exclude during the test execution via the Test.useTestNG(org.gradle.api.Action) setting, as seen here:

Example 492. Grouping TestNG tests

build.gradle

```
test {
    useTestNG {
        excludeGroups 'integrationTests'
        includeGroups 'unitTests'
    }
}
```

build.gradle.kts

```
tasks.named<Test>("test") {
    useTestNG {
        val options = this as TestNGOptions
        options.excludeGroups("integrationTests")
        options.includeGroups("unitTests")
    }
}
```

Using JUnit 5

JUnit 5 is the latest version of the well-known JUnit test framework. Unlike its predecessor, JUnit 5 is modularized and composed of several modules:

```
JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage
```

The JUnit Platform serves as a foundation for launching testing frameworks on the JVM. JUnit Jupiter is the combination of the new programming model and extension model for writing tests and extensions in JUnit 5. JUnit Vintage provides a `TestEngine` for running JUnit 3 and JUnit 4 based tests on the platform.

The following code enables JUnit Platform support in `build.gradle`:

Example 493. Enabling JUnit Platform to run your tests

build.gradle

```
test {
    useJUnitPlatform()
}
```

build.gradle.kts

```
tasks.named<Test>("test") {
    useJUnitPlatform()
}
```

See Test.useJUnitPlatform() for more details.

Compiling and executing JUnit Jupiter tests

To enable JUnit Jupiter support in Gradle, all you need to do is add the following dependency:

Example 494. JUnit Jupiter dependencies

build.gradle

```
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.7.1'
}
```

build.gradle.kts

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:5.7.1")
}
```

You can then put your test cases into *src/test/java* as normal and execute them with `gradle test`.

Executing legacy tests with JUnit Vintage

If you want to run JUnit 3/4 tests on JUnit Platform, or even mix them with Jupiter tests, you should add extra JUnit Vintage Engine dependencies:

Example 495. JUnit Vintage dependencies

build.gradle

```
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.7.1'
    testCompileOnly 'junit:junit:4.13'
    testRuntimeOnly 'org.junit.vintage:junit-vintage-engine'
}
```

build.gradle.kts

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:5.7.1")
    testCompileOnly("junit:junit:4.13")
    testRuntimeOnly("org.junit.vintage:junit-vintage-engine")
}
```

In this way, you can use `gradle test` to test JUnit 3/4 tests on JUnit Platform, without the need to rewrite them.

Filtering test engine

JUnit Platform allows you to use different test engines. JUnit currently provides two `TestEngine` implementations out of the box: junit-jupiter-engine and junit-vintage-engine. You can also write and plug in your own `TestEngine` implementation as documented here.

By default, all test engines on the test runtime classpath will be used. To control specific test engine implementations explicitly, you can add the following setting to your build script:

Example 496. Filter specific engines

build.gradle

```
test {
    useJUnitPlatform {
        includeEngines 'junit-vintage'
        // excludeEngines 'junit-jupiter'
    }
}
```

build.gradle.kts

```
tasks.test {
    useJUnitPlatform {
        includeEngines("junit-vintage")
        // excludeEngines("junit-jupiter")
    }
}
```

Test execution order in TestNG

TestNG allows explicit control of the execution order of tests when you use a *testng.xml* file. Without such a file — or an equivalent one configured by TestNGOptions.getSuiteXmlBuilder() — you can't specify the test execution order. However, what you *can* do is control whether all aspects of a test — including its associated `@BeforeXXX` and `@AfterXXX` methods, such as those annotated with `@Before/AfterClass` and `@Before/AfterMethod` — are executed before the next test starts. You do this by setting the TestNGOptions.getPreserveOrder() property to `true`. If you set it to `false`, you may encounter scenarios in which the execution order is something like: `TestA.doBeforeClass()` → `TestB.doBeforeClass()` → `TestA` tests.

While preserving the order of tests is the default behavior when directly working with *testng.xml* files, the TestNG API that is used by Gradle's TestNG integration executes tests in unpredictable order by default.[10] The ability to preserve test execution order was introduced with TestNG version 5.14.5. Setting the `preserveOrder` property to `true` for an older TestNG version will cause the build to fail.

Example 497. Preserving order of TestNG tests

build.gradle

```
test {
    useTestNG {
        preserveOrder true
    }
}
```

build.gradle.kts

```
tasks.test {
    useTestNG {
        preserveOrder = true
    }
}
```

The `groupByInstance` property controls whether tests should be grouped by instance rather than by class. The TestNG documentation explains the difference in more detail, but essentially, if you have a test method `A()` that depends on `B()`, grouping by instance ensures that each A-B pairing, e.g. `B(1)`-`A(1)`, is executed before the next pairing. With group by class, all `B()` methods are run and then all `A()` ones.

Note that you typically only have more than one instance of a test if you’re using a data provider to parameterize it. Also, grouping tests by instances was introduced with TestNG version 6.1. Setting the `groupByInstances` property to `true` for an older TestNG version will cause the build to fail.

Example 498. Grouping TestNG tests by instances

build.gradle

```
test {
    useTestNG {
        groupByInstances = true
    }
}
```

build.gradle.kts

```
tasks.test {
    useTestNG {
        groupByInstances = true
    }
}
```

TestNG parameterized methods and reporting

TestNG supports parameterizing test methods, allowing a particular test method to be executed multiple times with different inputs. Gradle includes the parameter values in its reporting of the test method execution.

Given a parameterized test method named `aTestMethod` that takes two parameters, it will be reported with the name `aTestMethod(toStringValueOfParam1, toStringValueOfParam2)`. This makes it easy to identify the parameter values for a particular iteration.

Configuring integration tests

A common requirement for projects is to incorporate integration tests in one form or another. Their aim is to verify that the various parts of the project are working together properly. This often means that they require special execution setup and dependencies compared to unit tests.

The simplest way to add integration tests to your build is by leveraging the incubating JVM Test Suite plugin. If an incubating solution is not something for you, here are the steps you need to take in your build:

1. Create a new source set for them
2. Add the dependencies you need to the appropriate configurations for that source set
3. Configure the compilation and runtime classpaths for that source set
4. Create a task to run the integration tests

You may also need to perform some additional configuration depending on what form the

integration tests take. We will discuss those as we go.

Let's start with a practical example that implements the first three steps in a build script, centered around a new source set `intTest`:

Example 499. Setting up working integration tests

build.gradle

```
sourceSets {
    intTest {
        compileClasspath += sourceSets.main.output
        runtimeClasspath += sourceSets.main.output
    }
}

configurations {
    intTestImplementation.extendsFrom implementation
    intTestRuntimeOnly.extendsFrom runtimeOnly
}

dependencies {
    intTestImplementation 'org.junit.jupiter:junit-jupiter:5.7.1'
}
```

build.gradle.kts

```
sourceSets {
    create("intTest") {
        compileClasspath += sourceSets.main.get().output
        runtimeClasspath += sourceSets.main.get().output
    }
}

val intTestImplementation by configurations.getting {
    extendsFrom(configurations.implementation.get())
}

configurations["intTestRuntimeOnly"].extendsFrom(configurations.runtimeOnly.g
et())

dependencies {
    intTestImplementation("org.junit.jupiter:junit-jupiter:5.7.1")
}
```

This will set up a new source set called `intTest` that automatically creates:

- `intTestImplementation`, `intTestCompileOnly`, `intTestRuntimeOnly` configurations (and a few others that are less commonly needed)
- A `compileIntTestJava` task that will compile all the source files under *src/intTest/java*

NOTE	If you are working with the IntelliJ IDE, you may wish to flag the directories in these additional source sets as containing test source rather than production source as explained in the Idea Plugin documentation.

The example also does the following, not all of which you may need for your specific integration tests:

- Adds the production classes from the `main` source set to the compilation and runtime classpaths of the integration tests — `sourceSets.main.output` is a file collection of all the directories containing compiled production classes and resources
- Makes the `intTestImplementation` configuration extend from `implementation`, which means that all the declared dependencies of the production code also become dependencies of the integration tests
- Does the same for the `intTestRuntimeOnly` configuration

In most cases, you want your integration tests to have access to the classes under test, which is why we ensure that those are included on the compilation and runtime classpaths in this example. But some types of test interact with the production code in a different way. For example, you may have tests that run your application as an executable and verify the output. In the case of web applications, the tests may interact with your application via HTTP. Since the tests don't need direct access to the classes under test in such cases, you don't need to add the production classes to the test classpath.

Another common step is to attach all the unit test dependencies to the integration tests as well — via `intTestImplementation.extendsFrom testImplementation` — but that only makes sense if the integration tests require *all* or nearly all the same dependencies that the unit tests have.

There are a couple of other facets of the example you should take note of:

- `+=` allows you to append paths and collections of paths to `compileClasspath` and `runtimeClasspath` instead of overwriting them
- If you want to use the convention-based configurations, such as `intTestImplementation`, you *must* declare the dependencies *after* the new source set

Creating and configuring a source set automatically sets up the compilation stage, but it does nothing with respect to running the integration tests. So the last piece of the puzzle is a custom test task that uses the information from the new source set to configure its runtime classpath and the test classes:

Example 500. Defining a working integration test task

build.gradle

```
tasks.register('integrationTest', Test) {
    description = 'Runs integration tests.'
    group = 'verification'

    testClassesDirs = sourceSets.intTest.output.classesDirs
    classpath = sourceSets.intTest.runtimeClasspath
    shouldRunAfter test

    useJUnitPlatform()

    testLogging {
        events "passed"
    }
}

check.dependsOn integrationTest
```

build.gradle.kts

```
val integrationTest = task<Test>("integrationTest") {
    description = "Runs integration tests."
    group = "verification"

    testClassesDirs = sourceSets["intTest"].output.classesDirs
    classpath = sourceSets["intTest"].runtimeClasspath
    shouldRunAfter("test")

    useJUnitPlatform()

    testLogging {
        events("passed")
    }
}

tasks.check { dependsOn(integrationTest) }
```

Again, we're accessing a source set to get the relevant information, i.e. where the compiled test classes are — the `testClassesDirs` property — and what needs to be on the classpath when running them — `classpath`.

Users commonly want to run integration tests after the unit tests, because they are often slower to

run and you want the build to fail early on the unit tests rather than later on the integration tests. That's why the above example adds a `shouldRunAfter()` declaration. This is preferred over `mustRunAfter()` so that Gradle has more flexibility in executing the build in parallel.

For information on how to determine code coverage for tests in additional source sets, see the JaCoCo Plugin and the JaCoCo Report Aggregation Plugin chapters.

Testing Java Modules

If you are developing Java Modules, everything described in this chapter still applies and any of the supported test frameworks can be used. However, there are some things to consider depending on whether you need module information to be available, and module boundaries to be enforced, during test execution. In this context, the terms *whitebox testing* (module boundaries are deactivated or relaxed) and *blackbox testing* (module boundaries are in place) are often used. Whitebox testing is used/needed for unit testing and blackbox testing fits functional or integration test requirements.

Sample: Java Modules multi-project with integration tests

Whitebox unit test execution on the classpath

The simplest setup to write unit tests for functions or classes in modules is to *not* use module specifics during test execution. For this, you just need to write tests the same way you would write them for normal libraries. If you don't have a `module-info.java` file in your test source set (`src/test/java`) this source set will be considered as traditional Java library during compilation and test runtime. This means, all dependencies, including Jars with module information, are put on the classpath. The advantage is that all internal classes of your (or other) modules are then accessible directly in tests. This may be a totally valid setup for unit testing, where we do not care about the larger module structure, but only about testing single functions.

NOTE	If you are using Eclipse: By default, Eclipse also runs unit tests as modules using module patching (see below). In an imported Gradle project, unit testing a module with the Eclipse test runner might fail. You then need to manually adjust the classpath/module path in the test run configuration or delegate test execution to Gradle. This only concerns the test execution. Unit test compilation and development works fine in Eclipse.

Blackbox integration testing

For integration tests, you have the option to define the test set itself as additional module. You do this similar to how you turn your main sources into a module: by adding a `module-info.java` file to the corresponding source set (e.g. `integrationTests/java/module-info.java`).

You can find a full example that includes blackbox integration tests here.

NOTE	In Eclipse, compiling multiple modules in one project is currently not support. Therefore the integration test (blackbox) setup described here only works in Eclipse if the tests are moved to a separate subproject.

Whitebox test execution with module patching

Another approach for whitebox testing is to stay in the module world by *patching* the tests into the module under test. This way, module boundaries stay in place, but the tests themselves become part of the module under test and can then access the module's internals.

For which uses cases this is relevant and how this is best done is a topic of discussion. There is no general best approach at the moment. Thus, there is no special support for this in Gradle right now.

You can however, setup module patching for tests like this:

- Add a `module-info.java` to your test source set that is a copy of the main `module-info.java` with additional dependencies needed for testing (e.g. `requires org.junit.jupiter.api`).
- Configure both the `testCompileJava` and `test` tasks with arguments to patch the main classes with the test classes as shown below.

Example 501. Patch module for testing using command line arguments

build.gradle

```
def moduleName = "org.gradle.sample"
def patchArgs = ["--patch-module", "$moduleName=${tasks.compileJava
.destinationDirectory.asFile.get().path}"]
tasks.named('compileTestJava') {
    options.compilerArgs += patchArgs
}
tasks.named('test') {
    jvmArgs += patchArgs
}
```

build.gradle.kts

```
val moduleName = "org.gradle.sample"
val patchArgs = listOf("--patch-module",
"$moduleName=${tasks.compileJava.get().destinationDirectory.asFile.get().path
}")
tasks.compileTestJava {
    options.compilerArgs.addAll(patchArgs)
}
tasks.test {
    jvmArgs(patchArgs)
}
```

NOTE If custom arguments are used for patching, these are not picked up by Eclipse and IDEA. You will most likely see invalid compilation errors in the IDE.

Skipping the tests

If you want to skip the tests when running a build, you have a few options. You can either do it via command line arguments or in the build script. To do it on the command line, you can use the `-x` or `--exclude-task` option like so:

```
gradle build -x test
```

This excludes the `test` task and any other task that it *exclusively* depends on, i.e. no other task depends on the same task. Those tasks will not be marked "SKIPPED" by Gradle, but will simply not appear in the list of tasks executed.

Skipping a test via the build script can be done a few ways. One common approach is to make test execution conditional via the Task.onlyIf(String, org.gradle.api.specs.Spec) method. The following sample skips the `test` task if the project has a property called `mySkipTests`:

Example 502. Skipping the unit tests based on a project property

build.gradle

```
test.onlyIf("mySkipTests property is not set") { !project.hasProperty(
'mySkipTests') }
```

build.gradle.kts

```
tasks.test { onlyIf("mySkipTests property is not set") {
!project.hasProperty("mySkipTests") } }
```

In this case, Gradle will mark the skipped tests as "SKIPPED" rather than exclude them from the build.

Forcing tests to run

In well-defined builds, you can rely on Gradle to only run tests if the tests themselves or the production code change. However, you may encounter situations where the tests rely on a third-party service or something else that might change but can't be modeled in the build.

You can always use the `--rerun` built-in task option to force a task to rerun.

```
gradle test --rerun
```

Alternatively, if build caching is not enabled, you can also force tests to run by cleaning the output of the relevant `Test` task — say `test` — and running the tests again, like so:

```
gradle cleanTest test
```

`cleanTest` is based on a task rule provided by the Base Plugin. You can use it for *any* task.

Debugging when running tests

On the few occasions that you want to debug your code while the tests are running, it can be helpful if you can attach a debugger at that point. You can either set the Test.getDebug() property to `true` or use the `--debug-jvm` command line option.

When debugging for tests is enabled, Gradle will start the test process suspended and listening on port 5005.

You can also enable debugging in the DSL, where you can also configure other properties:

```
test {
    debugOptions {
        enabled = true
        host = 'localhost'
        port = 4455
        server = true
        suspend = true
    }
}
```

With this configuration the test JVM will behave just like when passing the `--debug-jvm` argument but it will listen on port 4455.

To debug the test process remotely via network, the `host` needs to be set to the machine's IP address or `"*"` (listen on all interfaces).

Using test fixtures

Producing and using test fixtures within a single project

Test fixtures are commonly used to setup the code under test, or provide utilities aimed at facilitating the tests of a component. Java projects can enable test fixtures support by applying the `java-test-fixtures` plugin, in addition to the `java` or `java-library` plugins:

Example 503. Applying the Java test fixtures plugin

lib/build.gradle

```
plugins {
    // A Java Library
    id 'java-library'
    // which produces test fixtures
    id 'java-test-fixtures'
    // and is published
    id 'maven-publish'
}
```

lib/build.gradle.kts

```
plugins {
    // A Java Library
    `java-library`
    // which produces test fixtures
    `java-test-fixtures`
    // and is published
    `maven-publish`
}
```

This will automatically create a `testFixtures` source set, in which you can write your test fixtures. Test fixtures are configured so that:

- they can see the *main* source set classes
- *test* sources can see the *test fixtures* classes

For example for this main class:

src/main/java/com/acme/Person.java

```
public class Person {
    private final String firstName;
    private final String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    // ...
```

A test fixture can be written in `src/testFixtures/java`:

src/testFixtures/java/com/acme/Simpsons.java

```
public class Simpsons {
    private static final Person HOMER = new Person("Homer", "Simpson");
    private static final Person MARGE = new Person("Marjorie", "Simpson");
    private static final Person BART = new Person("Bartholomew", "Simpson");
    private static final Person LISA = new Person("Elisabeth Marie", "Simpson");
    private static final Person MAGGIE = new Person("Margaret Eve", "Simpson");
    private static final List<Person> FAMILY = new ArrayList<Person>() {{
        add(HOMER);
        add(MARGE);
        add(BART);
        add(LISA);
        add(MAGGIE);
    }};

    public static Person homer() { return HOMER; }

    public static Person marge() { return MARGE; }

    public static Person bart() { return BART; }

    public static Person lisa() { return LISA; }

    public static Person maggie() { return MAGGIE; }

    // ...
```

Declaring dependencies of test fixtures

Similarly to the Java Library Plugin, test fixtures expose an API and an implementation configuration:

Example 504. Declaring test fixture dependencies

lib/build.gradle

```
dependencies {
    testImplementation 'junit:junit:4.13'

    // API dependencies are visible to consumers when building
    testFixturesApi 'org.apache.commons:commons-lang3:3.9'

    // Implementation dependencies are not leaked to consumers when building
    testFixturesImplementation 'org.apache.commons:commons-text:1.6'
}
```

lib/build.gradle.kts

```
dependencies {
    testImplementation("junit:junit:4.13")

    // API dependencies are visible to consumers when building
    testFixturesApi("org.apache.commons:commons-lang3:3.9")

    // Implementation dependencies are not leaked to consumers when building
    testFixturesImplementation("org.apache.commons:commons-text:1.6")
}
```

It's worth noticing that if a dependency is an *implementation* dependency of test fixtures, then *when compiling tests that depend on those test fixtures*, the implementation dependencies will *not leak* into the compile classpath. This results in improved separation of concerns and better compile avoidance.

Consuming test fixtures of another project

Test fixtures are not limited to a single project. It is often the case that a dependent project tests also needs the test fixtures of the dependency. This can be achieved very easily using the `testFixtures` keyword:

Example 505. Adding a dependency on test fixtures of another project

build.gradle

```
dependencies {
    implementation(project(":lib"))

    testImplementation 'junit:junit:4.13'
    testImplementation(testFixtures(project(":lib")))
}
```

build.gradle.kts

```
dependencies {
    implementation(project(":lib"))

    testImplementation("junit:junit:4.13")
    testImplementation(testFixtures(project(":lib")))
}
```

Publishing test fixtures

One of the advantages of using the `java-test-fixtures` plugin is that test fixtures are published. By convention, test fixtures will be published with an artifact having the `test-fixtures` classifier. For both Maven and Ivy, an artifact with that classifier is simply published alongside the regular artifacts. However, if you use the `maven-publish` or `ivy-publish` plugin, test fixtures are published as additional variants in Gradle Module Metadata and you can directly depend on test fixtures of external libraries in another Gradle project:

Example 506. Adding a dependency on test fixtures of an external library

build.gradle

```
dependencies {
    // Adds a dependency on the test fixtures of Gson, however this
    // project doesn't publish such a thing
    functionalTest testFixtures("com.google.code.gson:gson:2.8.5")
}
```

build.gradle.kts

```
dependencies {
    // Adds a dependency on the test fixtures of Gson, however this
    // project doesn't publish such a thing
    functionalTest(testFixtures("com.google.code.gson:gson:2.8.5"))
}
```

It's worth noting that if the external project is *not* publishing Gradle Module Metadata, then resolution will fail with an error indicating that such a variant cannot be found:

Output of `gradle dependencyInsight --configuration functionalTestClasspath --dependency gson`

```
> gradle dependencyInsight --configuration functionalTestClasspath --dependency gson

> Task :dependencyInsight
com.google.code.gson:gson:2.8.5 FAILED
   Failures:
      - Could not resolve com.google.code.gson:gson:2.8.5.
          - Unable to find a variant of com.google.code.gson:gson:2.8.5 providing the
requested capability com.google.code.gson:gson-test-fixtures:
               - Variant compile provides com.google.code.gson:gson:2.8.5
               - Variant runtime provides com.google.code.gson:gson:2.8.5
               - Variant sources provides com.google.code.gson:gson:2.8.5
               - Variant javadoc provides com.google.code.gson:gson:2.8.5
               - Variant platform-compile provides com.google.code.gson:gson-derived-
platform:2.8.5
               - Variant platform-runtime provides com.google.code.gson:gson-derived-
platform:2.8.5
               - Variant enforced-platform-compile provides com.google.code.gson:gson-
derived-enforced-platform:2.8.5
               - Variant enforced-platform-runtime provides com.google.code.gson:gson-
derived-enforced-platform:2.8.5

com.google.code.gson:gson:2.8.5 FAILED
\--- functionalTestClasspath

A web-based, searchable dependency report is available by adding the --scan option.

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

The error message mentions the missing `com.google.code.gson:gson-test-fixtures` capability, which is indeed not defined for this library. That's because by convention, for projects that use the `java-test-fixtures` plugin, Gradle automatically creates test fixtures variants with a capability whose name is the name of the main component, with the appendix `-test-fixtures`.

NOTE If you publish your library and use test fixtures, but do not want to publish the fixtures, you can deactivate publishing of the *test fixtures variants* as shown below.

Example 507. Disable publishing of test fixtures variants

build.gradle

```
components.java.withVariantsFromConfiguration(configurations.testFixturesApiE
lements) { skip() }
components.java.withVariantsFromConfiguration(configurations.testFixturesRunt
imeElements) { skip() }
```

build.gradle.kts

```
val javaComponent = components["java"] as AdhocComponentWithVariants
javaComponent.withVariantsFromConfiguration(configurations["testFixturesApiEl
ements"]) { skip() }
javaComponent.withVariantsFromConfiguration(configurations["testFixturesRunti
meElements"]) { skip() }
```

Managing Dependencies of JVM Projects

This chapter explains how to apply basic dependency management concepts to JVM-based projects. For a detailed introduction to dependency management, see dependency management in Gradle.

Dissecting a typical build script

Let's have a look at a very simple build script for a JVM-based project. It applies the Java Library plugin which automatically introduces a standard project layout, provides tasks for performing typical work and adequate support for dependency management.

Example 508. Dependency declarations for a JVM-based project

build.gradle

```
plugins {
    id 'java-library'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.hibernate:hibernate-core:3.6.7.Final'
    api 'com.google.guava:guava:23.0'
    testImplementation 'junit:junit:4.+'
}
```

build.gradle.kts

```
plugins {
    `java-library`
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.hibernate:hibernate-core:3.6.7.Final")
    api("com.google.guava:guava:23.0")
    testImplementation("junit:junit:4.+")
}
```

The Project.dependencies{} code block declares that Hibernate core 3.6.7.Final is required to compile the project's production source code. It also states that junit >= 4.0 is required to compile the project's tests. All dependencies are supposed to be looked up in the Maven Central repository as defined by Project.repositories{}. The following sections explain each aspect in more detail.

Declaring module dependencies

There are various types of dependencies that you can declare. One such type is a *module dependency*. A module dependency represents a dependency on a module with a specific version built outside the current build. Modules are usually stored in a repository, such as Maven Central, a corporate Maven or Ivy repository, or a directory in the local file system.

To define an module dependency, you add it to a dependency configuration:

Example 509. Definition of a module dependency

build.gradle

```
dependencies {
    implementation 'org.hibernate:hibernate-core:3.6.7.Final'
}
```

build.gradle.kts

```
dependencies {
    implementation("org.hibernate:hibernate-core:3.6.7.Final")
}
```

To find out more about defining dependencies, have a look at Declaring Dependencies.

Using dependency configurations

A Configuration is a named set of dependencies and artifacts. There are three main purposes for a *configuration*:

Declaring dependencies

A plugin uses configurations to make it easy for build authors to declare what other subprojects or external artifacts are needed for various purposes during the execution of tasks defined by the plugin. For example a plugin may need the Spring web framework dependency to compile the source code.

Resolving dependencies

A plugin uses configurations to find (and possibly download) inputs to the tasks it defines. For example Gradle needs to download Spring web framework JAR files from Maven Central.

Exposing artifacts for consumption

A plugin uses configurations to define what *artifacts* it generates for other projects to consume. For example the project would like to publish its compiled source code packaged in the JAR file to an in-house Artifactory repository.

With those three purposes in mind, let's take a look at a few of the standard configurations defined by the Java Library Plugin.

implementation

The dependencies required to compile the production source of the project which *are not* part of the API exposed by the project. For example the project uses Hibernate for its internal

persistence layer implementation.

api

The dependencies required to compile the production source of the project which *are* part of the API exposed by the project. For example the project uses Guava and exposes public interfaces with Guava classes in their method signatures.

testImplementation

The dependencies required to compile and run the test source of the project. For example the project decided to write test code with the test framework JUnit.

Various plugins add further standard configurations. You can also define your own custom configurations in your build via Project.configurations{}. See What are dependency configurations for the details of defining and customizing dependency configurations.

Declaring common Java repositories

How does Gradle know where to find the files for external dependencies? Gradle looks for them in a *repository*. A repository is a collection of modules, organized by `group`, `name` and `version`. Gradle understands different repository types, such as Maven and Ivy, and supports various ways of accessing the repository via HTTP or other protocols.

By default, Gradle does not define any repositories. You need to define at least one with the help of Project.repositories{} before you can use module dependencies. One option is use the Maven Central repository:

Example 510. Usage of Maven central repository

build.gradle

```
repositories {
    mavenCentral()
}
```

build.gradle.kts

```
repositories {
    mavenCentral()
}
```

You can also have repositories on the local file system. This works for both Maven and Ivy repositories.

Example 511. Usage of a local Ivy directory

build.gradle

```
repositories {
    ivy {
        // URL can refer to a local directory
        url "../local-repo"
    }
}
```

build.gradle.kts

```
repositories {
    ivy {
        // URL can refer to a local directory
        url = uri("../local-repo")
    }
}
```

A project can have multiple repositories. Gradle will look for a dependency in each repository in the order they are specified, stopping at the first repository that contains the requested module.

To find out more about defining repositories, have a look at Declaring Repositories.

Publishing artifacts

To learn more about publishing artifacts, have a look at publishing plugins.

[7] In fact, any artifact added to the `archives` configuration will be built by `assemble`

[8] The JUnit wiki contains a detailed description on how to work with JUnit categories: https://github.com/junit-team/junit/wiki/Categories.

[9] The TestNG documentation contains more details about test groups: http://testng.org/doc/documentation-main.html#test-groups.

[10] The TestNG documentation contains more details about test ordering when working with `testng.xml` files: http://testng.org/doc/documentation-main.html#testng-xml.

C++ & Other Native Projects

Building C++ projects

Gradle uses a convention-over-configuration approach to building native projects. If you are coming from another native build system, these concepts may be unfamiliar at first, but they serve a purpose to simplify build script authoring.

We will look at C++ projects in detail in this chapter, but most of the topics will apply to other supported native languages as well. If you don't have much experience with building native projects with Gradle, take a look at the C++ tutorials for step-by-step instructions on how to build various types of basic C++ projects as well as some common use cases.

The C++ plugins covered in this chapter were introduced in 2018 and we recommend users to use those plugins over the older Native plugins that you may find references to.

Introduction

The simplest build script for a C++ project applies the C++ application plugin or the C++ library plugin and optionally sets the project version:

Example 512. Applying the C++ Plugin

build.gradle

```
plugins {
    id 'cpp-application' // or 'cpp-library'
}

version = '1.2.1'
```

build.gradle.kts

```
plugins {
    `cpp-application` // or `cpp-library`
}

version = "1.2.1"
```

By applying either of the C++ plugins, you get a whole host of features:

- `compileDebugCpp` and `compileReleaseCpp` tasks that compiles the C++ source files under *src/main/cpp* for the well-known debug and release build types, respectively.

- `linkDebug` and `linkRelease` tasks that link the compiled C++ object files into an executable for applications or shared library for libraries with shared linkage for the debug and release build types.
- `createDebug` and `createRelease` tasks that assemble the compiled C++ object files into a static library for libraries with static linkage for the debug and release build types.

For any non-trivial C++ project, you'll probably have some file dependencies and additional configuration specific to *your* project.

The C++ plugins also integrates the above tasks into the standard lifecycle tasks. The task that produces the development binary is attached to `assemble`. By default, the development binary is the debug variant.

The rest of the chapter explains the different ways to customize the build to your requirements when building libraries and applications.

Introducing build variants

Native projects can typically produce several different binaries, such as debug or release ones, or ones that target particular platforms and processor architectures. Gradle manages this through the concepts of *dimensions* and *variants*.

A dimension is simply a category, where each category is orthogonal to the rest. For example, the "build type" dimension is a category that includes debug and release. The "architecture" dimension covers processor architectures like x86-64 and PowerPC.

A variant is a combination of values for these dimensions, consisting of exactly one value for each dimension. You might have a "debug x86-64" or a "release PowerPC" variant.

Gradle has built-in support for several dimensions and several values within each dimension. You can find a list of them in the native plugin reference chapter.

Declaring your source files

Gradle's C++ support uses a `ConfigurableFileCollection` directly from the application or library script block to configure the set of sources to compile.

Libraries make a distinction between private (implementation details) and public (exported to consumer) headers.

You can also configure sources for each binary build for those cases where sources are compiled only on certain target machines.

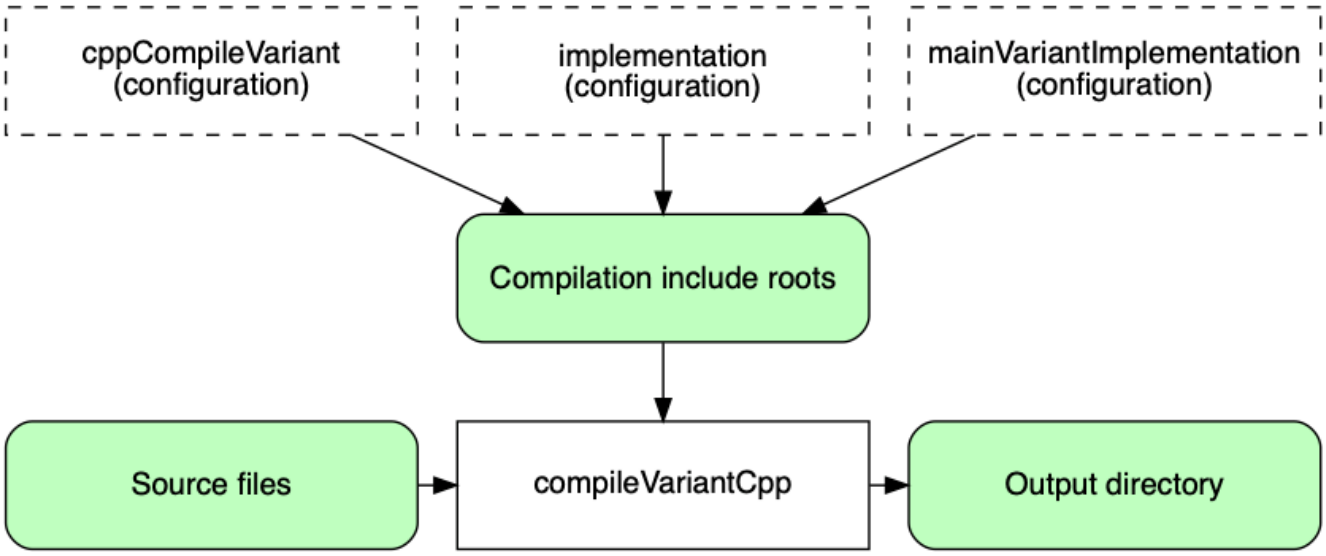


Figure 27. Sources and C++ compilation

Test sources are configured on each test suite script block. See Testing C++ projects chapter.

Managing your dependencies

The vast majority of projects rely on other projects, so managing your project's dependencies is an important part of building any project. Dependency management is a big topic, so we will only focus on the basics for C++ projects here. If you'd like to dive into the details, check out the introduction to dependency management.

Gradle provides support for consuming pre-built binaries from Maven repositories published by Gradle [11].

We will cover how to add dependencies between projects within a multi-build project.

Specifying dependencies for your C++ project requires two pieces of information:

- Identifying information for the dependency (project path, Maven GAV)
- What it's needed for, e.g. compilation, linking, runtime or all of the above.

This information is specified in a `dependencies {}` block of the C++ `application` or `library` script block. For example, to tell Gradle that your project requires library `common` to compile and link your production code, you can use the following fragment:

Example 513. Declaring dependencies

build.gradle

```
application {
    dependencies {
        implementation project(':common')
    }
}
```

build.gradle.kts

```
application {
    dependencies {
        implementation(project(":common"))
    }
}
```

The Gradle terminology for the three elements is as follows:

- *Configuration* (ex: `implementation`) - a named collection of dependencies, grouped together for a specific goal such as compiling or linking a module
- *Project reference* (ex: `project(':common')`) - the project referenced by the specified path

You can find a more comprehensive glossary of dependency management terms here.

As far as configurations go, the main ones of interest are:

- `implementation` - used for compilation, linking and runtime
- `cppCompileVariant` - for dependencies that are necessary to compile your production code but shouldn't be part of the linking or runtime process
- `nativeLinkVariant` - for dependencies that are necessary to link your code but shouldn't be part of the compilation or runtime process
- `nativeRuntimeVariant` - for dependencies that are necessary to run your component but shouldn't be part of the compilation or linking process

You can learn more about these and how they relate to one another in the native plugin reference chapter.

Be aware that the C++ Library Plugin creates an additional configuration — `api` — for dependencies that are required for compiling and linking both the module and any modules that depend on it.

We have only scratched the surface here, so we recommend that you read the dedicated dependency management chapters once you're comfortable with the basics of building C++ projects

with Gradle.

Some common scenarios that require further reading include:

- Defining a custom Maven-compatible repository
- Declaring dependencies with changing (e.g. SNAPSHOT) and dynamic (range) versions
- Declaring a sibling project as a dependency
- Controlling transitive dependencies and their versions
- Testing your fixes to 3rd-party dependency via composite builds (a better alternative to publishing to and consuming from Maven Local)

You'll discover that Gradle has a rich API for working with dependencies — one that takes time to master, but is straightforward to use for common scenarios.

Compiling and linking your code

Compiling both your code can be trivially easy if you follow the conventions:

1. Put your source code under the *src/main/cpp* directory
2. Declare your compile dependencies in the `implementation` configurations (see the previous section)
3. Run the `assemble` task

We recommend that you follow these conventions wherever possible, but you don't have to.

There are several options for customization, as you'll see next.

NOTE	All CppCompile tasks are incremental and cacheable.

Supported tool chain

Gradle offers the ability to execute the same build using different tool chains. When you build a native binary, Gradle will attempt to locate a tool chain installed on your machine that can build the binary. Gradle select the first tool chain that can build for the target operating system and architecture. In the future, Gradle will consider source and ABI compatibility when selecting a tool chain.

Gradle has general support for the three major tool chains on major operating system: Clang [12], GCC [13] and Visual C++ [14] (Windows-only). GCC and Clang installed using Macports and Homebrew have been reported to work fine, but this isn't tested continuously.

Windows

To build on Windows, install a compatible version of Visual Studio. The C++ plugins will discover the Visual Studio installations and select the latest version. There is no need to mess around with environment variables or batch scripts. This works fine from a Cygwin shell or the Windows command-line.

Alternatively, you can install Cygwin or MinGW with GCC. Clang is currently not supported.

macOS

To build on macOS, you should install Xcode. The C++ plugins will discover the Xcode installation using the system PATH.

The C++ plugins also work with GCC and Clang installed with Macports or Homebrew [15]. To use one of the Macports or Homebrew, you will need to add Macports/Homebrew to the system PATH.

Linux

To build on Linux, install a compatible version of GCC or Clang. The C++ plugins will discover GCC or Clang using the system PATH.

Customizing file and directory locations

Imagine you have a legacy library project that uses an *src* directory for the production code and private headers and *include* directory for exported headers. The conventional directory structure won't work, so you need to tell Gradle where to find the source and header files. You do that via the `application` or `library` script block.

Each component script block, as well as each binary, defines where it's source code resides. You can override the convention values by using the following syntax:

Example 514. Setting C++ source set

build.gradle

```
library {
    source.from file('src')
    privateHeaders.from file('src')
    publicHeaders.from file('include')
}
```

build.gradle.kts

```
library {
    source.from(file("src"))
    privateHeaders.from(file("src"))
    publicHeaders.from(file("include"))
}
```

Now Gradle will only search directly in *src* for the source and private headers and in *include* for public headers.

Changing compiler and linker options

Most of the compiler and linker options are accessible through the corresponding task, such as `compileVariantCpp`, `linkVariant` and `createVariant`. These tasks are of type CppCompile, LinkSharedLibrary and CreateStaticLibrary respectively. Read the task reference for an up-to-date and comprehensive list of the options.

For example, if you want to change the warning level generated by the compiler for all variants, you can use this configuration:

Example 515. Setting C++ compiler options for all variants

build.gradle

```
tasks.withType(CppCompile).configureEach {
    // Define a preprocessor macro for every binary
    macros.put("NDEBUG", null)

    // Define a compiler options
    compilerArgs.add '-W3'

    // Define toolchain-specific compiler options
    compilerArgs.addAll toolChain.map { toolChain ->
        if (toolChain in [ Gcc, Clang ]) {
            return ['-O2', '-fno-access-control']
        } else if (toolChain in VisualCpp) {
            return ['/Zi']
        }
        return []
    }
}
```

build.gradle.kts

```
tasks.withType(CppCompile::class.java).configureEach {
    // Define a preprocessor macro for every binary
    macros.put("NDEBUG", null)

    // Define a compiler options
    compilerArgs.add("-W3")

    // Define toolchain-specific compiler options
    compilerArgs.addAll(toolChain.map { toolChain ->
        when (toolChain) {
            is Gcc, is Clang -> listOf("-O2", "-fno-access-control")
            is VisualCpp -> listOf("/Zi")
            else -> listOf()
        }
    })
}
```

It's also possible to find the instance for a specific variant through the `BinaryCollection` on the `application` or `library` script block:

Example 516. Setting C++ compiler options per variant

build.gradle

```
application {
    binaries.configureEach(CppStaticLibrary) {
        // Define a preprocessor macro for every binary
        compileTask.get().macros.put("NDEBUG", null)

        // Define a compiler options
        compileTask.get().compilerArgs.add '-W3'

        // Define toolchain-specific compiler options
        if (toolChain in [ Gcc, Clang ]) {
            compileTask.get().compilerArgs.addAll(['-O2', '-fno-access
-control'])
        } else if (toolChain in VisualCpp) {
            compileTask.get().compilerArgs.add('/Zi')
        }
    }
}
```

build.gradle.kts

```
application {
    binaries.configureEach(CppStaticLibrary::class.java) {
        // Define a preprocessor macro for every binary
        compileTask.get().macros.put("NDEBUG", null)

        // Define a compiler options
        compileTask.get().compilerArgs.add("-W3")

        // Define toolchain-specific compiler options
        when (toolChain) {
            is Gcc, is Clang ->
compileTask.get().compilerArgs.addAll(listOf("-O2", "-fno-access-control"))
            is VisualCpp -> compileTask.get().compilerArgs.add("/Zi")
        }
    }
}
```

Selecting target machines

By default, Gradle will attempt to create a C++ binary variant for the host operating system and architecture. It is possible to override this by specifying the set of `TargetMachine` on the `application`

or `library` script block:

Example 517. Setting target machines

build.gradle

```
application {
    targetMachines = [
        machines.linux.x86_64,
        machines.windows.x86, machines.windows.x86_64,
        machines.macOS.x86_64
    ]
}
```

build.gradle.kts

```
application {
    targetMachines.set(listOf(machines.windows.x86, machines.windows.x86_64,
machines.macOS.x86_64, machines.linux.x86_64))
}
```

Packaging and publishing

How you package and potentially publish your C++ project varies greatly in the native world. Gradle comes with defaults, but custom packaging can be implemented without any issues.

- Executable files are published directly to Maven repositories.
- Shared and static library files are published directly to Maven repositories along with a zip of the public headers.
- For applications, Gradle also supports installing and running the executable with all of its shared library dependencies in a known location.

Cleaning the build

The C++ Application and Library Plugins add a `clean` task to you project by using the base plugin. This task simply deletes everything in the `$buildDir` directory, hence why you should always put files generated by the build in there. The task is an instance of Delete and you can change what directory it deletes by setting its `dir` property.

Building C++ libraries

The unique aspect of library projects is that they are used (or "consumed") by other C++ projects. That means the dependency metadata published with the binaries and headers — in the form of Gradle Module Metadata — is crucial. In particular, consumers of your library should be able to

distinguish between two different types of dependencies: those that are only required to compile your library and those that are also required to compile the consumer.

Gradle manages this distinction via the C++ Library Plugin, which introduces an *api* configuration in addition to the *implementation* once covered in this chapter. If the types from a dependency appear as unresolved symbols of the static library or within the public headers then that dependency is exposed via your library's public API and should, therefore, be added to the *api* configuration. Otherwise, the dependency is an internal implementation detail and should be added to *implementation*.

If you're unsure of the difference between an API and implementation dependency, the C++ Library Plugin chapter has a detailed explanation. In addition, you can see a basic, practical example of building a C++ library in the corresponding sample.

Building C++ applications

See the C++ Application Plugin chapter for more details, but here's a quick summary of what you get:

- `install` create a directory containing everything needed to run it
- Shell and Windows Batch scripts to start the application

You can see a basic example of building a C++ application in the corresponding sample.

Testing in C++ projects

Testing in the native ecosystem takes many forms.

There are different testing libraries and frameworks, as well as many different types of test. All need to be part of the build, whether they are executed frequently or infrequently. This chapter is dedicated to explaining how Gradle handles differing requirements between and within builds, with significant coverage of how it integrates with the executable-based testing frameworks, such as Google Test.

Testing C++ projects in Gradle is fairly limited when compared to Testing in Java & JVM projects. In this chapter, we explain the ways to control how tests are run (Test execution).

But first, we look at the basics of native testing in Gradle.

The basics

All C++ testing revolves around a single task type: RunTestExecutable. This runs a single test executable built with any testing framework and asserts the execution was successful using the exit code of the executable. The test case results aren't collected and no reports are generated.

In order to operate, the RunTestExecutable task type requires just one piece of information:

- Where to find the built test executable (property: RunTestExecutable.getExecutable())

When you're using the C++ Unit Test Plugin you will automatically get the following:

- A dedicated unitTest extension for configuring test component and its variants
- A `run` task of type RunTestExecutable that runs the test executable

The test plugins configure the required pieces of information appropriately. In addition, they attach the `run` task to the `check` lifecycle task. It also create the `testImplementation` dependency configuration. Dependencies that are only needed for test compilation, linking and runtime may be added to this configuration. The `unitTest` script block behave similarly to a `application` or `library` script block.

The RunTestExecutable task has many configuration options. We cover a number of them in the rest of the chapter.

Test execution

Gradle executes tests in a separate ('forked') process.

You can control how the test process is launched via several properties on the RunTestExecutable task, including the following:

`ignoreFailures` - default: false

If this property is `true`, Gradle will continue with the project's build once the tests have completed, even if some of them have failed. Note that, by default, RunTestExecutable task type always executes every test that it detects, irrespective of this setting.

See RunTestExecutable for details on all the available configuration options.

Building Swift projects

Gradle uses a convention-over-configuration approach to building native projects. If you are coming from another native build system, these concepts may be unfamiliar at first, but they serve a purpose to simplify build script authoring.

We will look at Swift projects in detail in this chapter, but most of the topics will apply to other supported native languages as well.

Introduction

The simplest build script for a Swift project applies the Swift application plugin or the Swift library plugin and optionally sets the project version:

Example 518. Applying the Swift Plugin

build.gradle

```
plugins {
    id 'swift-application' // or 'swift-library'
}

version = '1.2.1'
```

build.gradle.kts

```
plugins {
    `swift-application` // or `swift-library`
}

version = "1.2.1"
```

By applying either of the Swift plugins, you get a whole host of features:

- `compileDebugSwift` and `compileReleaseSwift` tasks that compiles the Swift source files under *src/main/swift* for the well-known debug and release build types, respectively.
- `linkDebug` and `linkRelease` tasks that link the compiled Swift object files into an executable for applications or shared library for libraries with shared linkage for the debug and release build types.
- `createDebug` and `createRelease` tasks that assemble the compiled Swift object files into a static library for libraries with static linkage for the debug and release build types.

For any non-trivial Swift project, you'll probably have some file dependencies and additional configuration specific to *your* project.

The Swift plugins also integrates the above tasks into the standard lifecycle tasks. The task that produces the development binary is attached to `assemble`. By default, the development binary is the debug variant.

The rest of the chapter explains the different ways to customize the build to your requirements when building libraries and applications.

Introducing build variants

Native projects can typically produce several different binaries, such as debug or release ones, or ones that target particular platforms and processor architectures. Gradle manages this through the concepts of *dimensions* and *variants*.

A dimension is simply a category, where each category is orthogonal to the rest. For example, the "build type" dimension is a category that includes debug and release. The "architecture" dimension covers processor architectures like x86-64 and x86.

A variant is a combination of values for these dimensions, consisting of exactly one value for each dimension. You might have a "debug x86-64" or a "release x86" variant.

Gradle has built-in support for several dimensions and several values within each dimension. You can find a list of them in the native plugin reference chapter.

Declaring your source files

Gradle's Swift support uses a `ConfigurableFileCollection` directly from the application or library script block to configure the set of sources to compile.

Libraries make a distinction between private (implementation details) and public (exported to consumer) headers.

You can also configure sources for each binary build for those cases where sources are compiled only on certain target machines.

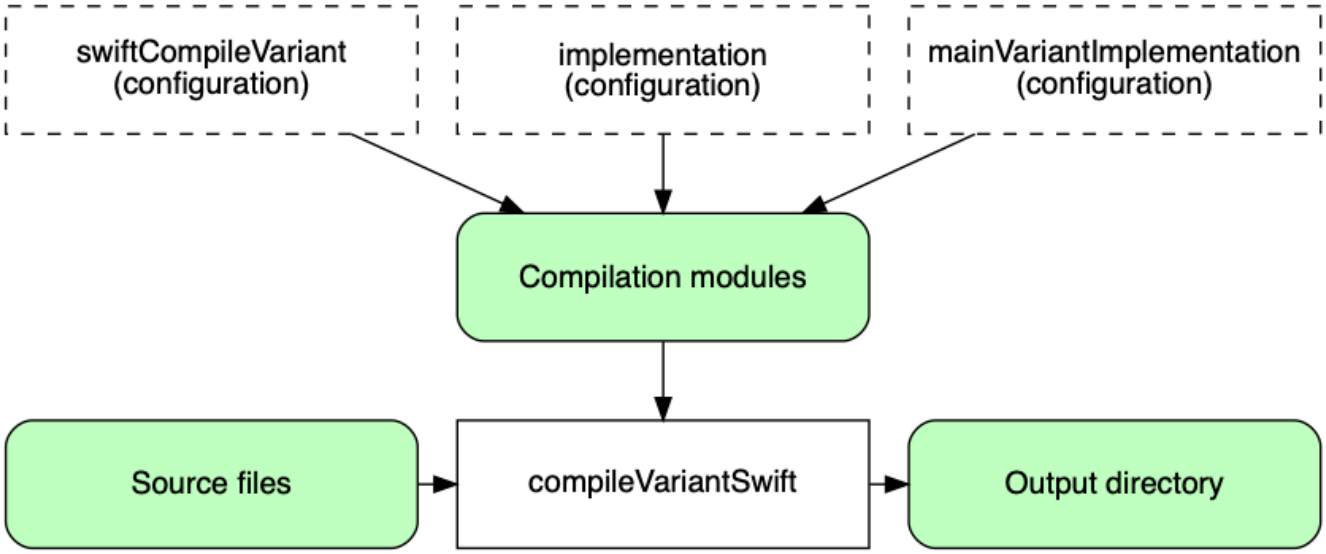


Figure 28. Sources and Swift compilation

Managing your dependencies

The vast majority of projects rely on other projects, so managing your project's dependencies is an important part of building any project. Dependency management is a big topic, so we will only focus on the basics for Swift projects here. If you'd like to dive into the details, check out the introduction to dependency management.

Gradle provides support for consuming pre-built binaries from Maven repositories published by Gradle [16].

We will cover how to add dependencies between projects within a multi-build project.

Specifying dependencies for your Swift project requires two pieces of information:

- Identifying information for the dependency (project path, Maven GAV)
- What it's needed for, e.g. compilation, linking, runtime or all of the above.

This information is specified in a `dependencies {}` block of the Swift `application` or `library` script block. For example, to tell Gradle that your project requires library `common` to compile and link your production code, you can use the following fragment:

Example 519. Declaring dependencies

build.gradle

```
application {
    dependencies {
        implementation project(':common')
    }
}
```

build.gradle.kts

```
application {
    dependencies {
        implementation(project(":common"))
    }
}
```

The Gradle terminology for the three elements is as follows:

- *Configuration* (ex: `implementation`) - a named collection of dependencies, grouped together for a specific goal such as compiling or linking a module
- *Project reference* (ex: `project(':common')`) - the project referenced by the specified path

You can find a more comprehensive glossary of dependency management terms here.

As far as configurations go, the main ones of interest are:

- `implementation` - used for compilation, linking and runtime
- `swiftCompileVariant` - for dependencies that are necessary to compile your production code but shouldn't be part of the linking or runtime process
- `nativeLinkVariant` - for dependencies that are necessary to link your code but shouldn't be part of the compilation or runtime process
- `nativeRuntimeVariant` - for dependencies that are necessary to run your component but shouldn't be part of the compilation or linking process

You can learn more about these and how they relate to one another in the native plugin reference

chapter.

Be aware that the Swift Library Plugin creates an additional configuration — `api` — for dependencies that are required for compiling and linking both the module and any modules that depend on it.

We have only scratched the surface here, so we recommend that you read the dedicated dependency management chapters once you're comfortable with the basics of building Swift projects with Gradle.

Some common scenarios that require further reading include:

- Defining a custom Maven-compatible repository
- Declaring dependencies with changing (e.g. SNAPSHOT) and dynamic (range) versions
- Declaring a sibling project as a dependency
- Controlling transitive dependencies and their versions
- Testing your fixes to 3rd-party dependency via composite builds (a better alternative to publishing to and consuming from Maven Local)

You'll discover that Gradle has a rich API for working with dependencies — one that takes time to master, but is straightforward to use for common scenarios.

Compiling and linking your code

Compiling both your code can be trivially easy if you follow the conventions:

1. Put your source code under the *src/main/swift* directory
2. Declare your compile dependencies in the `implementation` configurations (see the previous section)
3. Run the `assemble` task

We recommend that you follow these conventions wherever possible, but you don't have to.

There are several options for customization, as you'll see next.

NOTE	All SwiftCompile tasks are incremental and cacheable.

Supported tool chain

Gradle support the official Swift tool chain for macOS and Linux. When you build a native binary, Gradle will attempt to locate a tool chain installed on your machine that can build the binary. Gradle select the first tool chain that can build for the target operating system, architecture and Swift language support.

NOTE	For Linux users, Gradle will discover the tool chain using the system PATH.

Customizing file and directory locations

Imagine you are migrating a library project that follows the Swift Package Manager layout (e.g. `Sources/ModuleName_` directory for the production code). The conventional directory structure won't work, so you need to tell Gradle where to find the source files. You do that via the `application` or `library` script block.

Each component script block, as well as each binary, defines where it's source code resides. You can override the convention values by using the following syntax:

Example 520. Setting Swift source set

build.gradle

```
library {
    source.from file('src')
}
```

build.gradle.kts

```
extensions.configure<SwiftLibrary> {
    source.from(file("Sources/Common"))
}
```

Now Gradle will only search directly in *Sources/Common* for the source.

Changing compiler and linker options

Most of the compiler and linker options are accessible through the corresponding task, such as `compileVariantSwift`, `linkVariant` and `createVariant`. These tasks are of type SwiftCompile, LinkSharedLibrary and CreateStaticLibrary respectively. Read the task reference for an up-to-date and comprehensive list of the options.

For example, if you want to change the warning level generated by the compiler for all variants, you can use this configuration:

Example 521. Setting Swift compiler options for all variants

build.gradle

```
tasks.withType(SwiftCompile).configureEach {
    // Define a preprocessor macro for every binary
    macros.add("NDEBUG")

    // Define a compiler options
    compilerArgs.add '-O'
}
```

build.gradle.kts

```
tasks.withType(SwiftCompile::class.java).configureEach {
    // Define a preprocessor macro for every binary
    macros.add("NDEBUG")

    // Define a compiler options
    compilerArgs.add("-O")
}
```

It's also possible to find the instance for a specific variant through the `BinaryCollection` on the `application` or `library` script block:

Example 522. Setting Swift compiler options per variant

build.gradle

```
application {
    binaries.configureEach(SwiftStaticLibrary) {
        // Define a preprocessor macro for every binary
        compileTask.get().macros.add("NDEBUG")

        // Define a compiler options
        compileTask.get().compilerArgs.add '-O'
    }
}
```

build.gradle.kts

```
application {
    binaries.configureEach(SwiftStaticLibrary::class.java) {
        // Define a preprocessor macro for every binary
        compileTask.get().macros.add("NDEBUG")

        // Define a compiler options
        compileTask.get().compilerArgs.add("-O")
    }
}
```

Selecting target machines

By default, Gradle will attempt to create a Swift binary variant for the host operating system and architecture. It is possible to override this by specifying the set of `TargetMachine` on the `application` or `library` script block:

Example 523. Setting target machines

build.gradle

```
application {
    targetMachines = [
        machines.linux.x86_64,
        machines.macOS.x86_64
    ]
}
```

build.gradle.kts

```
application {
    targetMachines.set(listOf(machines.linux.x86_64, machines.macOS.x86_64))
}
```

Packaging and publishing

How you package and potentially publish your Swift project varies greatly in the native world. Gradle comes with defaults, but custom packaging can be implemented without any issues.

- Executable files are published directly to Maven repositories.
- Shared and static library files are published directly to Maven repositories along with a zip of the public headers.
- For applications, Gradle also supports installing and running the executable with all of its shared library dependencies in a known location.

Cleaning the build

The Swift Application and Library Plugins add a `clean` task to you project by using the base plugin. This task simply deletes everything in the `$buildDir` directory, hence why you should always put files generated by the build in there. The task is an instance of Delete and you can change what directory it deletes by setting its `dir` property.

Building Swift libraries

The unique aspect of library projects is that they are used (or "consumed") by other Swift projects. That means the dependency metadata published with the binaries and headers — in the form of Gradle Module Metadata — is crucial. In particular, consumers of your library should be able to distinguish between two different types of dependencies: those that are only required to compile your library and those that are also required to compile the consumer.

Gradle manages this distinction via the Swift Library Plugin, which introduces an *api* configuration

in addition to the *implementation* once covered in this chapter. If the types from a dependency appear as unresolved symbols of the static library or within the public headers then that dependency is exposed via your library's public API and should, therefore, be added to the *api* configuration. Otherwise, the dependency is an internal implementation detail and should be added to *implementation*.

If you're unsure of the difference between an API and implementation dependency, the Swift Library Plugin chapter has a detailed explanation. In addition, you can see a basic, practical example of building a Swift library in the corresponding sample.

Building Swift applications

See the Swift Application Plugin chapter for more details, but here's a quick summary of what you get:

- `install` create a directory containing everything needed to run it
- Shell and Windows Batch scripts to start the application

You can see a basic example of building a Swift application in the corresponding sample.

Testing in Swift projects

Testing in the native ecosystem is a rich subject matter. There are many different testing libraries and frameworks, as well as many different types of test. All need to be part of the build, whether they are executed frequently or infrequently. This chapter is dedicated to explaining how Gradle handles differing requirements between and within builds, with significant coverage of how it integrates with XCTest on both macOS and Linux.

It explains: - Ways to control how the tests are run (Test execution) - How to select specific tests to run (Test filtering) - What test reports are generated and how to influence the process (Test reporting) - How Gradle finds tests to run (Test detection)

But first, we look at the basics of native testing in Gradle.

The basics

Gradle supports deep integration with XCTest testing framework for the Swift language and revolves around the XCTest task type. This runs a collection of test cases using the Xcode XCTest on macOS or the open source Swift core library alternative on Linux and collates the results. You can then turn those results into a report via an instance of the TestReport task type.

In order to operate, the XCTest task type requires three pieces of information: - Where to find the built testable bundle (on macOS) or executable (on Linux) (property: XCTest.getTestInstalledDirectory()) - The run script for executing the bundle or executable (property: XCTest.getRunScriptFile()) - The working directory to execution the bundle or executable (property: XCTest.getWorkingDirectory())

When you're using the XCTest Plugin you will automatically get the following: - A dedicated `xctest` extension of type SwiftXCTestSuite for configuring test component and its variants - A `xcTest` task of

type XCTest that runs those unit tests - A testable bundle or executable linked with the main component's object files

The test plugins configure the required pieces of information appropriately. In addition, they attach the `xcTest` or `run` task to the `check` lifecycle task. It also create the `testImplementation` dependency configuration. Dependencies that are only needed for test compilation, linking and runtime may be added to this configuration. The `xctest` script block behave similarly to a `application` or `library` script block.

The XCTest task has many configuration options. We cover a significant number of them in the rest of the chapter.

Test execution

Gradle executes tests in a separate ('forked') process.

You can control how the test process is launched via several properties on the XCTest task, including the following:

`ignoreFailures` **- default: false**

If this property is `true`, Gradle will continue with the project's build once the tests have completed, even if some of them have failed. Note that, by default, both task type always executes every test that it detects, irrespective of this setting.

`testLogging` **- default: not set**

This property represents a set of options that control which test events are logged and at what level. You can also configure other logging behavior via this property. Set TestLoggingContainer for more detail.

See XCTest for details on all the available configuration options.

Test filtering

It's a common requirement to run subsets of a test suite, such as when you're fixing a bug or developing a new test case. Gradle provides filtering to do this. You can select tests to run based on:

- A simple class name or method name, e.g. `SomeTest`, `SomeTest.someMethod`
- '*' wildcard matching

You can enable filtering either in the build script or via the `--tests` command-line option. Here's an example of some filters that are applied every time the build runs:

Example 524. Filter tests on every build

build.gradle

```
xctest {
    binaries.configureEach {
        runTask.get().configure {
            // include all tests from test class
            filter.includeTestsMatching "SomeIntegTest.*" // or
`"Testing.SomeIntegTest.*"` on macOS
        }
    }
}
```

build.gradle.kts

```
xctest {
    binaries.configureEach {
        runTask.get().filter.includeTestsMatching("SomeIntegTest.*") // or
`"Testing.SomeIntegTest.*"` on macOS
    }
}
```

For more details and examples of declaring filters in the build script, please see the TestFilter reference.

The command-line option is especially useful to execute a single test method. It is also possible to supply multiple `--tests` options, all of whose patterns will take effect. The following sections have several examples of using command-line option.

NOTE	The test filtering only support XCTest compatible filters at the moment. It means the same filter will differ between macOS and Linux. On macOS, the bundle base name needs to be prepended to the filter, e.g. `TestBundle.SomeTest`, `TestBundle.SomeTest.someMethod` See the Simple name pattern section below for more information about valid filtering pattern.

The following section looks at the specific cases of simple class/method names.

Simple name pattern

Gradle support simple class name, or a class name + method name test filtering. For example, the following command lines run either all or exactly one of the tests in the `SomeTestClass` test case:

```
# Executes all tests in SomeTestClass
gradle xcTest --tests SomeTestClass
# or `gradle xcTest --tests TestBundle.SomeTestClass` on macOS

# Executes a single specified test in SomeTestClass
gradle xcTest --tests TestBundle.SomeTestClass.someSpecificMethod
# or `gradle xcTest --tests TestBundle.SomeTestClass.someSpecificMethod` on macOS
```

You can also combine filters defined at the command line with continuous build to re-execute a subset of tests immediately after every change to a production or test source file. The following executes all tests in the 'SomeTestClass' test class whenever a change triggers the tests to run:

```
gradle test --continuous --tests SomeTestClass
```

Test reporting

The XCTest task generates the following results by default:

- An HTML test report
- XML test results in a format compatible with the Ant JUnit report task - one that is supported by many other tools, such as CI servers
- An efficient binary format of the results used by the `XCTest` task to generate the other formats

In most cases, you'll work with the standard HTML report, which automatically includes the result from your `XCTest` tasks.

There is also a standalone TestReport task type that you can use to generate a custom HTML test report. All it requires are a value for `destinationDir` and the test results you want included in the report. Here is a sample which generates a combined report for the unit tests from all subprojects:

Example 525. Combine test reports from all subprojects

buildSrc/src/main/groovy/myproject.xctest-conventions.gradle

```
plugins {
    id 'xctest'
}

xctest {
    binaries.configureEach {
        runTask.get().configure {
            // Disable the test report for the individual test task
            reports.html.required = false
        }
    }
}

// Share the test report data to be aggregated for the whole project
configurations {
    binaryTestResultsElements {
        canBeResolved = false
        canBeConsumed = true
        attributes {
            attribute(Category.CATEGORY_ATTRIBUTE, objects.named(Category,
Category.DOCUMENTATION))
            attribute(DocsType.DOCS_TYPE_ATTRIBUTE, objects.named(DocsType,
'test-report-data'))
        }
        tasks.withType(XCTest).configureEach {
            outgoing.artifact(it.binaryResultsDirectory)
        }
    }
}
```

build.gradle

```
// A resolvable configuration to collect test reports data
plugins {
    id 'reporting-base'
}

configurations {
    testReportData {
        canBeResolved = true
        canBeConsumed = false
        attributes {
            attribute(Category.CATEGORY_ATTRIBUTE, objects.named(Category,
Category.DOCUMENTATION))
            attribute(DocsType.DOCS_TYPE_ATTRIBUTE, objects.named(DocsType,
'test-report-data'))
        }
    }
}

dependencies {
    testReportData project(':core')
    testReportData project(':util')
}

tasks.register('testReport', TestReport) {
    destinationDirectory = reporting.baseDirectory.dir('allTests')
    // Use test results from testReportData configuration
    testResults.from(configurations.testReportData)
}
```

buildSrc/src/main/kotlin/myproject.xctest-conventions.gradle.kts

```
plugins {
    id("xctest")
}

extensions.configure<SwiftXCTestSuite>() {
    binaries.configureEach {
        // Disable the test report for the individual test task
        runTask.get().reports.html.required.set(false)
    }
}

configurations.create("binaryTestResultsElements") {
    isCanBeResolved = false
    isCanBeConsumed = true
    attributes {
        attribute(Category.CATEGORY_ATTRIBUTE,
objects.named(Category.DOCUMENTATION))
        attribute(DocsType.DOCS_TYPE_ATTRIBUTE, objects.named("test-report-
data"))
    }
    tasks.withType<XCTest>() {
        outgoing.artifact(binaryResultsDirectory)
    }
}
```

build.gradle.kts

```
plugins {
    `reporting-base`
}

val testReportData by configurations.creating {
    isCanBeResolved = true
    isCanBeConsumed = false
    attributes {
        attribute(Category.CATEGORY_ATTRIBUTE,
objects.named(Category.DOCUMENTATION))
        attribute(DocsType.DOCS_TYPE_ATTRIBUTE, objects.named("test-report-
data"))
    }
}

dependencies {
    testReportData(project(":core"))
    testReportData(project(":util"))
}

tasks.register<TestReport>("testReport") {
    destinationDirectory.set(reporting.baseDirectory.dir("allTests"))
    // Use test results from testReportData configuration
    testResults.from(testReportData)
}
```

In this example, we use a convention plugin `myproject.xctest-conventions` to expose the test results from a project to Gradle's variant aware dependency management engine.

The plugin declares a consumable `binaryTestResultsElements` configuration that represents the binary test results of the `test` task. In the aggregation project's build file, we declare the `testReportData` configuration and depend on all of the projects that we want to aggregate the results from. Gradle will automatically select the binary test result variant from each of the subprojects instead of the project's jar file. Lastly, we add a `testReport` task that aggregates the test results from the `testResultsDirs` property, which contains all of the binary test results resolved from the `testReportData` configuration.

You should note that the TestReport type combines the results from multiple test tasks and needs to aggregate the results of individual test classes. This means that it a given test class is executed by multiple test tasks, then the test report will include executions of that class, but it can be hard to distinguish individual executions of that class and their output.

[11] Unfortunately, Conan and Nuget repositories aren't yet supported as core features

[12] Installed with Xcode on macOS

[13] Installed through Cygwin and MinGW for 32- and 64-bits architecture on Windows

[14] Installed with Visual Studio 2010 to 2019

[15] Macports and Homebrew installation of GCC and Clang is not officially supported

[16] Unfortunately, Cocoapods repositories aren't yet supported as core features

Native Projects using the Software Model

Building native software

CAUTION	The software model is being retired and the plugins mentioned in this chapter will eventually be deprecated and removed. We recommend new projects looking to build C++ applications and libraries use the newer replacement plugins.

The native software plugins add support for building native software components, such as executables or shared libraries, from code written in C++, C and other languages. While many excellent build tools exist for this space of software development, Gradle offers developers its trademark power and flexibility together with dependency management practices more traditionally found in the JVM development space.

The native software plugins make use of the Gradle software model.

Features

The native software plugins provide:

- Support for building native libraries and applications on Windows, Linux, macOS and other platforms.
- Support for several source languages.
- Support for building different variants of the same software, for different architectures, operating systems, or for any purpose.
- Incremental parallel compilation, precompiled headers.
- Dependency management between native software components.
- Unit test execution.
- Generate Visual studio solution and project files.
- Deep integration with various tool chain, including discovery of installed tool chains.

Supported languages

The following source languages are currently supported:

- C
- C++
- Objective-C
- Objective-C++
- Assembly
- Windows resources

Tool chain support

Gradle offers the ability to execute the same build using different tool chains. When you build a native binary, Gradle will attempt to locate a tool chain installed on your machine that can build the binary. You can fine tune exactly how this works, see Tool chain support for details.

The following tool chains are supported:

Operating System	Tool Chain	Notes
Linux	GCC	
Linux	Clang	
macOS	XCode	Uses the Clang tool chain bundled with XCode.
Windows	Visual C++	Windows XP and later, Visual C++ 2010/2012/2013/2015/2017/2019.
Windows	GCC with Cygwin 32 and Cygwin 64	Windows XP and later.
Windows	GCC with MinGW and MinGW64	Windows XP and later.

The following tool chains are unofficially supported. They generally work fine, but are not tested continuously:

Operating System	Tool Chain	Notes
macOS	GCC from Macports	
macOS	Clang from Macports	
UNIX-like	GCC	
UNIX-like	Clang	

Tool chain installation

NOTE	Note that if you are using GCC then you currently need to install support for C++, even if you are not building from C++ source. This restriction will be removed in a future Gradle version.

To build native software, you will need to have a compatible tool chain installed:

Windows

To build on Windows, install a compatible version of Visual Studio. The native plugins will discover the Visual Studio installations and select the latest version. There is no need to mess around with environment variables or batch scripts. This works fine from a Cygwin shell or the Windows command-line.

Alternatively, you can install Cygwin with GCC or MinGW. Clang is currently not supported.

macOS

To build on macOS, you should install XCode. The native plugins will discover the XCode installation using the system PATH.

The native plugins also work with GCC and Clang bundled with Macports. To use one of the Macports tool chains, you will need to make the tool chain the default using the `port select` command and add Macports to the system PATH.

Linux

To build on Linux, install a compatible version of GCC or Clang. The native plugins will discover GCC or Clang using the system PATH.

Native software model

The native software model builds on the base Gradle software model.

To build native software using Gradle, your project should define one or more *native components*. Each component represents either an executable or a library that Gradle should build. A project can define any number of components. Gradle does not define any components by default.

For each component, Gradle defines a *source set* for each language that the component can be built from. A source set is essentially just a set of source directories containing source files. For example, when you apply the `c` plugin and define a library called `helloworld`, Gradle will define, by default, a source set containing the C source files in the `src/helloworld/c` directory. It will use these source files to build the `helloworld` library. This is described in more detail below.

For each component, Gradle defines one or more *binaries* as output. To build a binary, Gradle will take the source files defined for the component, compile them as appropriate for the source language, and link the result into a binary file. For an executable component, Gradle can produce executable binary files. For a library component, Gradle can produce both static and shared library binary files. For example, when you define a library called `helloworld` and build on Linux, Gradle will, by default, produce `libhelloworld.so` and `libhelloworld.a` binaries.

In many cases, more than one binary can be produced for a component. These binaries may vary based on the tool chain used to build, the compiler/linker flags supplied, the dependencies provided, or additional source files provided. Each native binary produced for a component is referred to as a *variant*. Binary variants are discussed in detail below.

Parallel Compilation

Gradle uses the single build worker pool to concurrently compile and link native components, by default. No special configuration is required to enable concurrent building.

By default, the worker pool size is determined by the number of available processors on the build machine (as reported to the build JVM). To explicitly set the number of workers use the `--max-workers` command-line option or `org.gradle.workers.max` system property. There is generally no

need to change this setting from its default.

The build worker pool is shared across all build tasks. This means that when using parallel project execution, the maximum number of concurrent individual compilation operations does not increase. For example, if the build machine has 4 processing cores and 10 projects are compiling in parallel, Gradle will only use 4 total workers, not 40.

Building a library

To build either a static or shared native library, you define a library component in the `components` container. The following sample defines a library called `hello`:

Example: Defining a library component

build.gradle

```
model {
    components {
        hello(NativeLibrarySpec)
    }
}
```

A library component is represented using NativeLibrarySpec. Each library component can produce at least one shared library binary (SharedLibraryBinarySpec) and at least one static library binary (StaticLibraryBinarySpec).

Building an executable

To build a native executable, you define an executable component in the `components` container. The following sample defines an executable called `main`:

Example: Defining executable components

build.gradle

```
model {
    components {
        main(NativeExecutableSpec) {
            sources {
               c.lib library: "hello"
            }
        }
    }
}
```

An executable component is represented using NativeExecutableSpec. Each executable component can produce at least one executable binary (NativeExecutableBinarySpec).

For each component defined, Gradle adds a FunctionalSourceSet with the same name. Each of these

functional source sets will contain a language-specific source set for each of the languages supported by the project.

Assembling or building dependents

Sometimes, you may need to *assemble* (compile and link) or *build* (compile, link and test) a component or binary and its *dependents* (things that depend upon the component or binary). The native software model provides tasks that enable this capability. First, the *dependent components* report gives insight about the relationships between each component. Second, the *build and assemble dependents* tasks allow you to assemble or build a component and its dependents in one step.

In the following example, the build file defines `OpenSSL` as a dependency of `libUtil` and `libUtil` as a dependency of `LinuxApp` and `WindowsApp`. Test suites are treated similarly. Dependents can be thought of as reverse dependencies.

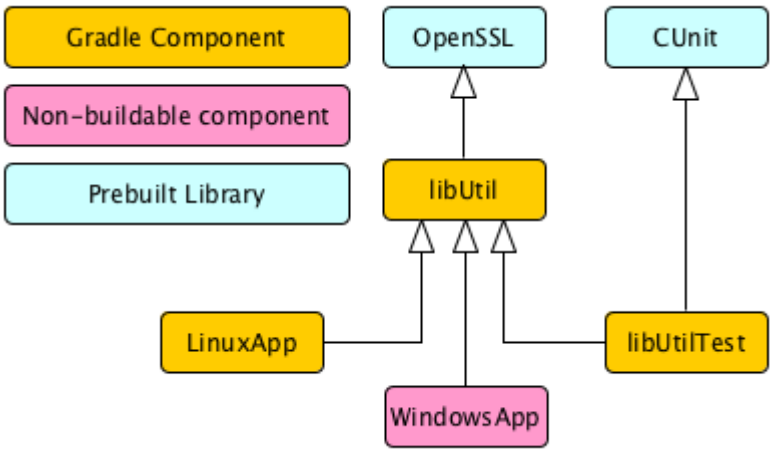


Figure 29. Dependent Components Example

NOTE	By following the dependencies backwards, you can see `LinuxApp` and `WindowsApp` are *dependents* of `libUtil`. When `libUtil` is changed, Gradle will need to recompile or relink `LinuxApp` and `WindowsApp`.

When you *assemble* dependents of a component, the component and all of its dependents are compiled and linked, including any test suite binaries. Gradle's up-to-date checks are used to only compile or link if something has changed. For instance, if you have changed source files in a way that do not affect the headers of your project, Gradle will be able to skip compilation for dependent components and only need to re-link with the new library. Tests are not run when assembling a component.

When you *build* dependents of a component, the component and all of its dependent binaries are compiled, linked *and checked*. Checking components means running any check task including executing any test suites, so tests *are* run when building a component.

In the following sections, we will demonstrate the usage of the `assembleDependents*`, `buildDependents*` and `dependentComponents` tasks with a sample build that contains a CUnit test suite. The build script for the sample is the following:

Example: Sample build

build.gradle

```
plugins {
    id 'c'
    id 'cunit-test-suite'
}

model {
    flavors {
        passing
        failing
    }
    platforms {
        x86 {
            if (operatingSystem.macOsX) {
                architecture "x64"
            } else {
                architecture "x86"
            }
        }
    }
    components {
        operators(NativeLibrarySpec) {
            targetPlatform "x86"
        }
    }
    testSuites {
        operatorsTest(CUnitTestSuiteSpec) {
            testing $.components.operators
        }
    }
}
```

Dependent components report

Gradle provides a report that you can run from the command-line that shows a graph of components in your project and components that depend upon them. The following is an example of running `gradle dependentComponents` on the sample project:

Example: Dependent components report

Output of `gradle dependentComponents`

```
> gradle dependentComponents

> Task :dependentComponents

------------------------------------------------------------
Root project 'cunit'
------------------------------------------------------------

operators - Components that depend on native library 'operators'
+--- operators:failingSharedLibrary
+--- operators:failingStaticLibrary
+--- operators:passingSharedLibrary
\--- operators:passingStaticLibrary

Some test suites were not shown, use --test-suites or --all to show them.

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

NOTE See DependentComponentsReport API documentation for more details.

By default, non-buildable binaries and test suites are hidden from the report. The `dependentComponents` task provides options that allow you to see all dependents by using the `--all` option:

Example: Dependent components report

Output of `gradle dependentComponents --all`

```
> gradle dependentComponents --all

> Task :dependentComponents

------------------------------------------------------------
Root project 'cunit'
------------------------------------------------------------

operators - Components that depend on native library 'operators'
+--- operators:failingSharedLibrary
+--- operators:failingStaticLibrary
|    \--- operatorsTest:failingCUnitExe (t)
+--- operators:passingSharedLibrary
\--- operators:passingStaticLibrary
     \--- operatorsTest:passingCUnitExe (t)

operatorsTest - Components that depend on Cunit test suite 'operatorsTest'
+--- operatorsTest:failingCUnitExe (t)
\--- operatorsTest:passingCUnitExe (t)

(t) - Test suite binary

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Here is the corresponding report for the `operators` component, showing dependents of all its binaries:

Example: Report of components that depends on the operators component

Output of `gradle dependentComponents --component operators`

```
> gradle dependentComponents --component operators

> Task :dependentComponents

------------------------------------------------------------
Root project 'cunit'
------------------------------------------------------------

operators - Components that depend on native library 'operators'
+--- operators:failingSharedLibrary
+--- operators:failingStaticLibrary
+--- operators:passingSharedLibrary
\--- operators:passingStaticLibrary

Some test suites were not shown, use --test-suites or --all to show them.

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Here is the corresponding report for the `operators` component, showing dependents of all its binaries, including test suites:

Example: Report of components that depends on the operators component, including test suites

Output of `gradle dependentComponents --test-suites --component operators`

```
> gradle dependentComponents --test-suites --component operators

> Task :dependentComponents

------------------------------------------------------------
Root project 'cunit'
------------------------------------------------------------

operators - Components that depend on native library 'operators'
+--- operators:failingSharedLibrary
+--- operators:failingStaticLibrary
|    \--- operatorsTest:failingCUnitExe (t)
+--- operators:passingSharedLibrary
\--- operators:passingStaticLibrary
     \--- operatorsTest:passingCUnitExe (t)

(t) - Test suite binary

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Assembling dependents

For each NativeBinarySpec, Gradle will create a task named `assembleDependents${component.name}${binary.variant}` that *assembles* (compile and link) the binary and all of its dependent binaries.

For each NativeComponentSpec, Gradle will create a task named `assembleDependents${component.name}` that *assembles* all the binaries of the component and all of their dependent binaries.

For example, to assemble the dependents of the "passing" flavor of the "static" library binary of the "operators" component, you would run the `assembleDependentsOperatorsPassingStaticLibrary` task:

Example: Assemble components that depends on the passing/static binary of the operators component

Output of `gradle assembleDependentsOperatorsPassingStaticLibrary --max-workers=1`

```
> gradle assembleDependentsOperatorsPassingStaticLibrary --max-workers=1
> Task :compileOperatorsTestPassingCUnitExeOperatorsC
> Task :operatorsTestCUnitLauncher
> Task :compileOperatorsTestPassingCUnitExeOperatorsTestC
> Task :compileOperatorsTestPassingCUnitExeOperatorsTestCunitLauncher
> Task :linkOperatorsTestPassingCUnitExe
> Task :operatorsTestPassingCUnitExe
> Task :assembleDependentsOperatorsTestPassingCUnitExe
> Task :compileOperatorsPassingStaticLibraryOperatorsC
> Task :createOperatorsPassingStaticLibrary
> Task :operatorsPassingStaticLibrary
> Task :assembleDependentsOperatorsPassingStaticLibrary

BUILD SUCCESSFUL in 0s
7 actionable tasks: 7 executed
```

In the output above, the targeted binary gets assembled as well as the test suite binary that depends on it.

You can also assemble *all* of the dependents of a component (i.e. of all its binaries/variants) using the corresponding component task, e.g. `assembleDependentsOperators`. This is useful if you have many combinations of build types, flavors and platforms and want to assemble all of them.

Building dependents

For each NativeBinarySpec, Gradle will create a task named `buildDependents${component.name}${binary.variant}` that *builds* (compile, link and check) the binary and all of its dependent binaries.

For each NativeComponentSpec, Gradle will create a task named `buildDependents${component.name}` that *builds* all the binaries of the component and all of their dependent binaries.

For example, to build the dependents of the "passing" flavor of the "static" library binary of the

"operators" component, you would run the `buildDependentsOperatorsPassingStaticLibrary` task:

Example: Build components that depends on the passing/static binary of the operators component

Output of `gradle buildDependentsOperatorsPassingStaticLibrary --max-workers=1`

```
> gradle buildDependentsOperatorsPassingStaticLibrary --max-workers=1
> Task :compileOperatorsTestPassingCUnitExeOperatorsC
> Task :operatorsTestCUnitLauncher
> Task :compileOperatorsTestPassingCUnitExeOperatorsTestC
> Task :compileOperatorsTestPassingCUnitExeOperatorsTestCunitLauncher
> Task :linkOperatorsTestPassingCUnitExe
> Task :operatorsTestPassingCUnitExe
> Task :installOperatorsTestPassingCUnitExe
> Task :runOperatorsTestPassingCUnitExe
> Task :checkOperatorsTestPassingCUnitExe
> Task :buildDependentsOperatorsTestPassingCUnitExe
> Task :compileOperatorsPassingStaticLibraryOperatorsC
> Task :createOperatorsPassingStaticLibrary
> Task :operatorsPassingStaticLibrary
> Task :buildDependentsOperatorsPassingStaticLibrary

BUILD SUCCESSFUL in 0s
9 actionable tasks: 9 executed
```

In the output above, the targeted binary as well as the test suite binary that depends on it are built and the test suite has run.

You can also build *all* of the dependents of a component (i.e. of all its binaries/variants) using the corresponding component task, e.g. `buildDependentsOperators`.

Tasks

For each NativeBinarySpec that can be produced by a build, a single *lifecycle task* is constructed that can be used to create that binary, together with a set of other tasks that do the actual work of compiling, linking or assembling the binary.

`${component.name}Executable`

Component Type

NativeExecutableSpec

Native Binary Type

NativeExecutableBinarySpec

Location of created binary

`${project.buildDir}/exe/${component.name}/${component.name}`

`${component.name}SharedLibrary`

Component Type

NativeLibrarySpec

Native Binary Type

SharedLibraryBinarySpec

Location of created binary

`${project.buildDir}/libs/${component.name}/shared/lib${component.name}.so`

`${component.name}StaticLibrary`

Component Type

NativeLibrarySpec

Native Binary Type

StaticLibraryBinarySpec

Location of created binary

`${project.buildDir}/libs/${component.name}/static/${component.name}.a`

Check tasks

For each NativeBinarySpec that can be produced by a build, a single *check task* is constructed that can be used to assemble and check that binary.

`check${component.name}Executable`

Component Type

NativeExecutableSpec

Native Binary Type

NativeExecutableBinarySpec

`check${component.name}SharedLibrary`

Component Type

NativeLibrarySpec

Native Binary Type

SharedLibraryBinarySpec

`check${component.name}StaticLibrary`

Component Type

NativeLibrarySpec

Native Binary Type

SharedLibraryBinarySpec

The built-in `check` task depends on all the *check tasks* for binaries in the project. Without either CUnit or GoogleTest plugins, the binary check task only depends on the *lifecycle task* that assembles

the binary, see Native tasks.

When the CUnit or GoogleTest plugins are applied, the task that executes the test suites for a component are automatically wired to the appropriate *check task*.

You can also add custom check tasks as follows:

Example: Adding a custom check task

build.gradle

```
plugins {
    id "cpp"
}
// You don't need to apply the plugin below if you're already using CUnit or
GoogleTest support
apply plugin: TestingModelBasePlugin

tasks.register('myCustomCheck') {
    doLast {
        println 'Executing my custom check'
    }
}

model {
    components {
        hello(NativeLibrarySpec) {
            binaries.all {
                // Register our custom check task to all binaries of this component
                checkedBy $.tasks.myCustomCheck
            }
        }
    }
}
```

Now, running `check` or any of the *check tasks* for the `hello` binaries will run the custom check task:

Example: Running checks for a given binary

Output of `gradle checkHelloSharedLibrary`

```
> gradle checkHelloSharedLibrary

> Task :myCustomCheck
Executing my custom check

> Task :checkHelloSharedLibrary

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Working with shared libraries

For each executable binary produced, the `cpp` plugin provides an `install${binary.name}` task, which creates a development install of the executable, along with the shared libraries it requires. This allows you to run the executable without needing to install the shared libraries in their final locations.

Finding out more about your project

Gradle provides a report that you can run from the command-line that shows some details about the components and binaries that your project produces. To use this report, just run `gradle components`. Below is an example of running this report for one of the sample projects:

Example: The components report

Output of `gradle components`

```
> gradle components

> Task :components

------------------------------------------------------------
Root project 'cpp'
------------------------------------------------------------

Native library 'hello'
----------------------

Source sets
    C++ source 'hello:cpp'
        srcDir: src/hello/cpp

Binaries
    Shared library 'hello:sharedLibrary'
        build using task: :helloSharedLibrary
        build type: build type 'debug'
        flavor: flavor 'default'
        target platform: platform 'current'
        tool chain: Tool chain 'clang' (Clang)
        shared library file: build/libs/hello/shared/libhello.dylib
    Static library 'hello:staticLibrary'
        build using task: :helloStaticLibrary
        build type: build type 'debug'
        flavor: flavor 'default'
        target platform: platform 'current'
        tool chain: Tool chain 'clang' (Clang)
        static library file: build/libs/hello/static/libhello.a

Native executable 'main'
------------------------
```

```
Source sets
    C++ source 'main:cpp'
        srcDir: src/main/cpp

Binaries
    Executable 'main:executable'
        build using task: :mainExecutable
        install using task: :installMainExecutable
        build type: build type 'debug'
        flavor: flavor 'default'
        target platform: platform 'current'
        tool chain: Tool chain 'clang' (Clang)
        executable file: build/exe/main/main

Note: currently not all plugins register their components, so some components may not
be visible here.

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Language support

Presently, Gradle supports building native software from any combination of source languages listed below. A native binary project will contain one or more named `FunctionalSourceSet` instances (e.g. 'main', 'test', etc), each of which can contain `LanguageSourceSet`s containing source files, one for each language.

- C
- C++
- Objective-C
- Objective-C++
- Assembly
- Windows resources

C++ sources

C++ language support is provided by means of the `'cpp'` plugin.

Example: The 'cpp' plugin

build.gradle

```
plugins {
    id 'cpp'
}
```

C++ sources to be included in a native binary are provided via a CppSourceSet, which defines a set

of C++ source files and optionally a set of exported header files (for a library). By default, for any named component the CppSourceSet contains `.cpp` source files in `src/${name}/cpp`, and header files in `src/${name}/headers`.

While the `cpp` plugin defines these default locations for each CppSourceSet, it is possible to extend or override these defaults to allow for a different project layout.

Example: C++ source set

build.gradle

```
sources {
    cpp {
        source {
            srcDir "src/source"
            include "**/*.cpp"
        }
    }
}
```

For a library named 'main', header files in `src/main/headers` are considered the "public" or "exported" headers. Header files that should not be exported should be placed inside the `src/main/cpp` directory (though be aware that such header files should always be referenced in a manner relative to the file including them).

C sources

C language support is provided by means of the `'c'` plugin.

Example: The 'c' plugin

build.gradle

```
plugins {
    id 'c'
}
```

C sources to be included in a native binary are provided via a CSourceSet, which defines a set of C source files and optionally a set of exported header files (for a library). By default, for any named component the CSourceSet contains `.c` source files in `src/${name}/c`, and header files in `src/${name}/headers`.

While the `c` plugin defines these default locations for each CSourceSet, it is possible to extend or override these defaults to allow for a different project layout.

Example: C source set

build.gradle

```
sources {
    c {
        source {
            srcDir "src/source"
            include "**/*.c"
        }
        exportedHeaders {
            srcDir "src/include"
        }
    }
}
```

For a library named 'main', header files in `src/main/headers` are considered the "public" or "exported" headers. Header files that should not be exported should be placed inside the `src/main/c` directory (though be aware that such header files should always be referenced in a manner relative to the file including them).

Assembler sources

Assembly language support is provided by means of the `'assembler'` plugin.

Example: The 'assembler' plugin

build.gradle

```
plugins {
    id 'assembler'
}
```

Assembler sources to be included in a native binary are provided via a AssemblerSourceSet, which defines a set of Assembler source files. By default, for any named component the AssemblerSourceSet contains `.s` source files under `src/${name}/asm`.

Objective-C sources

Objective-C language support is provided by means of the `'objective-c'` plugin.

Example: The 'objective-c' plugin

build.gradle

```
plugins {
    id 'objective-c'
}
```

Objective-C sources to be included in a native binary are provided via a ObjectiveCSourceSet, which defines a set of Objective-C source files. By default, for any named component the

ObjectiveCSourceSet contains `.m` source files under `src/${name}/objectiveC`.

Objective-C++ sources

Objective-C++ language support is provided by means of the `'objective-cpp'` plugin.

Example: The 'objective-cpp' plugin

build.gradle

```
plugins {
    id 'objective-cpp'
}
```

Objective-C++ sources to be included in a native binary are provided via a ObjectiveCppSourceSet, which defines a set of Objective-C++ source files. By default, for any named component the ObjectiveCppSourceSet contains `.mm` source files under `src/${name}/objectiveCpp`.

Configuring the compiler, assembler and linker

Each binary to be produced is associated with a set of compiler and linker settings, which include command-line arguments as well as macro definitions. These settings can be applied to all binaries, an individual binary, or selectively to a group of binaries based on some criteria.

Example: Settings that apply to all binaries

build.gradle

```
model {
    binaries {
        all {
            // Define a preprocessor macro for every binary
            cppCompiler.define "NDEBUG"

            // Define toolchain-specific compiler and linker options
            if (toolChain in Gcc) {
                cppCompiler.args "-O2", "-fno-access-control"
                linker.args "-Xlinker", "-S"
            }
            if (toolChain in VisualCpp) {
                cppCompiler.args "/Zi"
                linker.args "/DEBUG"
            }
        }
    }
}
```

Each binary is associated with a particular NativeToolChain, allowing settings to be targeted based on this value.

It is easy to apply settings to all binaries of a particular type:

Example: Settings that apply to all shared libraries

build.gradle

```
// For any shared library binaries built with Visual C++,
// define the DLL_EXPORT macro
model {
    binaries {
        withType(SharedLibraryBinarySpec) {
            if (toolChain in VisualCpp) {
                cCompiler.args "/Zi"
                cCompiler.define "DLL_EXPORT"
            }
        }
    }
}
```

Furthermore, it is possible to specify settings that apply to all binaries produced for a particular `executable` or `library` component:

Example: Settings that apply to all binaries produced for the 'main' executable component

build.gradle

```
model {
    components {
        main(NativeExecutableSpec) {
            targetPlatform "x86"
            binaries.all {
                if (toolChain in VisualCpp) {
                    sources {
                        platformAsm(AssemblerSourceSet) {
                            source.srcDir "src/main/asm_i386_masm"
                        }
                    }
                    assembler.args "/Zi"
                } else {
                    sources {
                        platformAsm(AssemblerSourceSet) {
                            source.srcDir "src/main/asm_i386_gcc"
                        }
                    }
                    assembler.args "-g"
                }
            }
        }
    }
}
```

The example above will apply the supplied configuration to all `executable` binaries built.

Similarly, settings can be specified to target binaries for a component that are of a particular type: e.g. all shared libraries for the main library component.

Example: Settings that apply only to shared libraries produced for the 'main' library component

build.gradle

```
model {
    components {
        main(NativeLibrarySpec) {
            binaries.withType(SharedLibraryBinarySpec) {
                // Define a preprocessor macro that only applies to shared libraries
                cppCompiler.define "DLL_EXPORT"
            }
        }
    }
}
```

Windows Resources

When using the VisualCpp tool chain, Gradle is able to compile Window Resource (`rc`) files and link them into a native binary. This functionality is provided by the `'windows-resources'` plugin.

Example: The 'windows-resources' plugin

build.gradle

```
plugins {
    id 'windows-resources'
}
```

Windows resources to be included in a native binary are provided via a WindowsResourceSet, which defines a set of Windows Resource source files. By default, for any named component the WindowsResourceSet contains `.rc` source files under `src/${name}/rc`.

As with other source types, you can configure the location of the windows resources that should be included in the binary.

Example: Configuring the location of Windows resource sources

build-resource-only-dll.gradle

```
sources {
    rc {
        source {
            srcDirs "src/hello/rc"
        }
        exportedHeaders {
            srcDirs "src/hello/headers"
        }
    }
}
```

You are able to construct a resource-only library by providing Windows Resource sources with no other language sources, and configure the linker as appropriate:

Example: Building a resource-only dll

build-resource-only-dll.gradle

```
model {
    components {
        helloRes(NativeLibrarySpec) {
            binaries.all {
                rcCompiler.args "/v"
                linker.args "/noentry", "/machine:x86"
            }
            sources {
                rc {
                    source {
                        srcDirs "src/hello/rc"
                    }
                    exportedHeaders {
                        srcDirs "src/hello/headers"
                    }
                }
            }
        }
    }
}
```

The example above also demonstrates the mechanism of passing extra command-line arguments to the resource compiler. The `rcCompiler` extension is of type PreprocessingTool.

Library Dependencies

Dependencies for native components are binary libraries that export header files. The header files are used during compilation, with the compiled binary dependency being used during linking and execution. Header files should be organized into subdirectories to prevent clashes of commonly

named headers. For instance, if your `mylib` project has a `logging.h` header, it will make it less likely the wrong header is used if you include it as `"mylib/logging.h"` instead of `"logging.h"`.

Dependencies within the same project

A set of sources may depend on header files provided by another binary component within the same project. A common example is a native executable component that uses functions provided by a separate native library component.

Such a library dependency can be added to a source set associated with the `executable` component:

Example: Providing a library dependency to the source set

build.gradle

```
sources {
    cpp {
        lib library: "hello"
    }
}
```

Alternatively, a library dependency can be provided directly to the `NativeExecutableBinarySpec` for the `executable`.

Example: Providing a library dependency to the binary

build.gradle

```
model {
    components {
        hello(NativeLibrarySpec) {
            sources {
                c {
                    source {
                        srcDir "src/source"
                        include "**/*.c"
                    }
                    exportedHeaders {
                        srcDir "src/include"
                    }
                }
            }
        }
        main(NativeExecutableSpec) {
            sources {
                cpp {
                    source {
                        srcDir "src/source"
                        include "**/*.cpp"
                    }
                }
            }
            binaries.all {
                // Each executable binary produced uses the 'hello' static library
binary
                lib library: 'hello', linkage: 'static'
            }
        }
    }
}
```

Project Dependencies

For a component produced in a different Gradle project, the notation is similar.

Example: Declaring project dependencies

lib/build.gradle

```
plugins {
    id 'cpp'
}

model {
    components {
        main(NativeLibrarySpec)
    }

    // For any shared library binaries built with Visual C++,
    // define the DLL_EXPORT macro
    binaries {
        withType(SharedLibraryBinarySpec) {
            if (toolChain in VisualCpp) {
                cppCompiler.define "DLL_EXPORT"
            }
        }
    }
}
```

exe/build.gradle

```
plugins {
    id 'cpp'
}

model {
    components {
        main(NativeExecutableSpec) {
            sources {
                cpp {
                    lib project: ':lib', library: 'main'
                }
            }
        }
    }
}
```

Precompiled Headers

Precompiled headers are a performance optimization that reduces the cost of compiling widely used headers multiple times. This feature *precompiles* a header such that the compiled object file can be reused when compiling each source file rather than recompiling the header each time. This support is available for C, C++, Objective-C, and Objective-C++ builds.

To configure a precompiled header, first a header file needs to be defined that includes all of the headers that should be precompiled. It must be specified as the first included header in every

source file where the precompiled header should be used. It is assumed that this header file, and any headers it contains, make use of header guards so that they can be included in an idempotent manner. If header guards are not used in a header file, it is possible the header could be compiled more than once and could potentially lead to a broken build.

Example: Creating a precompiled header file

src/hello/headers/pch.h

```
#ifndef PCH_H
#define PCH_H
#include <iostream>
#include "hello.h"
#endif
```

Example: Including a precompiled header file in a source file

src/hello/cpp/hello.cpp

```
#include "pch.h"

void LIB_FUNC Greeter::hello () {
    std::cout << "Hello world!" << std::endl;
}
```

Precompiled headers are specified on a source set. Only one precompiled header file can be specified on a given source set and will be applied to all source files that declare it as the first include. If a source files does not include this header file as the first header, the file will be compiled in the normal manner (without making use of the precompiled header object file). The string provided should be the same as that which is used in the "#include" directive in the source files.

Example: Configuring a precompiled header

build.gradle

```
model {
    components {
        hello(NativeLibrarySpec) {
            sources {
                cpp {
                    preCompiledHeader "pch.h"
                }
            }
        }
    }
}
```

A precompiled header must be included in the same way for all files that use it. Usually, this means

the header file should exist in the source set "headers" directory or in a directory included on the compiler include path.

Native Binary Variants

For each executable or library defined, Gradle is able to build a number of different native binary variants. Examples of different variants include debug vs release binaries, 32-bit vs 64-bit binaries, and binaries produced with different custom preprocessor flags.

Binaries produced by Gradle can be differentiated on build type, platform, and flavor. For each of these 'variant dimensions', it is possible to specify a set of available values as well as target each component at one, some or all of these. For example, a plugin may define a range of support platforms, but you may choose to only target Windows-x86 for a particular component.

Build types

A `build type` determines various non-functional aspects of a binary, such as whether debug information is included, or what optimisation level the binary is compiled with. Typical build types are 'debug' and 'release', but a project is free to define any set of build types.

Example: Defining build types

build.gradle

```
model {
    buildTypes {
        debug
        release
    }
}
```

If no build types are defined in a project, then a single, default build type called 'debug' is added.

For a build type, a Gradle project will typically define a set of compiler/linker flags per tool chain.

Example: Configuring debug binaries

build.gradle

```
model {
    binaries {
        all {
            if (toolChain in Gcc && buildType == buildTypes.debug) {
                cppCompiler.args "-g"
            }
            if (toolChain in VisualCpp && buildType == buildTypes.debug) {
                cppCompiler.define 'DEBUG'
                linker.args '/DEBUG'
            }
        }
    }
}
```

NOTE	At this stage, it is completely up to the build script to configure the relevant compiler/linker flags for each build type. Future versions of Gradle will automatically include the appropriate debug flags for any 'debug' build type, and may be aware of various levels of optimisation as well.

Platform

An executable or library can be built to run on different operating systems and cpu architectures, with a variant being produced for each platform. Gradle defines each OS/architecture combination as a NativePlatform, and a project may define any number of platforms. If no platforms are defined in a project, then a single, default platform 'current' is added.

NOTE	Presently, a `Platform` consists of a defined operating system and architecture. As we continue to develop the native binary support in Gradle, the concept of Platform will be extended to include things like C-runtime version, Windows SDK, ABI, etc. Sophisticated builds may use the extensibility of Gradle to apply additional attributes to each platform, which can then be queried to specify particular includes, preprocessor macros or compiler arguments for a native binary.

Example: Defining platforms

build.gradle

```
model {
    platforms {
        x86 {
            architecture "x86"
        }
        x64 {
            architecture "x86_64"
        }
        itanium {
            architecture "ia-64"
        }
    }
}
```

For a given variant, Gradle will attempt to find a NativeToolChain that is able to build for the target platform. Available tool chains are searched in the order defined. See the tool chains section below for more details.

Flavor

Each component can have a set of named `flavors`, and a separate binary variant can be produced for each flavor. While the `build type` and `target platform` variant dimensions have a defined meaning in Gradle, each project is free to define any number of flavors and apply meaning to them in any way.

An example of component flavors might differentiate between 'demo', 'paid' and 'enterprise' editions of the component, where the same set of sources is used to produce binaries with different functions.

Example: Defining flavors

build.gradle

```
model {
    flavors {
        english
        french
    }
    components {
        hello(NativeLibrarySpec) {
            binaries.all {
                if (flavor == flavors.french) {
                    cppCompiler.define "FRENCH"
                }
            }
        }
    }
}
```

In the example above, a library is defined with a 'english' and 'french' flavor. When compiling the 'french' variant, a separate macro is defined which leads to a different binary being produced.

If no flavor is defined for a component, then a single default flavor named 'default' is used.

Selecting the build types, platforms and flavors for a component

For a default component, Gradle will attempt to create a native binary variant for each and every combination of `buildType` and `flavor` defined for the project. It is possible to override this on a per-component basis, by specifying the set of `targetBuildTypes` and/or `targetFlavors`. By default, Gradle will build for the default platform, see above, unless specified explicitly on a per-component basis by specifying a set of `targetPlatforms`.

Example: Targeting a component at particular platforms

build.gradle

```
model {
    components {
        hello(NativeLibrarySpec) {
            targetPlatform "x86"
            targetPlatform "x64"
        }
        main(NativeExecutableSpec) {
            targetPlatform "x86"
            targetPlatform "x64"
            sources {
                cpp.lib library: 'hello', linkage: 'static'
            }
        }
    }
}
```

Here you can see that the TargetedNativeComponent.targetPlatform(java.lang.String) method is used to specify a platform that the `NativeExecutableSpec` named `main` should be built for.

A similar mechanism exists for selecting TargetedNativeComponent.targetBuildTypes(java.lang.String...) and TargetedNativeComponent.targetFlavors(java.lang.String...).

Building all possible variants

When a set of build types, target platforms, and flavors is defined for a component, a NativeBinarySpec model element is created for every possible combination of these. However, in many cases it is not possible to build a particular variant, perhaps because no tool chain is available to build for a particular platform.

If a binary variant cannot be built for any reason, then the NativeBinarySpec associated with that variant will not be `buildable`. It is possible to use this property to create a task to generate all possible variants on a particular machine.

Example: Building all possible variants

build.gradle

```
model {
    tasks {
        buildAllExecutables(Task) {
            dependsOn $.binaries.findAll { it.buildable }
        }
    }
}
```

Tool chains

A single build may utilize different tool chains to build variants for different platforms. To this end, the core 'native-binary' plugins will attempt to locate and make available supported tool chains. However, the set of tool chains for a project may also be explicitly defined, allowing additional cross-compilers to be configured as well as allowing the install directories to be specified.

Defining tool chains

The supported tool chain types are:

- Gcc
- Clang
- VisualCpp

Example: Defining tool chains

build.gradle

```
model {
    toolChains {
        visualCpp(VisualCpp) {
            // Specify the installDir if Visual Studio cannot be located
            // installDir "C:/Apps/Microsoft Visual Studio 10.0"
        }
        gcc(Gcc) {
            // Uncomment to use a GCC install that is not in the PATH
            // path "/usr/bin/gcc"
        }
        clang(Clang)
    }
}
```

Each tool chain implementation allows for a certain degree of configuration (see the API documentation for more details).

Using tool chains

It is not necessary or possible to specify the tool chain that should be used to build. For a given variant, Gradle will attempt to locate a NativeToolChain that is able to build for the target platform. Available tool chains are searched in the order defined.

NOTE	When a platform does not define an architecture or operating system, the default target of the tool chain is assumed. So if a platform does not define a value for `operatingSystem`, Gradle will find the first available tool chain that can build for the specified `architecture`.

The core Gradle tool chains are able to target the following architectures out of the box. In each case, the tool chain will target the current operating system. See the next section for information on

cross-compiling for other operating systems.

Tool Chain	Architectures
GCC	x86, x86_64, arm64 (macOS Only)
Clang	x86, x86_64, arm64 (macOS only)
Visual C++	x86, x86_64, ia-64

So for GCC running on linux, the supported target platforms are 'linux/x86' and 'linux/x86_64'. For GCC running on Windows via Cygwin, platforms 'windows/x86' and 'windows/x86_64' are supported. (The Cygwin POSIX runtime is not yet modelled as part of the platform, but will be in the future.)

If no target platforms are defined for a project, then all binaries are built to target a default platform named 'current'. This default platform does not specify any `architecture` or `operatingSystem` value, hence using the default values of the first available tool chain.

Gradle provides a *hook* that allows the build author to control the exact set of arguments passed to a tool chain executable. This enables the build author to work around any limitations in Gradle, or assumptions that Gradle makes. The arguments hook should be seen as a 'last-resort' mechanism, with preference given to truly modelling the underlying domain.

Example: Reconfigure tool arguments

build.gradle

```
model {
    toolChains {
        visualCpp(VisualCpp) {
            eachPlatform {
                cppCompiler.withArguments { args ->
                    args << "-DFRENCH"
                }
            }
        }
        clang(Clang) {
            eachPlatform {
                cCompiler.withArguments { args ->
                    Collections.replaceAll(args, "CUSTOM", "-DFRENCH")
                }
                linker.withArguments { args ->
                    args.remove "CUSTOM"
                }
                staticLibArchiver.withArguments { args ->
                    args.remove "CUSTOM"
                }
            }
        }
    }
}
```

Cross-compiling with GCC

Cross-compiling is possible with the Gcc and Clang tool chains, by adding support for additional target platforms. This is done by specifying a target platform for a toolchain. For each target platform a custom configuration can be specified.

Example: Defining target platforms

build.gradle

```
model {
    toolChains {
        gcc(Gcc) {
            target("arm"){
                cppCompiler.withArguments { args ->
                    args << "-m32"
                }
                linker.withArguments { args ->
                    args << "-m32"
                }
            }
            target("sparc")
        }
    }
    platforms {
        arm {
            architecture "arm"
        }
        sparc {
            architecture "sparc"
        }
    }
    components {
        main(NativeExecutableSpec) {
            targetPlatform "arm"
            targetPlatform "sparc"
        }
    }
}
```

Visual Studio IDE integration

Gradle has the ability to generate Visual Studio project and solution files for the native components defined in your build. This ability is added by the `visual-studio` plugin. For a multi-project build, all projects with native components (and the root project) should have this plugin applied.

When the `visual-studio` plugin is applied to the root project, a task named `visualStudio` is created, which will generate a Visual Studio solution file containing all components in the build. This solution will include a Visual Studio project for each component, as well as configuring each component to build using Gradle.

A task named `openVisualStudio` is also created by the `visual-studio` plugin when the project is the root project. This task generates the Visual Studio solution and then opens the solution in Visual Studio. This means you can simply run `gradlew openVisualStudio` from the root project to generate and open the Visual Studio solution in one convenient step.

The content of the generated visual studio files can be modified via API hooks, provided by the `visualStudio` extension. Take a look at the 'visual-studio' sample, or see

VisualStudioExtension.getProjects() and VisualStudioRootExtension.getSolution() in the API documentation for more details.

CUnit support

The Gradle `cunit` plugin provides support for compiling and executing CUnit tests in your native-binary project. For each NativeExecutableSpec and NativeLibrarySpec defined in your project, Gradle will create a matching CUnitTestSuiteSpec component, named `${component.name}Test`.

CUnit sources

Gradle will create a CSourceSet named 'cunit' for each CUnitTestSuiteSpec component in the project. This source set should contain the cunit test files for the component under test. Source files can be located in the conventional location (`src/${component.name}Test/cunit`) or can be configured like any other source set.

Gradle initialises the CUnit test registry and executes the tests, utilising some generated CUnit launcher sources. Gradle will expect and call a function with the signature `void gradle_cunit_register()` that you can use to configure the actual CUnit suites and tests to execute.

Example: Registering CUnit tests

suite_operators.c

```
#include <CUnit/Basic.h>
#include "gradle_cunit_register.h"
#include "test_operators.h"

int suite_init(void) {
    return 0;
}

int suite_clean(void) {
    return 0;
}

void gradle_cunit_register() {
    CU_pSuite pSuiteMath = CU_add_suite("operator tests", suite_init, suite_clean);
    CU_add_test(pSuiteMath, "test_plus", test_plus);
    CU_add_test(pSuiteMath, "test_minus", test_minus);
}
```

NOTE	Due to this mechanism, your CUnit sources may not contain a `main` method since this will clash with the method provided by Gradle.

Building CUnit executables

A CUnitTestSuiteSpec component has an associated NativeExecutableSpec or NativeLibrarySpec component. For each NativeBinarySpec configured for the main component, a matching

CUnitTestSuiteBinarySpec will be configured on the test suite component. These test suite binaries can be configured in a similar way to any other binary instance:

Example: Configuring CUnit tests

build.gradle

```
model {
    binaries {
        withType(CUnitTestSuiteBinarySpec) {
            lib library: "cunit", linkage: "static"

            if (flavor == flavors.failing) {
                cCompiler.define "PLUS_BROKEN"
            }
        }
    }
}
```

NOTE	Both the CUnit sources provided by your project and the generated launcher require the core CUnit headers and libraries. Presently, this library dependency must be provided by your project for each CUnitTestSuiteBinarySpec.

Running CUnit tests

For each CUnitTestSuiteBinarySpec, Gradle will create a task to execute this binary, which will run all of the registered CUnit tests. Test results will be found in the `${build.dir}/test-results` directory.

Example: Running CUnit tests

build.gradle

```
plugins {
    id 'c'
    id 'cunit-test-suite'
}

model {
    flavors {
        passing
        failing
    }
    platforms {
        x86 {
            if (operatingSystem.macOsX) {
                architecture "x64"
            } else {
                architecture "x86"
            }
```

```
        }
    }
    repositories {
        libs(PrebuiltLibraries) {
            cunit {
                headers.srcDir "libs/cunit/2.1-2/include"
                binaries.withType(StaticLibraryBinary) {
                    staticLibraryFile =
                        file("libs/cunit/2.1-2/lib/" +
                             findCUnitLibForPlatform(targetPlatform))
                }
            }
        }
    }
    components {
        operators(NativeLibrarySpec) {
            targetPlatform "x86"
        }
    }
    testSuites {
        operatorsTest(CUnitTestSuiteSpec) {
            testing $.components.operators
        }
    }
}
model {
    binaries {
        withType(CUnitTestSuiteBinarySpec) {
            lib library: "cunit", linkage: "static"

            if (flavor == flavors.failing) {
                cCompiler.define "PLUS_BROKEN"
            }
        }
    }
}
```

Output of `gradle -q runOperatorsTestFailingCUnitExe`

```
> gradle -q runOperatorsTestFailingCUnitExe

There were test failures:
  1. /home/user/gradle/samples/src/operatorsTest/c/test_plus.c:6  - plus(0, -2) == -2
  2. /home/user/gradle/samples/src/operatorsTest/c/test_plus.c:7  - plus(2, 2) == 4


FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':runOperatorsTestFailingCUnitExe'.
> There were failing tests. See the results at:
file:///home/user/gradle/samples/build/test-results/operatorsTest/failing/

* Try:
> Run with --stacktrace option to get the stack trace.
> Run with --info or --debug option to get more log output.
> Run with --scan to get full insights.

* Get more help at https://help.gradle.org

BUILD FAILED in 0s
```

NOTE

The current support for CUnit is quite rudimentary. Plans for future integration include:

- Allow tests to be declared with Javadoc-style annotations.
- Improved HTML reporting, similar to that available for JUnit.
- Real-time feedback for test execution.
- Support for additional test frameworks.

GoogleTest support

The Gradle `google-test` plugin provides support for compiling and executing GoogleTest tests in your native-binary project. For each NativeExecutableSpec and NativeLibrarySpec defined in your project, Gradle will create a matching GoogleTestTestSuiteSpec component, named `${component.name}Test`.

GoogleTest sources

Gradle will create a CppSourceSet named 'cpp' for each GoogleTestTestSuiteSpec component in the project. This source set should contain the GoogleTest test files for the component under test. Source files can be located in the conventional location (`src/${component.name}Test/cpp`) or can be configured like any other source set.

Building GoogleTest executables

A GoogleTestTestSuiteSpec component has an associated NativeExecutableSpec or NativeLibrarySpec component. For each NativeBinarySpec configured for the main component, a matching GoogleTestTestSuiteBinarySpec will be configured on the test suite component. These test suite binaries can be configured in a similar way to any other binary instance:

Example: Registering GoogleTest tests

build.gradle

```
model {
    binaries {
        withType(GoogleTestTestSuiteBinarySpec) {
            lib library: "googleTest", linkage: "static"

            if (flavor == flavors.failing) {
                cppCompiler.define "PLUS_BROKEN"
            }

            if (targetPlatform.operatingSystem.linux) {
                cppCompiler.args '-pthread'
                linker.args '-pthread'

                if (toolChain instanceof Gcc || toolChain instanceof Clang) {
                    // Use C++03 with the old ABIs, as this is what the googletest
binaries were built with
                    cppCompiler.args '-std=c++03', '-D_GLIBCXX_USE_CXX11_ABI=0'
                    linker.args '-std=c++03'
                }
            }
        }
    }
}
```

NOTE	The GoogleTest sources provided by your project require the core GoogleTest headers and libraries. Presently, this library dependency must be provided by your project for each GoogleTestTestSuiteBinarySpec.

Running GoogleTest tests

For each GoogleTestTestSuiteBinarySpec, Gradle will create a task to execute this binary, which will run all of the registered GoogleTest tests. Test results will be found in the `${build.dir}/test-results` directory.

NOTE

The current support for GoogleTest is quite rudimentary. Plans for future integration include:

- Improved HTML reporting, similar to that available for JUnit.
- Real-time feedback for test execution.
- Support for additional test frameworks.

Implementing model rules in a plugin

CAUTION

Rule based configuration will be deprecated. New plugins should not use this concept. Instead, use the standard approach described in the Writing Custom Plugins chapter.

A plugin can define rules by extending RuleSource and adding methods that define the rules. The plugin class can either extend RuleSource directly or can implement Plugin and include a nested RuleSource subclass.

Refer to the API docs for RuleSource for more details.

Applying additional rules

A rule method annotated with Rules can apply a RuleSource to a target model element.

Extending Gradle

Developing Custom Gradle Task Types

Gradle supports two types of task. One such type is the simple task, where you define the task with an action closure. We have seen these in Build Script Basics. For this type of task, the action closure determines the behaviour of the task. This type of task is good for implementing one-off tasks in your build script.

The other type of task is the enhanced task, where the behaviour is built into the task, and the task provides some properties which you can use to configure the behaviour. We have seen these in Authoring Tasks. Most Gradle plugins use enhanced tasks. With enhanced tasks, you don't need to implement the task behaviour as you do with simple tasks. You simply declare the task and configure the task using its properties. In this way, enhanced tasks let you reuse a piece of behaviour in many different places, possibly across different builds.

The behaviour and properties of an enhanced task are defined by the task's class. When you declare an enhanced task, you specify the type, or class of the task.

Implementing your own custom task class in Gradle is easy. You can implement a custom task class in pretty much any language you like, provided it ends up compiled to JVM bytecode. In our examples, we are going to use Groovy as the implementation language. Groovy, Java or Kotlin are all good choices as the language to use to implement a task class, as the Gradle API has been designed to work well with these languages. In general, a task implemented using Java or Kotlin, which are statically typed, will perform better than the same task implemented using Groovy.

Packaging a task class

There are several places where you can put the source for the task class.

Build script

You can include the task class directly in the build script. This has the benefit that the task class is automatically compiled and included in the classpath of the build script without you having to do anything. However, the task class is not visible outside the build script, and so you cannot reuse the task class outside the build script it is defined in.

`buildSrc` project

You can put the source for the task class in the `rootProjectDir/buildSrc/src/main/groovy` directory (or `rootProjectDir/buildSrc/src/main/java` or `rootProjectDir/buildSrc/src/main/kotlin` depending on which language you prefer). Gradle will take care of compiling and testing the task class and making it available on the classpath of the build script. The task class is visible to every build script used by the build. However, it is not visible outside the build, and so you cannot reuse the task class outside the build it is defined in. Using the `buildSrc` project approach separates the task declaration — that is, what the task should do — from the task implementation — that is, how the task does it.

See Organizing Gradle Projects for more details about the `buildSrc` project.

Standalone project

You can create a separate project for your task class. This project produces and publishes a JAR which you can then use in multiple builds and share with others. Generally, this JAR might include some custom plugins, or bundle several related task classes into a single library. Or some combination of the two.

In our examples, we will start with the task class in the build script, to keep things simple. Then we will look at creating a standalone project.

Writing a simple task class

To implement a custom task class, you extend DefaultTask.

Example 526. Defining a custom task

build.gradle

```
abstract class GreetingTask extends DefaultTask {
}
```

build.gradle.kts

```
abstract class GreetingTask : DefaultTask() {
}
```

This task doesn't do anything useful, so let's add some behaviour. To do so, we add a method to the task and mark it with the TaskAction annotation. Gradle will call the method when the task executes. You don't have to use a method to define the behaviour for the task. You could, for instance, call `doFirst()` or `doLast()` with a closure in the task constructor to add behaviour.

Example 527. A hello world task

build.gradle

```
abstract class GreetingTask extends DefaultTask {
    @TaskAction
    def greet() {
        println 'hello from GreetingTask'
    }
}

// Create a task using the task type
tasks.register('hello', GreetingTask)
```

build.gradle.kts

```
abstract class GreetingTask : DefaultTask() {
    @TaskAction
    fun greet() {
        println("hello from GreetingTask")
    }
}

// Create a task using the task type
tasks.register<GreetingTask>("hello")
```

Output of `gradle -q hello`

```
> gradle -q hello
hello from GreetingTask
```

Let's add a property to the task, so we can customize it. Tasks are objects, and when you declare a task, you can set the properties or call methods on the task object. Here we add a `greeting` property, and set the value when we declare the `greeting` task.

Example 528. A customizable hello world task

build.gradle

```
abstract class GreetingTask extends DefaultTask {
    @Input
    abstract Property<String> getGreeting()

    GreetingTask() {
        greeting.convention('hello from GreetingTask')
    }

    @TaskAction
    def greet() {
        println greeting.get()
    }
}

// Use the default greeting
tasks.register('hello', GreetingTask)

// Customize the greeting
tasks.register('greeting',GreetingTask) {
    greeting = 'greetings from GreetingTask'
}
```

build.gradle.kts

```
abstract class GreetingTask : DefaultTask() {
    @get:Input
    abstract val greeting: Property<String>

    init {
        greeting.convention("hello from GreetingTask")
    }

    @TaskAction
    fun greet() {
        println(greeting.get())
    }
}

// Use the default greeting
tasks.register<GreetingTask>("hello")

// Customize the greeting
tasks.register<GreetingTask>("greeting") {
    greeting.set("greetings from GreetingTask")
}
```

Output of `gradle -q hello greeting`

```
> gradle -q hello greeting
hello from GreetingTask
greetings from GreetingTask
```

A standalone project

Now we will move our task to a standalone project, so we can publish it and share it with others. This project is simply a Groovy project that produces a JAR containing the task class. Here is a simple build script for the project. It applies the Groovy plugin, and adds the Gradle API as a compile-time dependency.

Example 529. A build for a custom task

build.gradle

```
plugins {
    id 'groovy'
}

dependencies {
    implementation gradleApi()
}
```

build.gradle.kts

```
plugins {
    groovy
}

dependencies {
    implementation(gradleApi())
}
```

We just follow the convention for where the source for the task class should go.

Example: A custom task

src/main/groovy/org/gradle/GreetingTask.groovy

```
package org.gradle

import org.gradle.api.DefaultTask
import org.gradle.api.tasks.TaskAction
import org.gradle.api.tasks.Input

class GreetingTask extends DefaultTask {

    @Input
    String greeting = 'hello from GreetingTask'

    @TaskAction
    def greet() {
        println greeting
    }
}
```

Using your task class in another project

To use a task class in a build script, you need to add the class to the build script's classpath. To do this, you use a `buildscript { }` block, as described in External dependencies for the build script. The following example shows how you might do this when the JAR containing the task class has been published to a local repository:

Example 530. Using a custom task in another project

build.gradle

```
buildscript {
    repositories {
        maven {
            url = uri(repoLocation)
        }
    }
    dependencies {
        classpath 'org.gradle:task:1.0-SNAPSHOT'
    }
}

tasks.register('greeting', org.gradle.GreetingTask) {
    greeting = 'howdy!'
}
```

build.gradle.kts

```
buildscript {
    repositories {
        maven {
            url = uri(repoLocation)
        }
    }
    dependencies {
        classpath("org.gradle:task:1.0-SNAPSHOT")
    }
}

tasks.register<org.gradle.GreetingTask>("greeting") {
    greeting = "howdy!"
}
```

Writing tests for your task class

You can use the ProjectBuilder class to create Project instances to use when you test your task class.

Example: Testing a custom task

src/test/groovy/org/gradle/GreetingTaskTest.groovy

```
class GreetingTaskTest {
    @Test
    void canAddTaskToProject() {
        Project project = ProjectBuilder.builder().build()
        def task = project.task('greeting', type: GreetingTask)
        assertTrue(task instanceof GreetingTask)
    }
}
```

Incremental tasks

With Gradle, it's very simple to implement a task that is skipped when all of its inputs and outputs are up to date (see Incremental Builds). However, there are times when only a few input files have changed since the last execution, and you'd like to avoid reprocessing all of the unchanged inputs. This can be particularly useful for a transformer task that converts input files to output files on a 1:1 basis.

If you'd like to optimize your build so that only out-of-date input files are processed, you can do so with an *incremental task*.

NOTE

There is the IncrementalTaskInputs API, which is available in Gradle versions before 5.4. When using IncrementalTaskInputs, it is only possible to query for all file changes of the task inputs. It is not possible to query for changes of individual input file properties. Moreover, the old API does not distinguish between incremental and non-incremental task inputs, so the task itself needs to determine where the changes originated from. Therefore, the usage of this API is deprecated, and it will be removed eventually. The new InputChanges API, which is documented here, replaces the old API and addresses its shortcomings. If you need to use the old API, have a look at the documentation in the user manual for Gradle 5.3.1.

Implementing an incremental task

For a task to process inputs incrementally, that task must contain an *incremental task action*. This is a task action method that has a single InputChanges parameter. That parameter tells Gradle that the action only wants to process the changed inputs. In addition, the task needs to declare at least one incremental file input property by using either @Incremental or @SkipWhenEmpty.

IMPORTANT	To query incremental changes for an input file property, that property always needs to return the same instance. The easiest way to accomplish this is to use one of the following types for such properties: RegularFileProperty, DirectoryProperty or ConfigurableFileCollection. You can learn more about `RegularFileProperty` and `DirectoryProperty` in the Lazy Configuration chapter, especially the sections on using read-only and configurable properties and lazy file properties.

The incremental task action can use InputChanges.getFileChanges() to find out what files have changed for a given file-based input property, be it of type `RegularFileProperty`, `DirectoryProperty` or `ConfigurableFileCollection`. The method returns an `Iterable` of type FileChanges, which in turn can be queried for the following:

- the affected file
- the change type (`ADDED`, `REMOVED` or `MODIFIED`)
- the normalized path of the changed file
- the file type of the changed file

The following example demonstrates an incremental task that has a directory input. It assumes that the directory contains a collection of text files and copies them to an output directory, reversing the text within each file. The key things to note are the type of the `inputDir` property, its annotations, and how the action (`execute()`) uses `getFileChanges()` to process the subset of files that have actually changed since the last build. You can also see how the action deletes a target file if the corresponding input file has been removed:

Example 531. Defining an incremental task action

build.gradle

```
abstract class IncrementalReverseTask extends DefaultTask {
    @Incremental
    @PathSensitive(PathSensitivity.NAME_ONLY)
    @InputDirectory
    abstract DirectoryProperty getInputDir()

    @OutputDirectory
    abstract DirectoryProperty getOutputDir()

    @Input
    abstract Property<String> getInputProperty()

    @TaskAction
    void execute(InputChanges inputChanges) {
        println(inputChanges.incremental
            ? 'Executing incrementally'
            : 'Executing non-incrementally'
        )

        inputChanges.getFileChanges(inputDir).each { change ->
            if (change.fileType == FileType.DIRECTORY) return

            println "${change.changeType}: ${change.normalizedPath}"
            def targetFile = outputDir.file(change.normalizedPath).get()
.asFile
            if (change.changeType == ChangeType.REMOVED) {
                targetFile.delete()
            } else {
                targetFile.text = change.file.text.reverse()
            }
        }
    }
}
```

build.gradle.kts

```
abstract class IncrementalReverseTask : DefaultTask() {
    @get:Incremental
    @get:PathSensitive(PathSensitivity.NAME_ONLY)
    @get:InputDirectory
    abstract val inputDir: DirectoryProperty

    @get:OutputDirectory
    abstract val outputDir: DirectoryProperty

    @get:Input
    abstract val inputProperty: Property<String>

    @TaskAction
    fun execute(inputChanges: InputChanges) {
        println(
            if (inputChanges.isIncremental) "Executing incrementally"
            else "Executing non-incrementally"
        )

        inputChanges.getFileChanges(inputDir).forEach { change ->
            if (change.fileType == FileType.DIRECTORY) return@forEach

            println("${change.changeType}: ${change.normalizedPath}")
            val targetFile =
outputDir.file(change.normalizedPath).get().asFile
            if (change.changeType == ChangeType.REMOVED) {
                targetFile.delete()
            } else {
                targetFile.writeText(change.file.readText().reversed())
            }
        }
    }
}
```

If for some reason the task is executed non-incrementally, for example by running with `--rerun-tasks`, all files are reported as `ADDED`, irrespective of the previous state. In this case, Gradle automatically removes the previous outputs, so the incremental task only needs to process the given files.

For a simple transformer task like the above example, the task action simply needs to generate output files for any out-of-date inputs and delete output files for any removed inputs.

IMPORTANT	A task may only contain a single incremental task action.

Which inputs are considered out of date?

When there is a previous execution of the task, and the only changes since that execution are to incremental input file properties, then Gradle is able to determine which input files need to be processed (incremental execution). In this case, the InputChanges.getFileChanges() method returns details for all input files for the given property that were *added*, *modified* or *removed*.

However, there are many cases where Gradle is unable to determine which input files need to be processed (non-incremental execution). Examples include:

- There is no history available from a previous execution.
- You are building with a different version of Gradle. Currently, Gradle does not use task history from a different version.
- An `upToDateWhen` criterion added to the task returns `false`.
- An input property has changed since the previous execution.
- A non-incremental input file property has changed since the previous execution.
- One or more output files have changed since the previous execution.

In all of these cases, Gradle will report all input files as `ADDED` and the `getFileChanges()` method will return details for all the files that comprise the given input property.

You can check if the task execution is incremental or not with the InputChanges.isIncremental() method.

An incremental task in action

Given the example incremental task implementation above, let's walk through some scenarios based on it.

First, consider an instance of `IncrementalReverseTask` that is executed against a set of inputs for the first time. In this case, all inputs will be considered added, as shown here:

Example 532. Running the incremental task for the first time

build.gradle

```
tasks.register('incrementalReverse', IncrementalReverseTask) {
    inputDir = file('inputs')
    outputDir = file("$buildDir/outputs")
    inputProperty = project.properties['taskInputProperty'] ?: 'original'
}
```

build.gradle.kts

```
tasks.register<IncrementalReverseTask>("incrementalReverse") {
    inputDir.set(file("inputs"))
    outputDir.set(file("$buildDir/outputs"))
    inputProperty.set(project.findProperty("taskInputProperty") as String? ?:
"original")
}
```

Build layout

```
.
├─── build.gradle
└─── inputs
    ├─── 1.txt
    ├─── 2.txt
    └─── 3.txt
```

Output of `gradle -q incrementalReverse`

```
> gradle -q incrementalReverse
Executing non-incrementally
ADDED: 1.txt
ADDED: 2.txt
ADDED: 3.txt
```

Naturally when the task is executed again with no changes, then the entire task is up to date and the task action is not executed:

Example 533. Running the incremental task with unchanged inputs

Output of `gradle incrementalReverse`

```
> gradle incrementalReverse
> Task :incrementalReverse UP-TO-DATE

BUILD SUCCESSFUL in 0s
1 actionable task: 1 up-to-date
```

When an input file is modified in some way or a new input file is added, then re-executing the task results in those files being returned by InputChanges.getFileChanges(). The following example modifies the content of one file and adds another before running the incremental task:

Example 534. Running the incremental task with updated input files

build.gradle

```gradle
tasks.register('updateInputs') {
    outputs.dir('inputs')
    doLast {
        file('inputs/1.txt').text = 'Changed content for existing file 1.'
        file('inputs/4.txt').text = 'Content for new file 4.'
    }
}
```

build.gradle.kts

```gradle.kts
tasks.register("updateInputs") {
    outputs.dir("inputs")
    doLast {
        file("inputs/1.txt").writeText("Changed content for existing file
1.")
        file("inputs/4.txt").writeText("Content for new file 4.")
    }
}
```

Output of `gradle -q updateInputs incrementalReverse`

```
> gradle -q updateInputs incrementalReverse
Executing incrementally
MODIFIED: 1.txt
ADDED: 4.txt
```

NOTE	The various mutation tasks (`updateInputs`, `removeInput`, etc) are only present to demonstrate the behavior of incremental tasks. They should not be viewed as the kinds of tasks or task implementations you should have in your own build scripts.

When an existing input file is removed, then re-executing the task results in that file being returned by InputChanges.getFileChanges() as `REMOVED`. The following example removes one of the existing files before executing the incremental task:

Example 535. Running the incremental task with an input file removed

build.gradle

```
tasks.register('removeInput') {
    outputs.dir('inputs')
    doLast {
        file('inputs/3.txt').delete()
    }
}
```

build.gradle.kts

```
tasks.register("removeInput") {
    outputs.dir("inputs")
    doLast {
        file("inputs/3.txt").delete()
    }
}
```

Output of `gradle -q removeInput incrementalReverse`

```
> gradle -q removeInput incrementalReverse
Executing incrementally
REMOVED: 3.txt
```

When an *output* file is deleted (or modified), then Gradle is unable to determine which input files are out of date. In this case, details for *all* the input files for the given property are returned by InputChanges.getFileChanges(). The following example removes just one of the output files from the build directory, but notice how all the input files are considered to be `ADDED`:

Example 536. Running the incremental task with an output file removed

build.gradle

```
tasks.register('removeOutput') {
    outputs.dir("$buildDir/outputs")
    doLast {
        file("$buildDir/outputs/1.txt").delete()
    }
}
```

build.gradle.kts

```
tasks.register("removeOutput") {
    outputs.dir("$buildDir/outputs")
    doLast {
        file("$buildDir/outputs/1.txt").delete()
    }
}
```

Output of `gradle -q removeOutput incrementalReverse`

```
> gradle -q removeOutput incrementalReverse
Executing non-incrementally
ADDED: 1.txt
ADDED: 2.txt
ADDED: 3.txt
```

The last scenario we want to cover concerns what happens when a non-file-based input property is modified. In such cases, Gradle is unable to determine how the property impacts the task outputs, so the task is executed non-incrementally. This means that *all* input files for the given property are returned by InputChanges.getFileChanges() and they are all treated as `ADDED`. The following example sets the project property `taskInputProperty` to a new value when running the `incrementalReverse` task and that project property is used to initialize the task's `inputProperty` property, as you can see in the first example of this section. Here's the output you can expect in this case:

Example 537. Running the incremental task with an input property changed

Output of `gradle -q -PtaskInputProperty=changed incrementalReverse`

```
> gradle -q -PtaskInputProperty=changed incrementalReverse
Executing non-incrementally
ADDED: 1.txt
ADDED: 2.txt
ADDED: 3.txt
```

Storing incremental state for cached tasks

Using Gradle's `InputChanges` is not the only way to create tasks that only work on changes since the last execution. Tools like the Kotlin compiler provide incrementality as a built-in feature. The way this is typically implemented is that the tool stores some analysis data about the state of the previous execution in some file. If such state files are relocatable, then they can be declared as outputs of the task. This way when the task's results are loaded from cache, the next execution can already use the analysis data loaded from cache, too.

However, if the state files are non-relocatable, then they can't be shared via the build cache. Indeed, when the task is loaded from cache, any such state files must be cleaned up to prevent stale state from confusing the tool during the next execution. Gradle can ensure such stale files are removed if they are declared via task.localState.register() or if a property is marked with the @LocalState annotation.

Declaring and Using Command Line Options

Sometimes a user wants to declare the value of an exposed task property on the command line instead of the build script. Being able to pass in property values on the command line is particularly helpful if they change more frequently. The task API supports a mechanism for marking a property to automatically generate a corresponding command line parameter with a specific name at runtime.

Declaring a command-line option

Exposing a new command line option for a task property is straightforward. You just have to annotate the corresponding setter method of a property with Option. An option requires a mandatory identifier. Additionally, you can provide an optional description. A task can expose as many command line options as properties available in the class.

Options may be declared in superinterfaces of the task class as well. If multiple interfaces declare the same property, but with different option flags, they will both work to set the property.

Let's have a look at an example to illustrate the functionality. The custom task `UrlVerify` verifies whether a given URL can be resolved by making a HTTP call and checking the response code. The URL to be verified is configurable through the property `url`. The setter method for the property is annotated with @Option.

Example: Declaring a command line option

UrlVerify.java

```
import org.gradle.api.tasks.options.Option;

public class UrlVerify extends DefaultTask {
    private String url;

    @Option(option = "url", description = "Configures the URL to be verified.")
    public void setUrl(String url) {
        this.url = url;
    }

    @Input
    public String getUrl() {
        return url;
    }

    @TaskAction
    public void verify() {
        getLogger().quiet("Verifying URL '{}'", url);

        // verify URL by making a HTTP call
    }
}
```

All options declared for a task can be rendered as console output by running the `help` task and the `--task` option.

Using an option on the command line

Using an option on the command line has to adhere to the following rules:

- The option uses a double-dash as prefix e.g. `--url`. A single dash does not qualify as valid syntax for a task option.
- The option argument follows directly after the task declaration e.g. `verifyUrl --url=http://www.google.com/`.
- Multiple options of a task can be declared in any order on the command line following the task name.

Getting back to the previous example, the build script creates a task instance of type `UrlVerify` and provides a value from the command line through the exposed option.

Example 538. Using a command line option

build.gradle

```
tasks.register('verifyUrl', UrlVerify)
```

build.gradle.kts

```
tasks.register<UrlVerify>("verifyUrl")
```

Output of `gradle -q verifyUrl --url=http://www.google.com/`

```
> gradle -q verifyUrl --url=http://www.google.com/
Verifying URL 'http://www.google.com/'
```

Supported data types for options

Gradle limits the set of data types that can be used for declaring command line options. The use on the command line differ per type.

`boolean`, `Boolean`, `Property<Boolean>`

Describes an option with the value `true` or `false`. Passing the option on the command line treats the value as `true`. For example `--enabled` equates to `true`. The absence of the option uses the default value of the property.

`String`, `Property<String>`

Describes an option with an arbitrary String value. Passing the option on the command line also requires a value e.g. `--container-id=2x94held` or `--container-id 2x94held`.

`Integer`, `Property<Integer>`

Describes an option with an integer value. Passing the option on the command line also requires a value e.g. `--network-timeout=5000` or `--network-timeout 5000`.

`enum`, `Property<enum>`

Describes an option as an enumerated type. Passing the option on the command line also requires a value e.g. `--log-level=DEBUG` or `--log-level debug`. The value is not case sensitive.

`List<String>`, `List<enum>`

Describes an option that can takes multiple values of a given type. The values for the option have to be provided as multiple declarations e.g. `--image-id=123 --image-id=456`. Other notations such as comma-separated lists or multiple values separated by a space character are currently not supported.

Documenting available values for an option

In theory, an option for a property type `String` or `List<String>` can accept any arbitrary value. Expected values for such an option can be documented programmatically with the help of the annotation OptionValues. This annotation may be assigned to any method that returns a `List` of one of the supported data types. In addition, you have to provide the option identifier to indicate the relationship between option and available values.

NOTE	Passing a value on the command line that is not supported by the option does not fail the build or throw an exception. You'll have to implement custom logic for such behavior in the task action.

This example demonstrates the use of multiple options for a single task. The task implementation provides a list of available values for the option `output-type`.

Example: Declaring available values for an option

UrlProcess.java

```
import org.gradle.api.tasks.options.Option;
import org.gradle.api.tasks.options.OptionValues;

public class UrlProcess extends DefaultTask {
    private String url;
    private OutputType outputType;

    @Option(option = "url", description = "Configures the URL to be write to the
output.")
    public void setUrl(String url) {
        this.url = url;
    }

    @Input
    public String getUrl() {
        return url;
    }

    @Option(option = "output-type", description = "Configures the output type.")
    public void setOutputType(OutputType outputType) {
        this.outputType = outputType;
    }

    @OptionValues("output-type")
    public List<OutputType> getAvailableOutputTypes() {
        return new ArrayList<OutputType>(Arrays.asList(OutputType.values()));
    }

    @Input
    public OutputType getOutputType() {
        return outputType;
    }

    @TaskAction
    public void process() {
        getLogger().quiet("Writing out the URL response from '{}' to '{}'", url,
outputType);

        // retrieve content from URL and write to output
    }

    private static enum OutputType {
        CONSOLE, FILE
    }
}
```

Listing command line options

Command line options using the annotations Option and OptionValues are self-documenting. You will see declared options and their available values reflected in the console output of the `help` task. The output renders options in alphabetical order.

Example: Listing available values for option

Output of `gradle -q help --task processUrl`

```
> gradle -q help --task processUrl
Detailed task information for processUrl

Path
     :processUrl

Type
     UrlProcess (UrlProcess)

Options
     --output-type     Configures the output type.
                       Available values are:
                            CONSOLE
                            FILE

     --url     Configures the URL to be write to the output.

     --rerun     Causes the task to be re-run even if up-to-date.

Description
     -

Group
     -
```

Limitations

Support for declaring command line options currently comes with a few limitations.

- Command line options can only be declared for custom tasks via annotation. There's no programmatic equivalent for defining options.
- Options cannot be declared globally e.g. on a project-level or as part of a plugin.
- When assigning an option on the command line then the task exposing the option needs to be spelled out explicitly e.g. `gradle check --tests abc` does not work even though the `check` task depends on the `test` task.
- If you specify a task option name that conflicts with the name of a built-in Gradle option, use the `--` delimiter before calling your task to reference that option. For more information, see Disambiguate Task Options from Built-in Options.

The Worker API

As can be seen from the discussion of incremental tasks, the work that a task performs can be viewed as discrete units (i.e. a subset of inputs that are transformed to a certain subset of outputs). Many times, these units of work are highly independent of each other, meaning they can be performed in any order and simply aggregated together to form the overall action of the task. In a single threaded execution, these units of work would execute in sequence, however if we have multiple processors, it would be desirable to perform independent units of work concurrently. By doing so, we can fully utilize the available resources at build time and complete the activity of the task faster.

The Worker API provides a mechanism for doing exactly this. It allows for safe, concurrent execution of multiple items of work during a task action. But the benefits of the Worker API are not confined to parallelizing the work of a task. You can also configure a desired level of isolation such that work can be executed in an isolated classloader or even in an isolated process. Furthermore, the benefits extend beyond even the execution of a single task. Using the Worker API, Gradle can begin to execute tasks in parallel by default. In other words, once a task has submitted its work to be executed asynchronously, and has exited the task action, Gradle can then begin the execution of other independent tasks in parallel, even if those tasks are in the same project.

Using the Worker API

NOTE	A step-by-step description of converting a normal task action to use the worker API can be found in the section on developing parallel tasks.

In order to submit work to the Worker API, two things must be provided: an implementation of the unit of work, and the parameters for the unit of work.

The parameters for the unit of work are defined as an interface or abstract class that implements WorkParameters. The parameters type must be a managed type.

You can find out more about implementing work parameters in Developing Custom Gradle Types.

The implementation is a class that extends WorkAction. This class should be abstract and should not implement the `getParameters()` method. Gradle will inject an implementation of this method at runtime with the parameters object for each unit of work.

Example 539. Defining the unit of work parameters and implementation

build.gradle

```
// The parameters for a single unit of work
interface ReverseParameters extends WorkParameters {
    RegularFileProperty getFileToReverse()
    DirectoryProperty getDestinationDir()
}

// The implementation of a single unit of work.
abstract class ReverseFile implements WorkAction<ReverseParameters> {
    private final FileSystemOperations fileSystemOperations

    @Inject
    public ReverseFile(FileSystemOperations fileSystemOperations) {
        this.fileSystemOperations = fileSystemOperations
    }

    @Override
    void execute() {
        fileSystemOperations.copy {
            from parameters.fileToReverse
            into parameters.destinationDir
            filter { String line -> line.reverse() }
        }
    }
}
```

build.gradle.kts

```
// The parameters for a single unit of work
interface ReverseParameters : WorkParameters {
    val fileToReverse : RegularFileProperty
    val destinationDir : DirectoryProperty
}

// The implementation of a single unit of work
abstract class ReverseFile @Inject constructor(val fileSystemOperations:
FileSystemOperations) : WorkAction<ReverseParameters> {
    override fun execute() {
        fileSystemOperations.copy {
            from(parameters.fileToReverse)
            into(parameters.destinationDir)
            filter { line: String -> line.reversed() }
        }
    }
}
```

A `WorkAction` implementation can inject services that provide capabilities during work execution, such as the FileSystemOperations service in the example above. See Service Injection for further information on injecting service types.

In order to submit the unit of work, it is necessary to first acquire the WorkerExecutor. To do this, a task should have a constructor annotated with `javax.inject.Inject` that accepts a WorkerExecutor parameter. Gradle will inject the instance of WorkerExecutor at runtime when the task is created. Then a WorkQueue object can be created and individual items of work can be submitted.

Example 540. Submitting a unit of work for execution

build.gradle

```
abstract class ReverseFiles extends SourceTask {
    private final WorkerExecutor workerExecutor

    @OutputDirectory
    abstract DirectoryProperty getOutputDir()

    // The WorkerExecutor will be injected by Gradle at runtime
    @Inject
    ReverseFiles(WorkerExecutor workerExecutor) {
        this.workerExecutor = workerExecutor
    }

    @TaskAction
    void reverseFiles() {
        // Create a WorkQueue to submit work items
        WorkQueue workQueue = workerExecutor.noIsolation()

        // Create and submit a unit of work for each file
        source.each { file ->
            workQueue.submit(ReverseFile.class) { ReverseParameters
parameters ->
                parameters.fileToReverse = file
                parameters.destinationDir = outputDir
            }
        }
    }
}
```

build.gradle.kts

```
// The WorkerExecutor will be injected by Gradle at runtime
abstract class ReverseFiles @Inject constructor(private val workerExecutor:
WorkerExecutor) : SourceTask() {
    @get:OutputDirectory
    abstract val outputDir: DirectoryProperty

    @TaskAction
    fun reverseFiles() {
        // Create a WorkQueue to submit work items
        val workQueue = workerExecutor.noIsolation()

        // Create and submit a unit of work for each file
        source.forEach { file ->
            workQueue.submit(ReverseFile::class) {
                fileToReverse.set(file)
                destinationDir.set(outputDir)
            }
        }
    }
}
```

Once all of the work for a task action has been submitted, it is safe to exit the task action. The work will be executed asynchronously and in parallel (up to the setting of `max-workers`). Of course, any tasks that are dependent on this task (and any subsequent task actions of this task) will not begin executing until all of the asynchronous work completes. However, other independent tasks that have no relationship to this task can begin executing immediately.

If any failures occur while executing the asynchronous work, the task will fail and a WorkerExecutionException will be thrown detailing the failure for each failed work item. This will be treated like any failure during task execution and will prevent any dependent tasks from executing.

In some cases, however, it might be desirable to wait for work to complete before exiting the task action. This is possible using the WorkQueue.await() method. As in the case of allowing the work to complete asynchronously, any failures that occur while executing an item of work will be surfaced as a WorkerExecutionException thrown from the WorkQueue.await() method.

NOTE Note that Gradle will only begin running other independent tasks in parallel when a task has exited a task action and returned control of execution to Gradle. When WorkQueue.await() is used, execution does not leave the task action. This means that Gradle will not allow other tasks to begin executing and will wait for the task action to complete before doing so.

Example 541. Waiting for asynchronous work to complete

build.gradle

```
        // Create a WorkQueue to submit work items
        WorkQueue workQueue = workerExecutor.noIsolation()

        // Create and submit a unit of work for each file
        source.each { file ->
            workQueue.submit(ReverseFile.class) { ReverseParameters
parameters ->
                parameters.fileToReverse = file
                parameters.destinationDir = outputDir
            }
        }

        // Wait for all asynchronous work submitted to this queue to complete
before continuing
        workQueue.await()
        logger.lifecycle("Created ${outputDir.get().asFile.listFiles().
length} reversed files in ${projectLayout.projectDirectory.asFile
.relativePath(outputDir.get().asFile)}")
```

build.gradle.kts

```
        // Create a WorkQueue to submit work items
        val workQueue = workerExecutor.noIsolation()

        // Create and submit a unit of work for each file
        source.forEach { file ->
            workQueue.submit(ReverseFile::class) {
                fileToReverse.set(file)
                destinationDir.set(outputDir)
            }
        }

        // Wait for all asynchronous work submitted to this queue to complete
before continuing
        workQueue.await()
        logger.lifecycle("Created ${outputDir.get().asFile.listFiles().size}
reversed files in
${outputDir.get().asFile.toRelativeString(projectLayout.projectDirectory.asFi
le)}")
```

Isolation Modes

Gradle provides three isolation modes that can be configured when creating a WorkQueue and are specified using the one of the following methods on WorkerExecutor:

WorkerExecutor.noIsolation()

This states that the work should be run in a thread with a minimum of isolation. For instance, it will share the same classloader that the task is loaded from. This is the fastest level of isolation.

WorkerExecutor.classLoaderIsolation()

This states that the work should be run in a thread with an isolated classloader. The classloader will have the classpath from the classloader that the unit of work implementation class was loaded from as well as any additional classpath entries added through ClassLoaderWorkerSpec.getClasspath().

WorkerExecutor.processIsolation()

This states that the work should be run with a maximum level of isolation by executing the work in a separate process. The classloader of the process will use the classpath from the classloader that the unit of work was loaded from as well as any additional classpath entries added through ClassLoaderWorkerSpec.getClasspath(). Furthermore, the process will be a *Worker Daemon* which will stay alive and can be reused for future work items that may have the same requirements. This process can be configured with different settings than the Gradle JVM using ProcessWorkerSpec.forkOptions(org.gradle.api.Action).

Worker Daemons

When using `processIsolation()`, gradle will start a long-lived *Worker Daemon* process that can be reused for future work items.

Example 542. Submitting an item of work to run in a worker daemon

build.gradle

```
        // Create a WorkQueue with process isolation
        WorkQueue workQueue = workerExecutor.processIsolation() {
ProcessWorkerSpec spec ->
            // Configure the options for the forked process
            forkOptions { JavaForkOptions options ->
                options.maxHeapSize = "512m"
                options.systemProperty "org.gradle.sample.showFileSize",
"true"
            }
        }

        // Create and submit a unit of work for each file
        source.each { file ->
            workQueue.submit(ReverseFile.class) { ReverseParameters
parameters ->
                parameters.fileToReverse = file
                parameters.destinationDir = outputDir
            }
        }
```

build.gradle.kts

```
        // Create a WorkQueue with process isolation
        val workQueue = workerExecutor.processIsolation() {
            // Configure the options for the forked process
            forkOptions {
                maxHeapSize = "512m"
                systemProperty("org.gradle.sample.showFileSize", "true")
            }
        }

        // Create and submit a unit of work for each file
        source.forEach { file ->
            workQueue.submit(ReverseFile::class) {
                fileToReverse.set(file)
                destinationDir.set(outputDir)
            }
        }
```

When a unit of work for a Worker Daemon is submitted, Gradle will first look to see if a compatible, idle daemon already exists. If so, it will send the unit of work to the idle daemon, marking it as

busy. If not, it will start a new daemon. When evaluating compatibility, Gradle looks at a number of criteria, all of which can be controlled through ProcessWorkerSpec.forkOptions(org.gradle.api.Action).

By default, a worker daemon starts with a maximum heap of 512MB. This can be changed by adjusting the workers fork options.

executable

A daemon is considered compatible only if it uses the same java executable.

classpath

A daemon is considered compatible if its classpath contains all of the classpath entries requested. Note that a daemon is considered compatible only if the classpath exactly matches the requested classpath.

heap settings

A daemon is considered compatible if it has at least the same heap size settings as requested. In other words, a daemon that has higher heap settings than requested would be considered compatible.

jvm arguments

A daemon is considered compatible if it has set all of the jvm arguments requested. Note that a daemon is considered compatible if it has additional jvm arguments beyond those requested (except for arguments treated specially such as heap settings, assertions, debug, etc).

system properties

A daemon is considered compatible if it has set all of the system properties requested with the same values. Note that a daemon is considered compatible if it has additional system properties beyond those requested.

environment variables

A daemon is considered compatible if it has set all of the environment variables requested with the same values. Note that a daemon is considered compatible if it has more environment variables in addition to those requested.

bootstrap classpath

A daemon is considered compatible if it contains all of the bootstrap classpath entries requested. Note that a daemon is considered compatible if it has more bootstrap classpath entries in addition to those requested.

debug

A daemon is considered compatible only if debug is set to the same value as requested (true or false).

enable assertions

A daemon is considered compatible only if enable assertions is set to the same value as requested (true or false).

default character encoding

A daemon is considered compatible only if the default character encoding is set to the same value as requested.

Worker daemons will remain running until either the build daemon that started them is stopped, or system memory becomes scarce. When available system memory is low, Gradle will begin stopping worker daemons in an attempt to minimize memory consumption.

Cancellation and timeouts

In order to support cancellation (e.g. when the user stops the build with CTRL+C) and task timeouts, custom tasks should react to their executing thread being interrupted. The same is true for work items submitted via the worker API. If a task does not respond to an interrupt within 10s, the daemon will shut down in order to free up system resources.

More details

It's often a good approach to package custom task types in a custom Gradle plugin. The plugin can provide useful defaults and conventions for the task type, and provides a convenient way to use the task type from a build script or another plugin. Please see Developing Custom Gradle Plugins for more details.

Gradle provides a number of features that are helpful when developing Gradle types, including tasks. Please see Developing Custom Gradle Types for more details.

Developing Custom Gradle Plugins

A Gradle plugin packages up reusable pieces of build logic, which can be used across many different projects and builds. Gradle allows you to implement your own plugins, so you can reuse your build logic, and share it with others.

You can implement a Gradle plugin in any language you like, provided the implementation ends up compiled as JVM bytecode. In our examples, we are going to use Java as the implementation language for standalone plugin project and Groovy or Kotlin in the buildscript plugin examples. In general, a plugin implemented using Java or Kotlin, which are statically typed, will perform better than the same plugin implemented using Groovy.

Packaging a plugin

There are several places where you can put the source for the plugin.

Build script

You can include the source for the plugin directly in the build script. This has the benefit that the plugin is automatically compiled and included in the classpath of the build script without you having to do anything. However, the plugin is not visible outside the build script, and so you cannot reuse the plugin outside the build script it is defined in.

`buildSrc` project

You can put the source for the plugin in the `rootProjectDir/buildSrc/src/main/java` directory (or `rootProjectDir/buildSrc/src/main/groovy` or `rootProjectDir/buildSrc/src/main/kotlin` depending on which language you prefer). Gradle will take care of compiling and testing the plugin and making it available on the classpath of the build script. The plugin is visible to every build script used by the build. However, it is not visible outside the build, and so you cannot reuse the plugin outside the build it is defined in.

See Organizing Gradle Projects for more details about the `buildSrc` project.

Standalone project

You can create a separate project for your plugin. This project produces and publishes a JAR which you can then use in multiple builds and share with others. Generally, this JAR might include some plugins, or bundle several related task classes into a single library. Or some combination of the two.

In our examples, we will start with the plugin in the build script, to keep things simple. Then we will look at creating a standalone project.

Writing a simple plugin

To create a Gradle plugin, you need to write a class that implements the Plugin interface. When the plugin is applied to a project, Gradle creates an instance of the plugin class and calls the instance's Plugin.apply() method. The project object is passed as a parameter, which the plugin can use to configure the project however it needs to. The following sample contains a greeting plugin, which adds a `hello` task to the project.

Example 543. A custom plugin

build.gradle

```
class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.task('hello') {
            doLast {
                println 'Hello from the GreetingPlugin'
            }
        }
    }
}

// Apply the plugin
apply plugin: GreetingPlugin
```

build.gradle.kts

```
class GreetingPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        project.task("hello") {
            doLast {
                println("Hello from the GreetingPlugin")
            }
        }
    }
}

// Apply the plugin
apply<GreetingPlugin>()
```

Output of `gradle -q hello`

```
> gradle -q hello
Hello from the GreetingPlugin
```

One thing to note is that a new instance of a plugin is created for each project it is applied to. Also note that the Plugin class is a generic type. This example has it receiving the Project type as a type parameter. A plugin can instead receive a parameter of type Settings, in which case the plugin can be applied in a settings script, or a parameter of type Gradle, in which case the plugin can be applied in an initialization script.

Making the plugin configurable

Most plugins offer some configuration options for build scripts and other plugins to use to customize how the plugin works. Plugins do this using *extension objects*. The Gradle Project has an associated ExtensionContainer object that contains all the settings and properties for the plugins that have been applied to the project. You can provide configuration for your plugin by adding an extension object to this container. An extension object is simply an object with Java Bean properties that represent the configuration.

Let's add a simple extension object to the project. Here we add a `greeting` extension object to the project, which allows you to configure the greeting.

Example 544. A custom plugin extension

build.gradle

```
interface GreetingPluginExtension {
    Property<String> getMessage()
}

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        // Add the 'greeting' extension object
        def extension = project.extensions.create('greeting',
GreetingPluginExtension)
        extension.message.convention('Hello from GreetingPlugin')
        // Add a task that uses configuration from the extension object
        project.task('hello') {
            doLast {
                println extension.message.get()
            }
        }
    }
}

apply plugin: GreetingPlugin

// Configure the extension
greeting.message = 'Hi from Gradle'
```

build.gradle.kts

```
interface GreetingPluginExtension {
    val message: Property<String>
}

class GreetingPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        // Add the 'greeting' extension object
        val extension =
project.extensions.create<GreetingPluginExtension>("greeting")
        extension.message.convention("Hello from GreetingPlugin")
        // Add a task that uses configuration from the extension object
        project.task("hello") {
            doLast {
                println(extension.message.get())
            }
        }
    }
}

apply<GreetingPlugin>()

// Configure the extension
the<GreetingPluginExtension>().message.set("Hi from Gradle")
```

Output of `gradle -q hello`

```
> gradle -q hello
Hi from Gradle
```

In this example, `GreetingPluginExtension` is an object with a property called `message`. The extension object is added to the project with the name `greeting`. This object then becomes available as a project property with the same name as the extension object.

Oftentimes, you have several related properties you need to specify on a single plugin. Gradle adds a configuration block for each extension object, so you can group settings together. The following example shows you how this works.

Example 545. A custom plugin with configuration block

build.gradle

```
interface GreetingPluginExtension {
    Property<String> getMessage()
    Property<String> getGreeter()
}

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        def extension = project.extensions.create('greeting',
GreetingPluginExtension)
        project.task('hello') {
            doLast {
                println "${extension.message.get()} from ${extension.greeter
.get()}"
            }
        }
    }
}

apply plugin: GreetingPlugin

// Configure the extension using a DSL block
greeting {
    message = 'Hi'
    greeter = 'Gradle'
}
```

build.gradle.kts

```
interface GreetingPluginExtension {
    val message: Property<String>
    val greeter: Property<String>
}

class GreetingPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        val extension =
project.extensions.create<GreetingPluginExtension>("greeting")
        project.task("hello") {
            doLast {
                println("${extension.message.get()} from
${extension.greeter.get()}")
            }
        }
    }
}

apply<GreetingPlugin>()

// Configure the extension using a DSL block
configure<GreetingPluginExtension> {
    message.set("Hi")
    greeter.set("Gradle")
}
```

Output of `gradle -q hello`

```
> gradle -q hello
Hi from Gradle
```

In this example, several settings can be grouped together within the `greeting` closure. The name of the closure block in the build script (`greeting`) needs to match the extension object name. Then, when the closure is executed, the fields on the extension object will be mapped to the variables within the closure based on the standard Groovy closure delegate feature.

In this example, several settings can be grouped together within the `configure<GreetingPluginExtension>` block. The type used on the `configure` function in the build script (`GreetingPluginExtension`) needs to match the extension type. Then, when the block is executed, the receiver of the block is the extension.

In this way, using an extension object *extends* the Gradle DSL to add a project property and DSL block for the plugin. And because an extension object is simply a regular object, you can provide your own DSL nested inside the plugin block by adding properties and methods to the extension object.

Developing project extensions

You can find out more about implementing project extensions in Developing Custom Gradle Types.

Working with files in custom tasks and plugins

When developing custom tasks and plugins, it's a good idea to be very flexible when accepting input configuration for file locations. You should use Gradle's managed properties and `project.layout` to select file or directory locations. By this, the actual location will only be resolved when the file is needed and can be reconfigured at any time during build configuration.

Example 546. Evaluating file properties lazily

build.gradle

```
abstract class GreetingToFileTask extends DefaultTask {

    @OutputFile
    abstract RegularFileProperty getDestination()

    @TaskAction
    def greet() {
        def file = getDestination().get().asFile
        file.parentFile.mkdirs()
        file.write 'Hello!'
    }
}

def greetingFile = objects.fileProperty()

tasks.register('greet', GreetingToFileTask) {
    destination = greetingFile
}

tasks.register('sayGreeting') {
    dependsOn greet
    doLast {
        def file = greetingFile.get().asFile
        println "${file.text} (file: ${file.name})"
    }
}

greetingFile.set(layout.buildDirectory.file('hello.txt'))
```

build.gradle.kts

```
abstract class GreetingToFileTask : DefaultTask() {

    @get:OutputFile
    abstract val destination: RegularFileProperty

    @TaskAction
    fun greet() {
        val file = destination.get().asFile
        file.parentFile.mkdirs()
        file.writeText("Hello!")
    }
}

val greetingFile = objects.fileProperty()

tasks.register<GreetingToFileTask>("greet") {
    destination.set(greetingFile)
}

tasks.register("sayGreeting") {
    dependsOn("greet")
    doLast {
        val file = greetingFile.get().asFile
        println("${file.readText()} (file: ${file.name})")
    }
}

greetingFile.set(layout.buildDirectory.file("hello.txt"))
```

Output of `gradle -q sayGreeting`

```
> gradle -q sayGreeting
Hello! (file: hello.txt)
```

In this example, we configure the `greet` task `destination` property as a closure/provider, which is evaluated with the Project.file(java.lang.Object) method to turn the return value of the closure/provider into a `File` object at the last minute. You will notice that in the example above we specify the `greetingFile` property value after we have configured to use it for the task. This kind of lazy evaluation is a key benefit of accepting any value when setting a file property, then resolving that value when reading the property.

Mapping extension properties to task properties

Capturing user input from the build script through an extension and mapping it to input/output properties of a custom task is a useful pattern. The build script author interacts only with the DSL

defined by the extension. The imperative logic is hidden in the plugin implementation.

Gradle provides some types that you can use in task implementations and extensions to help you with this. Refer to Lazy Configuration for more information.

A standalone project

Now we will move our plugin to a standalone project so that we can publish it and share it with others. This project is simply a Java project that produces a JAR containing the plugin classes. The easiest and the recommended way to package and publish a plugin is to use the Java Gradle Plugin Development Plugin. This plugin will automatically apply the Java Plugin, add the `gradleApi()` dependency to the api configuration, generate the required plugin descriptors in the resulting JAR file and configure the Plugin Marker Artifact to be used when publishing. Here is a simple build script for the project.

Example 547. A build for a custom plugin

build.gradle

```
plugins {
    id 'java-gradle-plugin'
}

gradlePlugin {
    plugins {
        simplePlugin {
            id = 'org.example.greeting'
            implementationClass = 'org.example.GreetingPlugin'
        }
    }
}
```

build.gradle.kts

```
plugins {
    `java-gradle-plugin`
}

gradlePlugin {
    plugins {
        create("simplePlugin") {
            id = "org.example.greeting"
            implementationClass = "org.example.GreetingPlugin"
        }
    }
}
```

Creating a plugin id

Plugin ids are fully qualified in a manner similar to Java packages (i.e. a reverse domain name). This helps to avoid collisions and provides a way to group plugins with similar ownership.

Your plugin id should be a combination of components that reflect namespace (a reasonable pointer to you or your organization) and the name of the plugin it provides. For example if you had a Github account named "foo" and your plugin was named "bar", a suitable plugin id might be `com.github.foo.bar`. Similarly, if the plugin was developed at the baz organization, the plugin id might be `org.baz.bar`.

Plugin ids should conform to the following:

- May contain any alphanumeric character, '.', and '-'.
- Must contain at least one '.' character separating the namespace from the name of the plugin.
- Conventionally use a lowercase reverse domain name convention for the namespace.
- Conventionally use only lowercase characters in the name.
- `org.gradle` and `com.gradleware` namespaces may not be used.
- Cannot start or end with a '.' character.
- Cannot contain consecutive '.' characters (i.e. '..').

Although there are conventional similarities between plugin ids and package names, package names are generally more detailed than is necessary for a plugin id. For instance, it might seem reasonable to add "gradle" as a component of your plugin id, but since plugin ids are only used for Gradle plugins, this would be superfluous. Generally, a namespace that identifies ownership and a name are all that are needed for a good plugin id.

Publishing your plugin

If you are publishing your plugin internally for use within your organization, you can publish it like any other code artifact. See the Ivy and Maven chapters on publishing artifacts.

If you are interested in publishing your plugin to be used by the wider Gradle community, you can publish it to the Gradle Plugin Portal. This site provides the ability to search for and gather information about plugins contributed by the Gradle community. Please refer to the corresponding section on how to make your plugin available on this site.

Using your plugin in another project

To use a plugin in a build script, you need to configure the repository in `pluginManagement {}` block of the project's settings file. The following example shows how you might do this when the plugin has been published to a local repository:

Example 548. Using a custom plugin in another project

settings.gradle

```
pluginManagement {
    repositories {
        maven {
            url = uri(repoLocation)
        }
    }
}
```

build.gradle

```
plugins {
    id 'org.example.greeting' version '1.0-SNAPSHOT'
}
```

settings.gradle.kts

```
pluginManagement {
    repositories {
        maven {
            url = uri(repoLocation)
        }
    }
}
```

build.gradle.kts

```
plugins {
    id("org.example.greeting") version "1.0-SNAPSHOT"
}
```

Note for plugins published without `java-gradle-plugin`

If your plugin was published without using the Java Gradle Plugin Development Plugin, the publication will be lacking Plugin Marker Artifact, which is needed for plugins DSL to locate the plugin. In this case, the recommended way to resolve the plugin in another project is to add a `resolutionStrategy` section to the `pluginManagement {}` block of the project's settings file as shown below.

Example 549. Resolution strategy for plugins without Plugin Marker Artifact

settings.gradle

```
resolutionStrategy {
    eachPlugin {
        if (requested.id.namespace == 'org.example') {
            useModule("org.example:custom-plugin:${requested.version}")
        }
    }
}
```

settings.gradle.kts

```
resolutionStrategy {
    eachPlugin {
        if (requested.id.namespace == "org.example") {
            useModule("org.example:custom-plugin:${requested.version}")
        }
    }
}
```

Precompiled script plugins

In addition to plugins written as standalone projects, Gradle also allows you to provide build logic written in either Groovy or Kotlin DSLs as precompiled script plugins. You write these as `*.gradle` files in `src/main/groovy` directory or `*.gradle.kts` files in `src/main/kotlin` directory.

WARNING

Precompiled script plugin names have two important limitations:

- They cannot start with `org.gradle`.
- They cannot have the same name as a built-in plugin id.

This ensures that the precompiled script plugins won't be silently ignored.

Precompiled script plugins are compiled into class files and packaged into a jar. For all intents and purposes, they are binary plugins and can be applied by plugin ID, tested and published as binary plugins. In fact, the plugin metadata for them is generated using the Gradle Plugin Development Plugin.

NOTE

Kotlin DSL precompiled script plugins built with Gradle 6.0 cannot be used with earlier versions of Gradle. This limitation will be lifted in a future version of Gradle.

Groovy DSL precompiled script plugins are available starting with Gradle 6.4. Groovy DSL precompiled script plugins can be applied in projects that use Gradle 5.0 and later.

To apply a precompiled script plugin, you need to know its ID which is derived from the plugin script's filename (minus the `.gradle` extension).

To apply a precompiled script plugin, you need to know its ID which is derived from the plugin script's filename (minus the `.gradle.kts` extension) and its (optional) package declaration.

For example, the script `src/main/groovy/java-library-convention.gradle` would have a plugin ID of `java-library-convention`. Likewise, `src/main/groovy/my.java-library-convention.gradle` would result in a plugin ID of `my.java-library-convention`.

For example, the script `src/main/kotlin/java-library-convention.gradle.kts` would have a plugin ID of `java-library-convention` (assuming it has no package declaration). Likewise, `src/main/kotlin/my/java-library-convention.gradle.kts` would result in a plugin ID of `my.java-library-convention` as long as it has a package declaration of `my`.

To demonstrate how you can implement and use a precompiled script plugin, let's walk through an example based on a `buildSrc` project.

First, you need a `buildSrc/build.gradle` file that applies the `groovy-gradle-plugin` plugin:

First, you need a `buildSrc/build.gradle.kts` file that applies the `kotlin-dsl` plugin:

Example 550. Enabling precompiled script plugins

buildSrc/build.gradle

```
plugins {
    id 'groovy-gradle-plugin'
}
```

buildSrc/build.gradle.kts

```
plugins {
    `kotlin-dsl`
}

repositories {
    mavenCentral()
}
```

We recommend that you also create a `buildSrc/settings.gradle` file, which you may leave empty.

We recommend that you also create a `buildSrc/settings.gradle.kts` file, which you may leave empty.

Next, create a new `java-library-convention.gradle` file in the `buildSrc/src/main/groovy` directory and set its contents to the following:

Next, create a new `java-library-convention.gradle.kts` file in the `buildSrc/src/main/kotlin` directory and set its contents to the following:

Example 551. Creating a simple script plugin

buildSrc/src/main/groovy/java-library-convention.gradle

```
plugins {
    id 'java-library'
    id 'checkstyle'
}

java {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}

checkstyle {
    maxWarnings = 0
    // ...
}

tasks.withType(JavaCompile) {
    options.warnings = true
    // ...
}

dependencies {
    testImplementation("junit:junit:4.13")
    // ...
}
```

buildSrc/src/main/kotlin/java-library-convention.gradle.kts

```
plugins {
    `java-library`
    checkstyle
}

java {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}

checkstyle {
    maxWarnings = 0
    // ...
}

tasks.withType<JavaCompile> {
    options.isWarnings = true
    // ...
}

dependencies {
    testImplementation("junit:junit:4.13")
    // ...
}
```

This script plugin simply applies the Java Library and Checkstyle Plugins and configures them. Note that this will actually apply the plugins to the main project, i.e. the one that applies the precompiled script plugin.

Finally, apply the script plugin to the root project as follows:

Example 552. Applying the precompiled script plugin to the main project

build.gradle

```
plugins {
    id 'java-library-convention'
}
```

build.gradle.kts

```
plugins {
    `java-library-convention`
}
```

Applying external plugins in precompiled script plugins

In order to apply an external plugin in a precompiled script plugin, it has to be added to the plugin project's implementation classpath in the plugin's build file.

buildSrc/build.gradle

```
plugins {
    id 'groovy-gradle-plugin'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'com.bmuschko:gradle-docker-plugin:6.4.0'
}
```

buildSrc/build.gradle.kts

```
plugins {
    `kotlin-dsl`
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("com.bmuschko:gradle-docker-plugin:6.4.0")
}
```

It can then be applied in the precompiled script plugin.

buildSrc/src/main/groovy/my-plugin.gradle

```
plugins {
    id 'com.bmuschko.docker-remote-api'
}
```

buildSrc/src/main/kotlin/my-plugin.gradle.kts

```
plugins {
    id("com.bmuschko.docker-remote-api")
}
```

The plugin version in this case is defined in the dependency declaration.

Writing tests for your plugin

You can use the ProjectBuilder class to create Project instances to use when you test your plugin implementation.

Example: Testing a custom plugin

src/test/java/org/example/GreetingPluginTest.java

```
public class GreetingPluginTest {
    @Test
    public void greeterPluginAddsGreetingTaskToProject() {
        Project project = ProjectBuilder.builder().build();
        project.getPluginManager().apply("org.example.greeting");

        assertTrue(project.getTasks().getByName("hello") instanceof GreetingTask);
    }
}
```

More details

Plugins often also provide custom task types. Please see Developing Custom Gradle Task Types for more details.

Gradle provides a number of features that are helpful when developing Gradle types, including plugins. Please see Developing Custom Gradle Types for more details.

CAUTION	When developing Gradle Plugins, it is important to be cautious when logging information to the build log. Logging sensitive information (e.g. credentials, tokens, certain environment variables) is considered a security vulnerability. Build logs for public Continuous Integration services are world-viewable and can expose this sensitive information.

Behind the scenes

So how does Gradle find the Plugin implementation? The answer is - you need to provide a properties file in the JAR's `META-INF/gradle-plugins` directory that matches the id of your plugin, which is handled by Java Gradle Plugin Development Plugin.

Example: Wiring for a custom plugin

Given a plugin with ID `org.example.greeting` and implementation class `org.example.GreetingPlugin`:

src/main/resources/META-INF/gradle-plugins/org.example.greeting.properties

```
implementation-class=org.example.GreetingPlugin
```

Notice that the properties filename matches the plugin id and is placed in the resources folder, and that the `implementation-class` property identifies the Plugin implementation class.

Developing Custom Gradle Types

There are several different kinds of "add-ons" to Gradle that you can develop, such as plugins, tasks, project extensions or artifact transforms, that are all implemented as classes and other types that can run on the JVM. This chapter discusses some of the features and concepts that are common to these types. You can use these features to help implement custom Gradle types and provide a consistent DSL for your users.

This chapter applies to the following types:

- Plugin types.
- Task types.
- Artifact transform parameters types.
- Worker API work action parameters types.
- Extension objects created using `ExtensionContainer.create()`, for example a project extension registered by a plugin.
- Objects created using `ObjectFactory.newInstance()`.
- Objects created for a managed nested property.
- Elements of a `NamedDomainObjectContainer`.

Configuration using properties

The custom Gradle types that you implement often hold some configuration that you want to make available to build scripts and other plugins. For example, a download task may have configuration that specifies the URL to download from and the file system location to write the result to.

Managed properties

Gradle provides its own *managed properties* concept that allows you to declare each property as an abstract getter (Java, Groovy) or an abstract property (Kotlin). Gradle then provides the implementation for such a property automatically. It is called a *managed property*, as Gradle takes care of managing the state of the property. A property may be *mutable*, meaning that it has both a `get()` method and `set()` method, or *read-only*, meaning that it has only a `get()` method. *Read-only* properties are also called *providers*.

Mutable managed properties

To declare a mutable managed property, add an abstract getter method of type `Property<T>` - where `T` can be any serializable type or a fully Gradle managed type. (See the list further down for more specific property types.) The property must not have any setter methods. Here is an example of a task type with an `uri` property of type `URI`:

Example 553. Mutable managed property

Download.java

```
public abstract class Download extends DefaultTask {

    @Input
    public abstract Property<URI> getUri(); // abstract getter of type Property<T>

    @TaskAction
    void run() {
        System.out.println("Downloading " + getUri().get()); // Use the `uri`
property
    }
}
```

Note that for a property to be considered a mutable managed property, the property's getter methods must be `abstract` and have `public` or `protected` visibility. The property type must be one of the following:

- `Property<T>`
- `RegularFileProperty`
- `DirectoryProperty`
- `ListProperty<T>`
- `SetProperty<T>`

- `MapProperty<K, V>`
- `ConfigurableFileCollection`
- `ConfigurableFileTree`
- `DomainObjectSet<T>`
- `NamedDomainObjectContainer<T>`

Gradle creates values for managed properties in the same way as ObjectFactory.

Read-only managed properties

To declare a read-only managed property, also called provider, add a getter method of type `Provider<T>`. The method implementation then needs to derive the value, for example from other properties.

Here is an example of a task type with a `uri` provider that is derived from a `location` property:

Example 554. Read-only managed property

Download.java

```
public abstract class Download extends DefaultTask {
    @Input
    public abstract Property<String> getLocation();

    @Internal
    public Provider<URI> getUri() {
        return getLocation().map(l -> URI.create("https://" + l));
    }

    @TaskAction
    void run() {
        System.out.println("Downloading " + getUri().get());  // Use the `uri`
provider (read-only property)
    }
}
```

Read-only managed nested properties

To declare a read-only managed nested property, add an abstract getter method for the property to the type annotated with `@Nested`. The property should not have any setter methods. Gradle provides an implementation for the getter method, and also creates a value for the property. The nested type is also treated as a custom type, and can use the features discussed in this chapter.

This pattern is useful when a custom type has a nested complex type which has the same lifecycle. If the lifecycle is different, consider using `Property<NestedType>` instead.

Here is an example of a task type with a `resource` property. The `Resource` type is also a custom Gradle type and defines some managed properties:

Example 555. Read-only managed nested property

Download.java

```
public abstract class Download extends DefaultTask {
    @Nested
    public abstract Resource getResource(); // Use an abstract getter method
annotated with @Nested

    @TaskAction
    void run() {
        // Use the `resource` property
        System.out.println("Downloading https://" + getResource().getHostName()
.get() + "/" + getResource().getPath().get());
    }
}

public interface Resource {
    @Input
    Property<String> getHostName();
    @Input
    Property<String> getPath();
}
```

Note that for a property to be considered a read-only managed nested property, the property's getter methods must be `abstract` and have `public` or `protected` visibility. The property must not have any setter methods. In addition, the property getter must be annotated with `@Nested`.

Read-only managed "name" property

If the type contains an abstract property called "name" of type `String`, Gradle provides an implementation for the getter method, and extends each constructor with a "name" parameter, which comes before all other constructor parameters. If the type is an interface, Gradle will provide a constructor with a single "name" parameter and `@Inject` semantics.

You can have your type implement or extend the Named interface, which defines such a read-only "name" property.

Managed types

A *managed type* is an abstract class or interface with no fields and whose properties are all managed. That is, it is a type whose state is entirely managed by Gradle.

A *named managed type* is a managed type that additionally has an abstract property "name" of type `String`. Named managed types are especially useful as the element type of NamedDomainObjectContainer (see below).

Example 556. Managed type defined as interface

Resource.java

```
public interface Resource {
    @Input
    Property<String> getHostName();
    @Input
    Property<String> getPath();
}
```

Java bean properties.

Sometimes you may see properties implemented in the Java bean property style. That is, they do not use a `Property<T>` or `Provider<T>` types but are instead implemented with concrete setter and getter methods (or corresponding conveniences in Groovy or Kotlin). This style of property definition is legacy in Gradle and is discouraged. Properties in Gradle's core plugins that are still of this style will be migrated to managed properties in future versions.

DSL support and extensibility

When Gradle creates an instance of a custom type, it *decorates* the instance to mix-in DSL and extensibility support.

Each decorated instance implements ExtensionAware, and so can have extension objects attached to it.

Note that plugins and the elements of containers created using Project.container() are currently not decorated, due to backwards compatibility issues.

Service injection

Gradle provides a number of useful services that can be used by custom Gradle types. For example, the WorkerExecutor service can be used by a task to run work in parallel, as seen in the worker API section. The services are made available through *service injection*.

Available services

The following services are available for injection:

- ObjectFactory - Allows model objects to be created. See Creating objects explicitly for more details.
- ProjectLayout - Provides access to key project locations. See lazy configuration for more details. This service is unavailable in Worker API actions.
- ProviderFactory - Creates `Provider` instances. See lazy configuration for more details.
- WorkerExecutor - Allows a task to run work in parallel. See the worker API for more details.
- FileSystemOperations - Allows a task to run operations on the filesystem such as deleting files,

copying files or syncing directories.

- ArchiveOperations - Allows a task to run operations on archive files such as ZIP or TAR files.
- ExecOperations - Allows a task to run external processes with dedicated support for running external `java` programs.
- ToolingModelBuilderRegistry - Allows a plugin to registers a Gradle tooling API model.

Out of the above, `ProjectLayout` and `WorkerExecutor` services are only available for injection in project plugins.

Constructor injection

There are 2 ways that an object can receive the services that it needs. The first option is to add the service as a parameter of the class constructor. The constructor must be annotated with the `javax.inject.Inject` annotation. Gradle uses the declared type of each constructor parameter to determine the services that the object requires. The order of the constructor parameters and their names are not significant and can be whatever you like.

Here is an example that shows a task type that receives an `ObjectFactory` via its constructor:

Example 557. Constructor service injection

Download.java

```
public class Download extends DefaultTask {
    private final DirectoryProperty outputDirectory;

    // Inject an ObjectFactory into the constructor
    @Inject
    public Download(ObjectFactory objectFactory) {
        // Use the factory
        outputDirectory = objectFactory.directoryProperty();
    }

    @OutputDirectory
    public DirectoryProperty getOutputDirectory() {
        return outputDirectory;
    }

    @TaskAction
    void run() {
        // ...
    }
}
```

Property injection

Alternatively, a service can be injected by adding a property getter method annotated with the `javax.inject.Inject` annotation to the class. This can be useful, for example, when you cannot

change the constructor of the class due to backwards compatibility constraints. This pattern also allows Gradle to defer creation of the service until the getter method is called, rather than when the instance is created. This can help with performance. Gradle uses the declared return type of the getter method to determine the service to make available. The name of the property is not significant and can be whatever you like.

The property getter method must be `public` or `protected`. The method can be `abstract` or, in cases where this isn't possible, can have a dummy method body. The method body is discarded.

Here is an example that shows a task type that receives a two services via property getter methods:

Example 558. Property service injection

Download.java

```
public abstract class Download extends DefaultTask {
    // Use an abstract getter method
    @Inject
    protected abstract ObjectFactory getObjectFactory();

    // Alternatively, use a getter method with a dummy implementation
    @Inject
    protected WorkerExecutor getWorkerExecutor() {
        // Method body is ignored
        throw new UnsupportedOperationException();
    }

    @TaskAction
    void run() {
        WorkerExecutor workerExecutor = getWorkerExecutor();
        ObjectFactory objectFactory = getObjectFactory();
        // Use the executor and factory ...
    }
}
```

Creating objects explicitly

NOTE Prefer letting Gradle create objects automatically by using managed properties.

A custom Gradle type can use the ObjectFactory service to create instances of Gradle types to use for its property values. These instances can make use of the features discussed in this chapter, allowing you to create objects and a nested DSL.

In the following example, a project extension receives an `ObjectFactory` instance through its constructor. The constructor uses this to create a nested `Resource` object (also a custom Gradle type) and makes this object available through the `resource` property.

Example 559. Nested object creation

DownloadExtension.java

```
public class DownloadExtension {
    // A nested instance
    private final Resource resource;

    @Inject
    public DownloadExtension(ObjectFactory objectFactory) {
        // Use an injected ObjectFactory to create a Resource object
        resource = objectFactory.newInstance(Resource.class);
    }

    public Resource getResource() {
        return resource;
    }
}

public interface Resource {
    Property<URI> getUri();
}
```

Collection types

Gradle provides types for maintaining collections of objects, intended to work well to extends Gradle's DSLs and provide useful features such as lazy configuration.

NamedDomainObjectContainer

A NamedDomainObjectContainer manages a set of objects, where each element has a name associated with it. The container takes care of creating and configuring the elements, and provides a DSL that build scripts can use to define and configure elements. It is intended to hold objects which are themselves configurable, for example a set of custom Gradle objects.

Gradle uses `NamedDomainObjectContainer` type extensively throughout the API. For example, the `project.tasks` object used to manage the tasks of a project is a `NamedDomainObjectContainer<Task>`.

You can create a container instance using the ObjectFactory service, which provides the ObjectFactory.domainObjectContainer() method. This is also available using the Project.container() method, however in a custom Gradle type it's generally better to use the injected `ObjectFactory` service instead of passing around a `Project` instance.

You can also create a container instance using a read-only managed property, described above.

In order to use a type with any of the `domainObjectContainer()` methods, it must either

- be a named managed type; or
- expose a property named "name" as the unique, and constant, name for the object. The

`domainObjectContainer(Class)` variant of the method creates new instances by calling the constructor of the class that takes a string argument, which is the desired name of the object.

Objects created this way are treated as custom Gradle types, and so can make use of the features discussed in this chapter, for example service injection or managed properties.

See the above link for `domainObjectContainer()` method variants that allow custom instantiation strategies.

Example 560. Managing a collection of objects

DownloadExtension.java

```
public interface DownloadExtension {
    NamedDomainObjectContainer<Resource> getResources();
}

public interface Resource {
    // Type must have a read-only 'name' property
    String getName();

    Property<URI> getUri();

    Property<String> getUserName();
}
```

For each container property, Gradle automatically adds a block to the Groovy and Kotlin DSL that you can use to configure the contents of the container:

Example 561. Configure block

build.gradle.kts

```
plugins {
    id("org.gradle.sample.download")
}

download {
    // Can use a block to configure the container contents
    resources {
        register("gradle") {
            uri.set(uri("https://gradle.org"))
        }
    }
}
```

build.gradle

```
plugins {
    id("org.gradle.sample.download")
}

download {
    // Can use a block to configure the container contents
    resources {
        register('gradle') {
            uri = uri('https://gradle.org')
        }
    }
}
```

ExtensiblePolymorphicDomainObjectContainer

An ExtensiblePolymorphicDomainObjectContainer is a `NamedDomainObjectContainer` that allows you to define instantiation strategies for different types of objects.

You can create an instance using the ObjectFactory.polymorphicDomainObjectContainer() method.

NamedDomainObjectSet

A NamedDomainObjectSet holds a set of configurable objects, where each element has a name associated with it. This is similar to `NamedDomainObjectContainer`, however a `NamedDomainObjectSet` doesn't manage the objects in the collection. They need to be created and added manually.

You can create an instance using the ObjectFactory.namedDomainObjectSet() method.

NamedDomainObjectList

A NamedDomainObjectList holds a list of configurable objects, where each element has a name associated with it. This is similar to `NamedDomainObjectContainer`, however a `NamedDomainObjectList` doesn't manage the objects in the collection. They need to be created and added manually.

You can create an instance using the ObjectFactory.namedDomainObjectList() method.

DomainObjectSet

A DomainObjectSet simply holds a set of configurable objects. Compared to `NamedDomainObjectContainer`, a `DomainObjectSet` doesn't manage the objects in the collection. They need to be created and added manually.

You can create an instance using the ObjectFactory.domainObjectSet() method.

Gradle Plugin Development Plugin

The Java Gradle Plugin development plugin can be used to assist in the development of Gradle plugins. It automatically applies the Java Library (`java-library`) plugin, adds the `gradleApi()` dependency to the `api` configuration and performs validation of plugin metadata during `jar` task execution.

The plugin also integrates with TestKit, a library that aids in writing and executing functional tests for plugin code. It automatically adds the `gradleTestKit()` dependency to the `testImplementation` configuration and generates a plugin classpath manifest file consumed by a `GradleRunner` instance if found. Please refer to Automatic classpath injection with the Plugin Development Plugin for more on its usage, configuration options and samples.

Usage

To use the Java Gradle Plugin Development plugin, include the following in your build script:

Example 562. Using the Java Gradle Plugin Development plugin

build.gradle

```
plugins {
    id 'java-gradle-plugin'
}
```

build.gradle.kts

```
plugins {
    `java-gradle-plugin`
}
```

Applying the plugin automatically applies the Java Library(`java-library`) plugin and adds the `gradleApi()` dependency to the `api` configuration. It also adds some validations to the build.

The following validations are performed:

- There is a plugin descriptor defined for the plugin.
- The plugin descriptor contains an `implementation-class` property.
- The `implementation-class` property references a valid class file in the jar.
- Each property getter or the corresponding field must be annotated with a property annotation like `@InputFile` and `@OutputDirectory`. Properties that don't participate in up-to-date checks should be annotated with `@Internal`.

Any failed validations will result in a warning message.

For each plugin you are developing, add an entry to the `gradlePlugin {}` script block:

Example 563. Using the gradlePlugin {} block.

build.gradle

```
gradlePlugin {
    plugins {
        simplePlugin {
            id = 'org.gradle.sample.simple-plugin'
            implementationClass = 'org.gradle.sample.SimplePlugin'
        }
    }
}
```

build.gradle.kts

```
gradlePlugin {
    plugins {
        create("simplePlugin") {
            id = "org.gradle.sample.simple-plugin"
            implementationClass = "org.gradle.sample.SimplePlugin"
        }
    }
}
```

The `gradlePlugin {}` block defines the plugins being built by the project including the `id` and `implementationClass` of the plugin. From this data about the plugins being developed, Gradle can automatically:

- Generate the plugin descriptor in the `jar` file's `META-INF` directory.
- Configure the Plugin Marker Artifact publications (Maven or Ivy) for each plugin.
- Publish each plugin to the Gradle Plugin Portal (see Publishing Plugins to Gradle Plugin Portal for details), but only if the Plugin Publishing Plugin has also been applied.

Interactions

Some of the plugin's behaviour depends on other, related plugins also being applied in your build, namely the Maven Publish (`maven-publish`) and Ivy Publish (`ivy-publish`) plugins.

Other plugins auto apply the Java Gradle Plugin, like the Plugin Publishing Plugin.

Maven Publish Plugin

When the Java Gradle Plugin (`java-gradle-plugin`) detects that the Maven Publish Plugin (`maven-publish`) is also applied by the build, it will automatically configure the following

MavenPublications:

- a single "main" publication, named `pluginMaven`, based on the main Java component
- multiple "marker" publications (one for each plugin defined in the `gradlePlugin {}` block), named `<pluginName>PluginMarkerMaven` (for example in the above example it would be `simplePluginPluginMarkerMaven`)

This automatic configuration happens in a Project.afterEvaluate() block (so at the end of the build configuration phase), and only if these publications haven't already been defined, so it's possible to create and customise them during the earlier stages of build configuration.

Ivy Publish Plugin

When the Java Gradle Plugin(`java-gradle-plugin`) detects that the Ivy Publish Plugin (`ivy-publish`) is also applied by the build, it will automatically configure the following IvyPublications:

- a single "main" publication, named `pluginIvy`, based on the main Java component
- multiple "marker" publications (one for each plugin defined in the `gradlePlugin {}` block), named `<pluginName>PluginMarkerIvy` (for example in the above example it would be `simplePluginPluginMarkerIvy`)

This automatic configuration happens in a Project.afterEvaluate() block (so at the end of the build configuration phase), and only if these publications haven't already been defined, so it's possible to create and customise them during the earlier stages of build configuration.

Plugin Publish Plugin

Starting from version 1.0.0, the Plugin Publish Plugin always auto-applies the Java Gradle Plugin (`java-gradle-plugin`) and the Maven Publish Plugin (`maven-publish`).

Reference

A Groovy Build Script Primer

Ideally, a Groovy build script looks mostly like configuration: setting some properties of the project, configuring dependencies, declaring tasks, and so on. That configuration is based on Groovy language constructs. This primer aims to explain what those constructs are and — most importantly — how they relate to Gradle's API documentation.

The `Project` object

As Groovy is an object-oriented language based on Java, its properties and methods apply to objects. In some cases, the object is implicit — particularly at the top level of a build script, i.e. not nested inside a `{}` block.

Consider this fragment of build script, which contains an unqualified property and block:

```
version = '1.0.0.GA'

configurations {
    ...
}
```

Both `version` and `configurations {}` are part of org.gradle.api.Project.

This example reflects how every Groovy build script is backed by an implicit instance of `Project`. If you see an unqualified element and you don't know where it's defined, always check the `Project` API documentation to see if that's where it's coming from.

CAUTION	Avoid using Groovy MetaClass programming techniques in your build scripts. Gradle provides its own API for adding dynamic runtime properties. Use of Groovy-specific metaprogramming can cause builds to retain large amounts of memory between builds that will eventually cause the Gradle daemon to run out-of-memory.

Properties

```
<obj>.<name>                // Get a property value
<obj>.<name> = <value>      // Set a property to a new value
"$<name>"                   // Embed a property value in a string
"${<obj>.<name>}"           // Same as previous (embedded value)
```

Examples

```
version = '1.0.1'
myCopyTask.description = 'Copies some files'

file("$buildDir/classes")
println "Destination: ${myCopyTask.destinationDir}"
```

A property represents some state of an object. The presence of an `=` sign is a clear indicator that you're looking at a property. Otherwise, a qualified name — it begins with `<obj>.` — without any other decoration is also a property.

If the name is unqualified, then it may be one of the following:

- A task instance with that name.
- A property on Project.
- An extra property defined elsewhere in the project.
- A property of an implicit object within a block.
- A local variable defined earlier in the build script.

Note that plugins can add their own properties to the `Project` object. The API documentation lists all the properties added by core plugins. If you're struggling to find where a property comes from, check the documentation for the plugins that the build uses.

TIP	When referencing a project property in your build script that is added by a non-core plugin, consider prefixing it with `project.` — it's clear then that the property belongs to the project object.

Properties in the API documentation

The Groovy DSL reference shows properties as they are used in your build scripts, but the Javadocs only display methods. That's because properties are implemented as methods behind the scenes:

- A property can be *read* if there is a method named `get<PropertyName>` with zero arguments that returns the same type as the property.
- A property can be *modified* if there is a method named `set<PropertyName>` with one argument that has the same type as the property and a return type of `void`.

Note that property names usually start with a lower-case letter, but that letter is upper case in the method names. So the getter method `getProjectVersion()` corresponds to the property `projectVersion`. This convention does not apply when the name begins with at least two upper-case letters, in which case there is not change in case. For example, `getRAM()` corresponds to the property `RAM`.

Examples

```
project.getVersion()
project.version

project.setVersion('1.0.1')
project.version = '1.0.1'
```

Methods

```
<obj>.<name>()              // Method call with no arguments
<obj>.<name>(<arg>, <arg>)  // Method call with multiple arguments
<obj>.<name> <arg>, <arg>   // Method call with multiple args (no parentheses)
```

Examples

```
myCopyTask.include '**/*.xml', '**/*.properties'

ext.resourceSpec = copySpec()   // `copySpec()` comes from `Project`

file('src/main/java')
println 'Hello, World!'
```

A method represents some behavior of an object, although Gradle often uses methods to configure the state of objects as well. Methods are identifiable by their arguments or empty parentheses. Note that parentheses are sometimes required, such as when a method has zero arguments, so you may find it simplest to always use parentheses.

NOTE	Gradle has a convention whereby if a method has the same name as a collection-based property, then the method *appends* its values to that collection.

Blocks

Blocks are also methods, just with specific types for the last argument.

```
<obj>.<name> {
     ...
}

<obj>.<name>(<arg>, <arg>) {
     ...
}
```

Examples

```
plugins {
    id 'java-library'
}

configurations {
    assets
}

sourceSets {
    main {
        java {
            srcDirs = ['src']
        }
    }
}

dependencies {
    implementation project(':util')
}
```

Blocks are a mechanism for configuring multiple aspects of a build element in one go. They also provide a way to nest configuration, leading to a form of structured data.

There are two important aspects of blocks that you should understand:

1. They are implemented as methods with specific signatures.
2. They can change the target ("delegate") of unqualified methods and properties.

Both are based on Groovy language features and we explain them in the following sections.

Block method signatures

You can easily identify a method as the implementation behind a block by its signature, or more specifically, its argument types. If a method corresponds to a block:

- It must have at least one argument.
- The *last* argument must be of type `groovy.lang.Closure` or org.gradle.api.Action.

For example, Project.copy(Action) matches these requirements, so you can use the syntax:

```
copy {
    into "$buildDir/tmp"
    from 'custom-resources'
}
```

That leads to the question of how `into()` and `from()` work. They're clearly methods, but where would you find them in the API documentation? The answer comes from understanding object

delegation.

Delegation

The section on properties lists where unqualified properties might be found. One common place is on the `Project` object. But there is an alternative source for those unqualified properties and methods inside a block: the block's *delegate object*.

To help explain this concept, consider the last example from the previous section:

```
copy {
    into "$buildDir/tmp"
    from 'custom-resources'
}
```

All the methods and properties in this example are unqualified. You can easily find `copy()` and `buildDir` in the `Project` API documentation, but what about `into()` and `from()`? These are resolved against the delegate of the `copy {}` block. What is the type of that delegate? You'll need to check the API documentation for that.

There are two ways to determine the delegate type, depending on the signature of the block method:

- For `Action` arguments, look at the type's parameter.

  In the example above, the method signature is `copy(Action<? super CopySpec>)` and it's the bit inside the angle brackets that tells you the delegate type — CopySpec in this case.

- For `Closure` arguments, the documentation will explicitly say in the description what type is being configured or what type the delegate it (different terminology for the same thing).

Hence you can find both into() and from() on `CopySpec`. You might even notice that both of those methods have variants that take an `Action` as their last argument, which means you can use block syntax with them.

All new Gradle APIs declare an `Action` argument type rather than `Closure`, which makes it very easy to pick out the delegate type. Even older APIs have an `Action` variant in addition to the old `Closure` one.

Local variables

```
def <name> = <value>        // Untyped variable
<type> <name> = <value>     // Typed variable
```

Examples

```
def i = 1
String errorMsg = 'Failed, because reasons'
```

Local variables are a Groovy construct — unlike extra properties — that can be used to share values within a build script.

CAUTION	Avoid using local variables in the root of the project, i.e. as pseudo project properties. They cannot be read outside of the build script and Gradle has no knowledge of them. Within a narrower context — such as configuring a task — local variables can occasionally be helpful.

Gradle Kotlin DSL Primer

Gradle's Kotlin DSL provides an alternative syntax to the traditional Groovy DSL with an enhanced editing experience in supported IDEs, with superior content assist, refactoring, documentation, and more. This chapter provides details of the main Kotlin DSL constructs and how to use it to interact with the Gradle API.

TIP	If you are interested in migrating an existing Gradle build to the Kotlin DSL, please also check out the dedicated migration section.

Prerequisites

- The embedded Kotlin compiler is known to work on Linux, macOS, Windows, Cygwin, FreeBSD and Solaris on x86-64 architectures.
- Knowledge of Kotlin syntax and basic language features is very helpful. The Kotlin reference documentation and Kotlin Koans will help you to learn the basics.
- Use of the plugins {} block to declare Gradle plugins significantly improves the editing experience and is highly recommended.

IDE support

The Kotlin DSL is fully supported by IntelliJ IDEA and Android Studio. Other IDEs do not yet provide helpful tools for editing Kotlin DSL files, but you can still import Kotlin-DSL-based builds and work with them as usual.

Table 23. IDE support matrix

	Build import	Syntax highlighting [1]	Semantic editor [2]
IntelliJ IDEA	▯	▯	▯
Android Studio	▯	▯	▯
Eclipse IDE	▯	▯	▯
CLion	▯	▯	▯
Apache NetBeans	▯	▯	▯
Visual Studio Code (LSP)	▯	▯	▯

	Build import	Syntax highlighting [1]	Semantic editor [2]
Visual Studio	▯	▯	▯

1 Kotlin syntax highlighting in Gradle Kotlin DSL scripts

2 code completion, navigation to sources, documentation, refactorings etc... in Gradle Kotlin DSL scripts

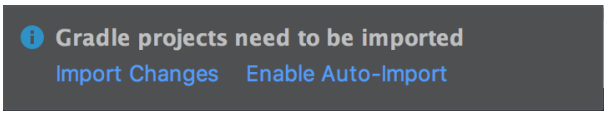
As mentioned in the limitations, you must import your project from the Gradle model to get content-assist and refactoring tools for Kotlin DSL scripts in IntelliJ IDEA.

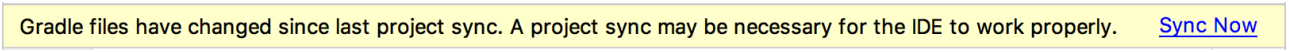
Builds with slow configuration time might affect the IDE responsiveness, so please check out the performance section to help resolve such issues.

Automatic build import vs. automatic reloading of script dependencies

Both IntelliJ IDEA and Android Studio — which is derived from IntelliJ IDEA — will detect when you make changes to your build logic and offer two suggestions:

1. Import the whole build again





2. Reload script dependencies when editing a build script

We recommend that you *disable automatic build import*, but *enable automatic reloading of script dependencies*. That way you get early feedback while editing Gradle scripts and control over when the whole build setup gets synchronized with your IDE.

Troubleshooting

The IDE support is provided by two components:

- The Kotlin Plugin used by IntelliJ IDEA/Android Studio
- Gradle

The level of support varies based on the versions of each.

If you run into trouble, the first thing you should try is running `./gradlew tasks` from the command line to see whether your issue is limited to the IDE. If you encounter the same problem from the command line, then the issue is with the build rather than the IDE integration.

If you can run the build successfully from the command line but your script editor is complaining, then you should try restarting your IDE and invalidating its caches.

If the above doesn't work and you suspect an issue with the Kotlin DSL script editor, you can:

- Run `./gradle tasks` to get more details
- Check the logs in one of these locations:
  - `$HOME/Library/Logs/gradle-kotlin-dsl` on Mac OS X
  - `$HOME/.gradle-kotlin-dsl/log` on Linux
  - `$HOME/AppData/Local/gradle-kotlin-dsl/log` on Windows
- Open an issue on the Gradle issue tracker, including as much detail as you can.

From version 5.1 onwards, the log directory is cleaned up automatically. It is checked periodically (at most every 24 hours) and log files are deleted if they haven't been used for 7 days.

If the above isn't enough to pinpoint the problem, you can enable the `org.gradle.kotlin.dsl.logging.tapi` system property in your IDE. This will cause the Gradle Daemon to log extra information in its log file located in `$HOME/.gradle/daemon`. In IntelliJ IDEA this can be done by opening `Help > Edit Custom VM Options…` and adding `-Dorg.gradle.kotlin.dsl.logging.tapi=true`.

For IDE problems outside of the Kotlin DSL script editor, please open issues in the corresponding IDE's issue tracker:

- JetBrains's IDEA issue tracker,
- Google's Android Studio issue tracker.

Lastly, if you face problems with Gradle itself or with the Kotlin DSL, please open issues on the Gradle issue tracker.

Kotlin DSL scripts

Just like the Groovy-based equivalent, the Kotlin DSL is implemented on top of Gradle's Java API. Everything you can read in a Kotlin DSL script is Kotlin code compiled and executed by Gradle. Many of the objects, functions and properties you use in your build scripts come from the Gradle API and the APIs of the applied plugins.

Script file names

NOTE	Groovy DSL script files use the `.gradle` file name extension. Kotlin DSL script files use the `.gradle.kts` file name extension.

To activate the Kotlin DSL, simply use the `.gradle.kts` extension for your build scripts in place of `.gradle`. That also applies to the settings file — for example `settings.gradle.kts` — and initialization scripts.

Note that you can mix Groovy DSL build scripts with Kotlin DSL ones, i.e. a Kotlin DSL build script can apply a Groovy DSL one and each project in a multi-project build can use either one.

We recommend that you apply the following conventions to get better IDE support:

- Name settings scripts (or any script that is backed by a Gradle `Settings` object) according to the

pattern `*.settings.gradle.kts` — this includes script plugins that are applied from settings scripts

- Name initialization scripts according to the pattern `*.init.gradle.kts` or simply `init.gradle.kts`.

This is so that the IDE knows what type of object "backs" the script, be it Project, Settings or Gradle.

Implicit imports

All Kotlin DSL build scripts have implicit imports consisting of:

- The default Gradle API imports
- The Kotlin DSL API, which is all types within the `org.gradle.kotlin.dsl` and `org.gradle.kotlin.dsl.plugins.dsl` packages currently

CAUTION	*Avoid using internal Kotlin DSL APIs* Use of internal Kotlin DSL APIs in plugins and build scripts has the potential to break builds when either Gradle or plugins change. The Kotlin DSL API extends the Gradle public API with the types listed in the corresponding API docs that are in the `org.gradle.kotlin.dsl` or `org.gradle.kotlin.dsl.plugins.dsl` packages (but not subpackages of those).

Type-safe model accessors

The Groovy DSL allows you to reference many elements of the build model by name, even when they are defined at runtime. Think named configurations, named source sets, and so on. For example, you can get hold of the `implementation` configuration via `configurations.implementation`.

The Kotlin DSL replaces such dynamic resolution with type-safe model accessors that work with model elements contributed by plugins.

Understanding when type-safe model accessors are available

The Kotlin DSL currently supports type-safe model accessors for any of the following that are contributed by plugins:

- Dependency and artifact configurations (such as `implementation` and `runtimeOnly` contributed by the Java Plugin)
- Project extensions and conventions (such as `sourceSets`)
- Extensions on the `dependencies` and `repositories` containers
- Elements in the `tasks` and `configurations` containers
- Elements in project-extension containers (for example the source sets contributed by the Java Plugin that are added to the `sourceSets` container)
- Extensions on each of the above

IMPORTANT	Only the main project build scripts and precompiled project script plugins have type-safe model accessors. Initialization scripts, settings scripts, script plugins do not. These limitations will be removed in a future Gradle release.

The set of type-safe model accessors available is calculated right before evaluating the script body, immediately after the `plugins {}` block. Any model elements contributed after that point do not work with type-safe model accessors. For example, this includes any configurations you might define in your own build script. However, this approach does mean that you can use type-safe accessors for any model elements that are contributed by plugins that are *applied by parent projects*.

The following project build script demonstrates how you can access various configurations, extensions and other elements using type-safe accessors:

Example 564. Using type-safe model accessors

build.gradle.kts

```
plugins {
    `java-library`
}

dependencies {                                  ①
    api("junit:junit:4.13")
    implementation("junit:junit:4.13")
    testImplementation("junit:junit:4.13")
}

configurations {                                ①
    implementation {
        resolutionStrategy.failOnVersionConflict()
    }
}

sourceSets {                                    ②
    main {                                      ③
        java.srcDir("src/core/java")
    }
}

java {                                          ④
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}

tasks {
    test {                                      ⑤
        testLogging.showExceptions = true
    }
}
```

① Uses type-safe accessors for the `api`, `implementation` and `testImplementation` dependency configurations contributed by the Java Library Plugin

② Uses an accessor to configure the `sourceSets` project extension

③ Uses an accessor to configure the `main` source set

④ Uses an accessor to configure the `java` source for the `main` source set

⑤ Uses an accessor to configure the `test` task

TIP	Your IDE knows about the type-safe accessors, so it will include them in its suggestions. This will happen both at the top level of your build scripts — most plugin extensions are added to the `Project` object — and within the blocks that configure an extension.

Note that accessors for elements of containers such as `configurations`, `tasks` and `sourceSets` leverage Gradle's configuration avoidance APIs. For example, on `tasks` they are of type `TaskProvider<T>` and provide a lazy reference and lazy configuration of the underlying task. Here are some examples that illustrate the situations in which configuration avoidance applies:

```
tasks.test {
    // lazy configuration
}

// Lazy reference
val testProvider: TaskProvider<Test> = tasks.test

testProvider {
    // lazy configuration
}

// Eagerly realized Test task, defeat configuration avoidance if done out of a lazy
context
val test: Test = tasks.test.get()
```

For all other containers than `tasks`, accessors for elements are of type `NamedDomainObjectProvider<T>` and provide the same behavior.

Understanding what to do when type-safe model accessors are not available

Consider the sample build script shown above that demonstrates the use of type-safe accessors. The following sample is exactly the same except that is uses the `apply()` method to apply the plugin. The build script can not use type-safe accessors in this case because the `apply()` call happens in the body of the build script. You have to use other techniques instead, as demonstrated here:

Example 565. Configuring plugins without type-safe accessors

build.gradle.kts

```
apply(plugin = "java-library")

dependencies {
    "api"("junit:junit:4.13")
    "implementation"("junit:junit:4.13")
    "testImplementation"("junit:junit:4.13")
}

configurations {
    "implementation" {
        resolutionStrategy.failOnVersionConflict()
    }
}

configure<SourceSetContainer> {
    named("main") {
        java.srcDir("src/core/java")
    }
}

configure<JavaPluginConvention> {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}

tasks {
    named<Test>("test") {
        testLogging.showExceptions = true
    }
}
```

Type-safe accessors are unavailable for model elements contributed by the following:

- Plugins applied via the `apply(plugin = "id")` method
- The project build script
- Script plugins, via `apply(from = "script-plugin.gradle.kts")`
- Plugins applied via cross-project configuration

You also can not use type-safe accessors in Binary Gradle plugins implemented in Kotlin.

If you can't find a type-safe accessor, *fall back to using the normal API* for the corresponding types. To do that, you need to know the names and/or types of the configured model elements. We'll now show you how those can be discovered by looking at the above script in detail.

Artifact configurations

The following sample demonstrates how to reference and configure artifact configurations without type accessors:

Example 566. Artifact configurations

build.gradle.kts

```
apply(plugin = "java-library")

dependencies {
    "api"("junit:junit:4.13")
    "implementation"("junit:junit:4.13")
    "testImplementation"("junit:junit:4.13")
}

configurations {
    "implementation" {
        resolutionStrategy.failOnVersionConflict()
    }
}
```

The code looks similar to that for the type-safe accessors, except that the configuration names are string literals in this case. You can use string literals for configuration names in dependency declarations and within the `configurations {}` block.

The IDE won't be able to help you discover the available configurations in this situation, but you can look them up either in the corresponding plugin's documentation or by running `gradle dependencies`.

Project extensions and conventions

Project extensions and conventions have both a name and a unique type, but the Kotlin DSL only needs to know the type in order to configure them. As the following sample shows for the `sourceSets {}` and `java {}` blocks from the original example build script, you can use the `configure<T>()` function with the corresponding type to do that:

Example 567. Project extensions and conventions

build.gradle.kts

```
apply(plugin = "java-library")

configure<SourceSetContainer> {
    named("main") {
        java.srcDir("src/core/java")
    }
}

configure<JavaPluginConvention> {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}
```

Note that `sourceSets` is a Gradle extension on `Project` of type `SourceSetContainer` and `java` is an extension on `Project` of type `JavaPluginExtension`.

You can discover what extensions and conventions are available either by looking at the documentation for the applied plugins or by running `gradle kotlinDslAccessorsReport`, which prints the Kotlin code necessary to access the model elements contributed by all the applied plugins. The report provides both names and types. As a last resort, you can also check a plugin's source code, but that shouldn't be necessary in the majority of cases.

Note that you can also use the `the<T>()` function if you only need a reference to the extension or convention without configuring it, or if you want to perform a one-line configuration, like so:

```
the<SourceSetContainer>()["main"].srcDir("src/core/java")
```

The snippet above also demonstrates one way of configuring the elements of a project extension that is a container.

Elements in project-extension containers

Container-based project extensions, such as `SourceSetContainer`, also allow you to configure the elements held by them. In our sample build script, we want to configure a source set named `main` within the source set container, which we can do by using the named() method in place of an accessor, like so:

Example 568. Elements of project extensions that are containers

build.gradle.kts

```
apply(plugin = "java-library")

configure<SourceSetContainer> {
    named("main") {
        java.srcDir("src/core/java")
    }
}
```

All elements within a container-based project extension have a name, so you can use this technique in all such cases.

As for project extensions and conventions themselves, you can discover what elements are present in any container by either looking at the documentation of the applied plugins or by running `gradle kotlinDslAccessorsReport`. And as a last resort, you may be able to view the plugin's source code to find out what it does, but that shouldn't be necessary in the majority of cases.

Tasks

Tasks are not managed through a container-based project extension, but they are part of a container that behaves in a similar way. This means that you can configure tasks in the same way as you do for source sets, as you can see in this example:

Example 569. Tasks

build.gradle.kts

```
apply(plugin = "java-library")

tasks {
    named<Test>("test") {
        testLogging.showExceptions = true
    }
}
```

We are using the Gradle API to refer to the tasks by name and type, rather than using accessors. Note that it's necessary to specify the type of the task explicitly, otherwise the script won't compile because the inferred type will be `Task`, not `Test`, and the `testLogging` property is specific to the `Test` task type. You can, however, omit the type if you only need to configure properties or to call methods that are common to all tasks, i.e. they are declared on the `Task` interface.

One can discover what tasks are available by running `gradle tasks`. You can then find out the type of a given task by running `gradle help --task <taskName>`, as demonstrated here:

```
 ./gradlew help --task test
...
Type
     Test (org.gradle.api.tasks.testing.Test)
```

Note that the IDE can assist you with the required imports, so you only need the simple names of the types, i.e. without the package name part. In this case, there's no need to import the `Test` task type as it is part of the Gradle API and is therefore imported implicitly.

About conventions

Some of the Gradle core plugins expose configurability with the help of a so-called *convention* object. These serve a similar purpose to — and have now been superseded by — *extensions*. Please avoid using convention objects when writing new plugins. The long term plan is to migrate all Gradle core plugins to use extensions and remove the convention objects altogether.

As seen above, the Kotlin DSL provides accessors only for convention objects on `Project`. There are situations that require you to interact with a Gradle plugin that uses convention objects on other types. The Kotlin DSL provides the `withConvention(T::class) {}` extension function to do this:

Example 570. Configuring source set conventions

build.gradle.kts

```
plugins {
    groovy
}

sourceSets {
    main {
        withConvention(GroovySourceSet::class) {
            groovy.srcDir("src/core/groovy")
        }
    }
}
```

This technique is most commonly required for source sets that are added by language plugins other than the Java Plugin, e.g. the Groovy Plugin and the Scala Plugin. You can see which plugins add which properties to source sets in the SourceSet reference documentation.

Multi-project builds

As with single-project builds, you should try to use the `plugins {}` block in your multi-project builds

so that you can use the type-safe accessors. Another consideration with multi-project builds is that you won't be able to use type-safe accessors when configuring subprojects within the root build script or with other forms of cross configuration between projects. We discuss both topics in more detail in the following sections.

Applying plugins

You can declare your plugins within the subprojects to which they apply, but we recommend that you also declare them within the root project build script. This makes it easier to keep plugin versions consistent across projects within a build. The approach also improves the performance of the build.

The Using Gradle plugins chapter explains how you can declare plugins in the root project build script with a version and then apply them to the appropriate subprojects' build scripts. What follows is an example of this approach using three subprojects and three plugins. Note how the root build script only declares the community plugins as the Java Library Plugin is tied to the version of Gradle you are using:

Example 571. Declare plugin dependencies in the root build script using the `plugins {}` block

settings.gradle.kts

```
rootProject.name = "multi-project-build"
include("domain", "infra", "http")
```

build.gradle.kts

```
plugins {
    id("com.github.johnrengelman.shadow") version "4.0.1" apply false
    id("io.ratpack.ratpack-java") version "1.8.2" apply false
}
```

domain/build.gradle.kts

```
plugins {
    `java-library`
}

dependencies {
    api("javax.measure:unit-api:1.0")
    implementation("tec.units:unit-ri:1.0.3")
}
```

infra/build.gradle.kts

```
plugins {
    `java-library`
    id("com.github.johnrengelman.shadow")
}

shadow {
    applicationDistribution.from("src/dist")
}

tasks.shadowJar {
    minimize()
}
```

http/build.gradle.kts

```
plugins {
    java
    id("io.ratpack.ratpack-java")
}

dependencies {
    implementation(project(":domain"))
    implementation(project(":infra"))
    implementation(ratpack.dependency("dropwizard-metrics"))
}

application {
    mainClass.set("example.App")
}

ratpack.baseDir = file("src/ratpack/baseDir")
```

If your build requires additional plugin repositories on top of the Gradle Plugin Portal, you should declare them in the `pluginManagement {}` block in your `settings.gradle.kts` file, like so:

Example 572. Declare additional plugin repositories

settings.gradle.kts

```
pluginManagement {
    repositories {
        mavenCentral()
        gradlePluginPortal()
    }
}
```

Plugins fetched from a source other than the Gradle Plugin Portal can only be declared via the `plugins {}` block if they are published with their plugin marker artifacts.

NOTE At the time of writing, all versions of the Android Plugin for Gradle up to 3.2.0 present in the `google()` repository lack plugin marker artifacts.

If those artifacts are missing, then you can't use the `plugins {}` block. You must instead fall back to declaring your plugin dependencies using the `buildscript {}` block in the root project build script. Here's an example of doing that for the Android Plugin:

Example 573. Declare plugin dependencies in the root build script using the `buildscript {}` block

settings.gradle.kts

```
include("lib", "app")
```

build.gradle.kts

```
buildscript {
    repositories {
        google()
        gradlePluginPortal()
    }
    dependencies {
        classpath("com.android.tools.build:gradle:4.1.2")
    }
}
```

lib/build.gradle.kts

```
plugins {
    id("com.android.library")
}

android {
    // ...
}
```

app/build.gradle.kts

```
plugins {
    id("com.android.application")
}

android {
    // ...
}
```

This technique is not that different from what Android Studio produces when creating a new build. The main difference is that the subprojects' build scripts in the above sample declare their plugins using the `plugins {}` block. This means that you can use type-safe accessors for the model elements that they contribute.

Note that you can't use this technique if you want to apply such a plugin either to the root project build script of a multi-project build (rather than solely to its subprojects) or to a single-project build. You'll need to use a different approach in those cases that we detail in another section.

Cross-configuring projects

Cross project configuration is a mechanism by which you can configure a project from another project's build script. A common example is when you configure subprojects in the root project build script.

Taking this approach means that you won't be able to use type-safe accessors for model elements contributed by the plugins. You will instead have to rely on string literals and the standard Gradle APIs.

As an example, let's modify the Java/Ratpack sample build to fully configure its subprojects from the root project build script:

Example 574. Cross-configuring projects

settings.gradle.kts

```
rootProject.name = "multi-project-build"
include("domain", "infra", "http")
```

build.gradle.kts

```
import com.github.jengelman.gradle.plugins.shadow.ShadowExtension
import com.github.jengelman.gradle.plugins.shadow.tasks.ShadowJar
import ratpack.gradle.RatpackExtension

plugins {
    id("com.github.johnrengelman.shadow") version "4.0.1" apply false
    id("io.ratpack.ratpack-java") version "1.8.2" apply false
}

project(":domain") {
    apply(plugin = "java-library")
    dependencies {
        "api"("javax.measure:unit-api:1.0")
        "implementation"("tec.units:unit-ri:1.0.3")
    }
}

project(":infra") {
    apply(plugin = "java-library")
    apply(plugin = "com.github.johnrengelman.shadow")
    configure<ShadowExtension> {
        applicationDistribution.from("src/dist")
    }
    tasks.named<ShadowJar>("shadowJar") {
        minimize()
    }
}

project(":http") {
    apply(plugin = "java")
    apply(plugin = "io.ratpack.ratpack-java")
    repositories { mavenCentral() }
    val ratpack = the<RatpackExtension>()
    dependencies {
        "implementation"(project(":domain"))
        "implementation"(project(":infra"))
        "implementation"(ratpack.dependency("dropwizard-metrics"))
        "runtimeOnly"("org.slf4j:slf4j-simple:1.7.25")
    }
    configure<JavaApplication> {
        mainClass.set("example.App")
    }
    ratpack.baseDir = file("src/ratpack/baseDir")
}
```

Note how we're using the `apply()` method to apply the plugins since the `plugins {}` block doesn't work in this context. We are also using standard APIs instead of type-safe accessors to configure

tasks, extensions and conventions — an approach that we discussed in more detail elsewhere.

When you can't use the `plugins {}` block

Plugins fetched from a source other than the Gradle Plugin Portal may or may not be usable with the `plugins {}` block. It depends on how they have been published and, specifically, whether they have been published with the necessary plugin marker artifacts.

For example, the Android Plugin for Gradle is not published to the Gradle Plugin Portal and — at least up to version 3.2.0 of the plugin — the metadata required to resolve the artifacts for a given plugin identifier is not published to the Google repository.

If your build is a multi-project build and you don't need to apply such a plugin to your *root* project, then you can get round this issue using the technique described above. For any other situation, keep reading.

> **TIP** When publishing plugins, please use Gradle's built-in Gradle Plugin Development Plugin. It automates the publication of the metadata necessary to make your plugins usable with the `plugins {}` block.

We will show you in this section how to apply the Android Plugin to a single-project build or the root project of a multi-project build. The goal is to instruct your build on how to map the `com.android.application` plugin identifier to a resolvable artifact. This is done in two steps:

- Add a plugin repository to the build's settings script
- Map the plugin ID to the corresponding artifact coordinates

You accomplish both steps by configuring a `pluginManagement {}` block in the build's settings script. To demonstrate, the following sample adds the `google()` repository — where the Android plugin is published — to the repository search list, and uses a `resolutionStrategy {}` block to map the `com.android.application` plugin ID to the `com.android.tools.build:gradle:<version>` artifact available in the `google()` repository:

Example 575. Mapping plugin IDs to dependency coordinates

settings.gradle.kts

```
pluginManagement {
    repositories {
        google()
        gradlePluginPortal()
    }
    resolutionStrategy {
        eachPlugin {
            if(requested.id.namespace == "com.android") {

useModule("com.android.tools.build:gradle:${requested.version}")
            }
        }
    }
}
```

build.gradle.kts

```
plugins {
    id("com.android.application") version "4.1.2"
}

android {
    // ...
}
```

In fact, the above sample will work for all `com.android.*` plugins that are provided by the specified module. That's because the packaged module contains the details of which plugin ID maps to which plugin implementation class, using the properties-file mechanism described in the Writing Custom Plugins chapter.

See the Plugin Management section of the Gradle user manual for more information on the `pluginManagement {}` block and what it can be used for.

Working with container objects

The Gradle build model makes heavy use of container objects (or just "containers"). For example, both `configurations` and `tasks` are container objects that contain `Configuration` and `Task` objects respectively. Community plugins also contribute containers, like the `android.buildTypes` container contributed by the Android Plugin.

The Kotlin DSL provides several ways for build authors to interact with containers. We look at each of those ways next, using the `tasks` container as an example.

TIP	Note that you can leverage the type-safe accessors described in another section if you are configuring existing elements on supported containers. That section also describes which containers support type-safe accessors.

Using the container API

All containers in Gradle implement NamedDomainObjectContainer<DomainObjectType>. Some of them can contain objects of different types and implement PolymorphicDomainObjectContainer<BaseType>. The simplest way to interact with containers is through these interfaces.

The following sample demonstrates how you can use the named() method to configure existing tasks and the register() method to create new ones.

Example 576. Using the container API

build.gradle.kts

```
tasks.named("check")                    ①
tasks.register("myTask1")               ②

tasks.named<JavaCompile>("compileJava") ③
tasks.register<Copy>("myCopy1")         ④

tasks.named("assemble") {               ⑤
    dependsOn(":myTask1")
}
tasks.register("myTask2") {             ⑥
    description = "Some meaningful words"
}

tasks.named<Test>("test") {             ⑦
    testLogging.showStackTraces = true
}
tasks.register<Copy>("myCopy2") {       ⑧
    from("source")
    into("destination")
}
```

① Gets a reference of type `Task` to the existing task named `check`

② Registers a new untyped task named `myTask1`

③ Gets a reference to the existing task named `compileJava` of type `JavaCompile`

④ Registers a new task named `myCopy1` of type `Copy`

⑤ Gets a reference to the existing (untyped) task named `assemble` and configures it — you can only configure properties and methods that are available on `Task` with this syntax

⑥ Registers a new untyped task named `myTask2` and configures it — you can only configure properties and methods that are available on `Task` in this case

⑦ Gets a reference to the existing task named `test` of type `Test` and configures it — in this case you have access to the properties and methods of the specified type

⑧ Registers a new task named `myCopy2` of type `Copy` and configures it

NOTE	The above sample relies on the configuration avoidance APIs. If you need or want to eagerly configure or register container elements, simply replace `named()` with `getByName()` and `register()` with `create()`.

Using Kotlin delegated properties

Another way to interact with containers is via Kotlin delegated properties. These are particularly useful if you need a reference to a container element that you can use elsewhere in the build. In addition, Kotlin delegated properties can easily be renamed via IDE refactoring.

The following sample does the exact same things as the one in the previous section, but it uses delegated properties and reuses those references in place of string-literal task paths:

Example 577. Using Kotlin delegated properties

build.gradle.kts

```
val check by tasks.existing
val myTask1 by tasks.registering

val compileJava by tasks.existing(JavaCompile::class)
val myCopy1 by tasks.registering(Copy::class)

val assemble by tasks.existing {
    dependsOn(myTask1)  ①
}
val myTask2 by tasks.registering {
    description = "Some meaningful words"
}

val test by tasks.existing(Test::class) {
    testLogging.showStackTraces = true
}
val myCopy2 by tasks.registering(Copy::class) {
    from("source")
    into("destination")
}
```

① Uses the reference to the `myTask1` task rather than a task path

NOTE	The above rely on configuration avoidance APIs. If you need to eagerly configure or register container elements simply replace `existing()` with `getting()` and `registering()` with `creating()`.

Configuring multiple container elements together

When configuring several elements of a container one can group interactions in a block in order to avoid repeating the container's name on each interaction. The following example uses a combination of type-safe accessors, the container API and Kotlin delegated properties:

Example 578. Container scope

build.gradle.kts

```
tasks {
    test {
        testLogging.showStackTraces = true
    }
    val myCheck by registering {
        doLast { /* assert on something meaningful */ }
    }
    check {
        dependsOn(myCheck)
    }
    register("myHelp") {
        doLast { /* do something helpful */ }
    }
}
```

Working with runtime properties

Gradle has two main sources of properties that are defined at runtime: *project properties* and *extra properties*. The Kotlin DSL provides specific syntax for working with these types of properties, which we look at in the following sections.

Project properties

The Kotlin DSL allows you to access project properties by binding them via Kotlin delegated properties. Here's a sample snippet that demonstrates the technique for a couple of project properties, one of which *must* be defined:

build.gradle.kts

```
val myProperty: String by project  ①
val myNullableProperty: String? by project ②
```

① Makes the `myProperty` project property available via a `myProperty` delegated property — the

project property must exist in this case, otherwise the build will fail when the build script attempts to use the `myProperty` value

② Does the same for the `myNullableProperty` project property, but the build won't fail on using the `myNullableProperty` value as long as you check for null (standard Kotlin rules for null safety apply)

The same approach works in both settings and initialization scripts, except you use `by settings` and `by gradle` respectively in place of `by project`.

Extra properties

Extra properties are available on any object that implements the ExtensionAware interface. Kotlin DSL allows you to access extra properties and create new ones via delegated properties, using any of the `by extra` forms demonstrated in the following sample:

build.gradle.kts

```
val myNewProperty by extra("initial value")  ①
val myOtherNewProperty by extra { "calculated initial value" }  ②

val myProperty: String by extra  ③
val myNullableProperty: String? by extra  ④
```

① Creates a new extra property called `myNewProperty` in the current context (the project in this case) and initializes it with the value `"initial value"`, which also determines the property's *type*

② Create a new extra property whose initial value is calculated by the provided lambda

③ Binds an existing extra property from the current context (the project in this case) to a `myProperty` reference

④ Does the same as the previous line but allows the property to have a null value

This approach works for all Gradle scripts: project build scripts, script plugins, settings scripts and initialization scripts.

You can also access extra properties on a root project from a subproject using the following syntax:

my-sub-project/build.gradle.kts

```
val myNewProperty: String by rootProject.extra  ①
```

① Binds the root project's `myNewProperty` extra property to a reference of the same name

Extra properties aren't just limited to projects. For example, `Task` extends `ExtensionAware`, so you can attach extra properties to tasks as well. Here's an example that defines a new `myNewTaskProperty` on the `test` task and then uses that property to initialize another task:

build.gradle.kts

```
tasks {
    test {
        val reportType by extra("dev")  ①
        doLast {
            // Use 'suffix' for post processing of reports
        }
    }

    register<Zip>("archiveTestReports") {
        val reportType: String by test.get().extra  ②
        archiveAppendix.set(reportType)
        from(test.get().reports.html.destination)
    }
}
```

① Creates a new `reportType` extra property on the `test` task

② Makes the `test` task's `reportType` extra property available to configure the `archiveTestReports` task

If you're happy to use eager configuration rather than the configuration avoidance APIs, you could use a single, "global" property for the report type, like this:

build.gradle.kts

```
tasks.test.doLast { ... }

val testReportType by tasks.test.get().extra("dev")  ①

tasks.create<Zip>("archiveTestReports") {
    archiveAppendix.set(testReportType)  ②
    from(test.get().reports.html.destination)
}
```

① Creates and initializes an extra property on the `test` task, binding it to a "global" property

② Uses the "global" property to initialize the `archiveTestReports` task

There is one last syntax for extra properties that we should cover, one that treats `extra` as a map. We recommend against using this in general as you lose the benefits of Kotlin's type checking and it prevents IDEs from providing as much support as they could. However, it is more succinct than the delegated properties syntax and can reasonably be used if you only need to set the value of an extra property without referencing it later.

Here's a simple example demonstrating how to set and read extra properties using the map syntax:

build.gradle.kts

```
extra["myNewProperty"] = "initial value"  ①

tasks.create("myTask") {
    doLast {
        println("Property: ${project.extra["myNewProperty"]}")  ②
    }
}
```

① Creates a new project extra property called `myNewProperty` and sets its value

② Reads the value from the project extra property we created — note the `project.` qualifier on `extra[…]`, otherwise Gradle will assume we want to read an extra property from the *task*

The Kotlin DSL Plugin

The Kotlin DSL Plugin provides a convenient way to develop Kotlin-based projects that contribute build logic. That includes buildSrc projects, included builds and Gradle plugins.

The plugin achieves this by doing the following:

- Applies the Kotlin Plugin, which adds support for compiling Kotlin source files.
- Adds the `kotlin-stdlib-jdk8`, `kotlin-reflect` and `gradleKotlinDsl()` dependencies to the `compileOnly` and `testImplementation` configurations, which allows you to make use of those Kotlin libraries and the Gradle API in your Kotlin code.
- Configures the Kotlin compiler with the same settings that are used for Kotlin DSL scripts, ensuring consistency between your build logic and those scripts.
- Enables support for precompiled script plugins.

CAUTION

Avoid specifying a version for the `kotlin-dsl` plugin

Each Gradle release is meant to be used with a specific version of the `kotlin-dsl` plugin and compatibility between arbitrary Gradle releases and `kotlin-dsl` plugin versions is not guaranteed. Using an unexpected version of the `kotlin-dsl` plugin in a build will emit a warning and can cause hard to diagnose problems.

This is the basic configuration you need to use the plugin:

Example 579. Applying the Kotlin DSL Plugin to a `buildSrc` project

buildSrc/build.gradle.kts

```
plugins {
    `kotlin-dsl`
}

repositories {
    // The org.jetbrains.kotlin.jvm plugin requires a repository
    // where to download the Kotlin compiler dependencies from.
    mavenCentral()
}
```

The embedded Kotlin

Gradle embeds Kotlin in order to provide support for Kotlin-based scripts.

Kotlin versions

Gradle ships with `kotlin-compiler-embeddable` plus matching versions of `kotlin-stdlib` and `kotlin-reflect` libraries. For details see the Kotlin section of Gradle's compatibility matrix. The `kotlin` package from those modules is visible through the Gradle classpath.

The compatibility guarantees provided by Kotlin apply for both backward and forward compatibility.

Backward compatibility

Our approach is to only do backwards-breaking Kotlin upgrades on a major Gradle release. We will always clearly document which Kotlin version we ship and announce upgrade plans before a major release.

Plugin authors who want to stay compatible with older Gradle versions need to limit their API usage to a subset that is compatible with these old versions. It's not really different from any other new API in Gradle. E.g. if we introduce a new API for dependency resolution and a plugin wants to use that API, then they either need to drop support for older Gradle versions or they need to do some clever organization of their code to only execute the new code path on newer versions.

Forward compatibility

The biggest issue is the compatibility between the external `kotlin-gradle-plugin` version and the `kotlin-stdlib` version shipped with Gradle. More generally, between any plugin that transitively depends on `kotlin-stdlib` and its version shipped with Gradle. As long as the combination is compatible everything should work. This will become less of an issue as the language matures.

Kotlin compiler arguments

These are the Kotlin compiler arguments used for compiling Kotlin DSL scripts and Kotlin sources and scripts in a project that has the `kotlin-dsl` plugin applied:

`-jvm-target=1.8`
: Sets the target version of the generated JVM bytecode to `1.8`.

`-Xjsr305=strict`
: Sets up Kotlin's Java interoperability to strictly follow JSR-305 annotations for increased null safety. See Calling Java code from Kotlin in the Kotlin documentation for more information.

Interoperability

When mixing languages in your build logic, you may have to cross language boundaries. An extreme example would be a build that uses tasks and plugins that are implemented in Java, Groovy and Kotlin, while also using both Kotlin DSL and Groovy DSL build scripts.

Quoting the Kotlin reference documentation:

> Kotlin is designed with Java Interoperability in mind. Existing Java code can be called from Kotlin in a natural way, and Kotlin code can be used from Java rather smoothly as well.

Both calling Java from Kotlin and calling Kotlin from Java are very well covered in the Kotlin reference documentation.

The same mostly applies to interoperability with Groovy code. In addition, the Kotlin DSL provides several ways to opt into Groovy semantics, which we look at next.

Static extensions

Both the Groovy and Kotlin languages support extending existing classes via Groovy Extension modules and Kotlin extensions.

To call a Kotlin extension function from Groovy, call it as a static function, passing the receiver as the first parameter:

Example 580. Calling a Kotlin extension from Groovy

build.gradle

```
TheTargetTypeKt.kotlinExtensionFunction(receiver, "parameters", 42,
aReference)
```

Kotlin extension functions are package-level functions and you can learn how to locate the name of the type declaring a given Kotlin extension in the Package-Level Functions section of the Kotlin

reference documentation.

To call a Groovy extension method from Kotlin, the same approach applies: call it as a static function passing the receiver as the first parameter. Here's an example:

Example 581. Calling a Groovy extension from Kotlin

build.gradle.kts

```
TheTargetTypeGroovyExtension.groovyExtensionMethod(receiver, "parameters",
42, aReference)
```

Named parameters and default arguments

Both the Groovy and Kotlin languages support named function parameters and default arguments, although they are implemented very differently. Kotlin has fully-fledged support for both, as described in the Kotlin language reference under named arguments and default arguments. Groovy implements named arguments in a non-type-safe way based on a `Map<String, ?>` parameter, which means they cannot be combined with default arguments. In other words, you can only use one or the other in Groovy for any given method.

Calling Kotlin from Groovy

To call a Kotlin function that has named arguments from Groovy, just use a normal method call with positional parameters. There is no way to provide values by argument name.

To call a Kotlin function that has default arguments from Groovy, always pass values for all the function parameters.

Calling Groovy from Kotlin

To call a Groovy function with named arguments from Kotlin, you need to pass a `Map<String, ?>`, as shown in this example:

Example 582. Call Groovy function with named arguments from Kotlin

build.gradle.kts

```
groovyNamedArgumentTakingMethod(mapOf(
    "parameterName" to "value",
    "other" to 42,
    "and" to aReference))
```

To call a Groovy function with default arguments from Kotlin, always pass values for all the parameters.

Groovy closures from Kotlin

You may sometimes have to call Groovy methods that take Closure arguments from Kotlin code. For example, some third-party plugins written in Groovy expect closure arguments.

NOTE	Gradle plugins written in any language should prefer the type `Action<T>` type in place of closures. Groovy closures and Kotlin lambdas are automatically mapped to arguments of that type.

In order to provide a way to construct closures while preserving Kotlin's strong typing, two helper methods exist:

- `closureOf<T> {}`
- `delegateClosureOf<T> {}`

Both methods are useful in different circumstances and depend upon the method you are passing the `Closure` instance into.

Some plugins expect simple closures, as with the Bintray plugin:

Example 583. Use `closureOf<T> {}`

```
bintray { pkg(closureOf<PackageConfig> { // Config for the package here }) }
```

In other cases, like with the Gretty Plugin when configuring farms, the plugin expects a delegate closure:

Example 584. Use `delegateClosureOf<T> {}`

build.gradle.kts

```
dependencies {
    implementation("group:artifact:1.2.3") {
        artifact(delegateClosureOf<DependencyArtifact> {
            // configuration for the artifact
            name = "artifact-name"
        })
    }
}
```

There sometimes isn't a good way to tell, from looking at the source code, which version to use. Usually, if you get a `NullPointerException` with `closureOf<T> {}`, using `delegateClosureOf<T> {}` will resolve the problem.

These two utility functions are useful for *configuration closures*, but some plugins might expect Groovy closures for other purposes. The `KotlinClosure0` to `KotlinClosure2` types allows adapting

Kotlin functions to Groovy closures with more flexibility.

Example 585. Use `KotlinClosureX` types

build.gradle.kts

```
somePlugin {

    // Adapt parameter-less function
    takingParameterLessClosure(KotlinClosure0({
        "result"
    }))

    // Adapt unary function
    takingUnaryClosure(KotlinClosure1<String, String>({
        "result from single parameter $this"
    }))

    // Adapt binary function
    takingBinaryClosure(KotlinClosure2<String, String, String>({ a, b ->
        "result from parameters $a and $b"
    }))
}
```

The Kotlin DSL Groovy Builder

If some plugin makes heavy use of Groovy metaprogramming, then using it from Kotlin or Java or any statically-compiled language can be very cumbersome.

The Kotlin DSL provides a `withGroovyBuilder {}` utility extension that attaches the Groovy metaprogramming semantics to objects of type `Any`. The following example demonstrates several features of the method on the object `target`:

Example 586. Use `withGroovyBuilder {}`

build.gradle.kts

```
target.withGroovyBuilder {                                          ①

    // GroovyObject methods available                               ②
    val foo = getProperty("foo")
    setProperty("foo", "bar")
    invokeMethod("name", arrayOf("parameters", 42, aReference))

    // Kotlin DSL utilities
    "name"("parameters", 42, aReference)                            ③
        "blockName" {                                               ④
            // Same Groovy Builder semantics on `blockName`
        }
    "another"("name" to "example", "url" to "https://example.com/") ⑤
}
```

① The receiver is a GroovyObject and provides Kotlin helpers

② The `GroovyObject` API is available

③ Invoke the `methodName` method, passing some parameters

④ Configure the `blockName` property, maps to a `Closure` taking method invocation

⑤ Invoke `another` method taking named arguments, maps to a Groovy named arguments `Map<String, ?>` taking method invocation

Using a Groovy script

Another option when dealing with problematic plugins that assume a Groovy DSL build script is to configure them in a Groovy DSL build script that is applied from the main Kotlin DSL build script:

Example 587. Using a Groovy script

build.gradle.kts

```
plugins {
    id("dynamic-groovy-plugin") version "1.0"              ①
}
apply(from = "dynamic-groovy-plugin-configuration.gradle")  ②
```

dynamic-groovy-plugin-configuration.gradle

```
native {                                                    ③
    dynamic {
        groovy as Usual
    }
}
```

① The Kotlin build script requests and applies the plugin

② The Kotlin build script applies the Groovy script

③ The Groovy script uses dynamic Groovy to configure plugin

Limitations

- The Kotlin DSL is known to be slower than the Groovy DSL on first use, for example with clean checkouts or on ephemeral continuous integration agents. Changing something in the *buildSrc* directory also has an impact as it invalidates build-script caching. The main reason for this is the slower script compilation for Kotlin DSL.
- In IntelliJ IDEA, you must import your project from the Gradle model in order to get content assist and refactoring support for your Kotlin DSL build scripts.
- The Kotlin DSL will not support the `model {}` block, which is part of the discontinued Gradle Software Model.
- We recommend against enabling the incubating configuration on demand feature as it can lead to very hard-to-diagnose problems.

If you run into trouble or discover a suspected bug, please report the issue in the Gradle issue tracker.

Gradle Plugin Reference

This page contains links and short descriptions for all the core plugins provided by Gradle itself.

JVM languages and frameworks

Java

Provides support for building any type of Java project.

Java Library

Provides support for building a Java library.

Java Platform

Provides support for building a Java platform.

Groovy

Provides support for building any type of Groovy project.

Scala

Provides support for building any type of Scala project.

ANTLR

Provides support for generating parsers using ANTLR.

JVM Test Suite

Provides support for modeling and configuring multiple test suite invocations.

Test Report Aggregation

Aggregates the results of multiple Test task invocations (potentially spanning multiple Gradle projects) into a single HTML report.

Native languages

C++ Application

Provides support for building C++ applications on Windows, Linux, and macOS.

C++ Library

Provides support for building C++ libraries on Windows, Linux, and macOS.

C++ Unit Test

Provides support for building and running C++ executable-based tests on Windows, Linux, and macOS.

Swift Application

Provides support for building Swift applications on Linux and macOS.

Swift Library

Provides support for building Swift libraries on Linux and macOS.

XCTest

Provides support for building and running XCTest-based tests on Linux and macOS.

Packaging and distribution

Application

Provides support for building JVM-based, runnable applications.

WAR

Provides support for building and packaging WAR-based Java web applications.

EAR

Provides support for building and packaging Java EE applications.

Maven Publish

Provides support for publishing artifacts to Maven-compatible repositories.

Ivy Publish

Provides support for publishing artifacts to Ivy-compatible repositories.

Distribution

Makes it easy to create ZIP and tarball distributions of your project.

Java Library Distribution

Provides support for creating a ZIP distribution of a Java library project that includes its runtime dependencies.

Code analysis

Checkstyle

Performs quality checks on your project's Java source files using Checkstyle and generates associated reports.

PMD

Performs quality checks on your project's Java source files using PMD and generates associated reports.

JaCoCo

Provides code coverage metrics for your Java project using JaCoCo.

JaCoCo Report Aggregation

Aggregates the results of multiple JaCoCo code coverage reports (potentially spanning multiple Gradle projects) into a single HTML report.

CodeNarc

Performs quality checks on your Groovy source files using CodeNarc and generates associated reports.

IDE integration

Eclipse

Generates Eclipse project files for the build that can be opened by the IDE. This set of plugins can also be used to fine tune Buildship's import process for Gradle builds.

IntelliJ IDEA

Generates IDEA project files for the build that can be opened by the IDE. It can also be used to fine tune IDEA's import process for Gradle builds.

Visual Studio

Generates Visual Studio solution and project files for build that can be opened by the IDE.

Xcode

Generates Xcode workspace and project files for the build that can be opened by the IDE.

Utility

Base

Provides common lifecycle tasks, such as `clean`, and other features common to most builds.

Build Init

Generates a new Gradle build of a specified type, such as a Java library. It can also generate a build script from a Maven POM — see Migrating from Maven to Gradle for more details.

Signing

Provides support for digitally signing generated files and artifacts.

Plugin Development

Makes it easier to develop and publish a Gradle plugin.

Project Report Plugin

Helps to generate reports containing useful information about your build.

Command-Line Interface

The command-line interface is one of the primary methods of interacting with Gradle. The following serves as a reference of executing and customizing Gradle use of a command-line or when writing scripts or configuring continuous integration.

Use of the Gradle Wrapper is highly encouraged. You should substitute `./gradlew` or `gradlew.bat` for `gradle` in all following examples when using the Wrapper.

Executing Gradle on the command-line conforms to the following structure. Options are allowed before and after task names.

```
gradle [taskName...] [--option-name...]
```

If multiple tasks are specified, they should be separated with a space.

Options that accept values can be specified with or without `=` between the option and argument; however, use of `=` is recommended.

```
--console=plain
```

Options that enable behavior have long-form options with inverses specified with `--no-`. The following are opposites.

```
--build-cache
--no-build-cache
```

Many long-form options, have short option equivalents. The following are equivalent:

```
--help
-h
```

NOTE	Many command-line flags can be specified in `gradle.properties` to avoid needing to be typed. See the configuring build environment guide for details.

The following sections describe use of the Gradle command-line interface, grouped roughly by user goal. Some plugins also add their own command line options, for example `--tests` for Java test filtering. For more information on exposing command line options for your own tasks, see Declaring and using command-line options.

Executing tasks

You can learn about what projects and tasks are available in the project reporting section. Most builds support a common set of tasks known as *lifecycle tasks*. These include the `build`, `assemble`, and `check` tasks.

In order to execute a task called "myTask" on the root project, type:

```
$ gradle :myTask
```

This will run the single "myTask" and also all of its task dependencies.

Specify options for a task

To pass an option to a task, prefix the option name with `--` after the task name:

```
$ gradle exampleTask --exampleOption=exampleValue
```

Disambiguate task options from built-in options

Gradle does not prevent tasks from registering options that conflict with Gradle's built-in options, like `--profile` or `--help`. You can disambiguate conflicting task options from Gradle's built-in options with a `--` delimiter before the task name in your command:

```
$ gradle [--built-in-option-name...] -- [taskName...] [--task-option-name...]
```

Consider a task named "mytask" that accepts an option named "profile":

- If you run `gradle mytask --profile`, Gradle accepts `--profile` as the built-in Gradle option.
- If you run `gradle␣–␣mytask --profile=exampleValue`, Gradle passes `--profile` as a task option.

Executing tasks in multi-project builds

In a multi-project build, subproject tasks can be executed with ":" separating subproject name and task name. The following are equivalent *when run from the root project*:

```
$ gradle :my-subproject:taskName
$ gradle my-subproject:taskName
```

You can also run a task for *all* subprojects by using a task *selector* that consists of the task name only. For example, this will run the "test" task for all subprojects when invoked from the root project directory:

```
$ gradle test
```

NOTE	Some tasks selectors, like `help` or `dependencies`, will only run the task on the project they are invoked on and not on all the subprojects. The main motivation for this is that these tasks print out information that would be hard to process if it combined the information from all projects.

When invoking Gradle from within a subproject, the project name should be omitted:

```
$ cd my-subproject
$ gradle taskName
```

NOTE	When executing the Gradle Wrapper from subprojects, one must reference `gradlew` relatively. For example: `../gradlew taskName`. The community gdub project aims to make this more convenient.

Executing multiple tasks

You can also specify multiple tasks. The tasks will be executed as quickly as possible while still honoring task dependencies. Precise order of execution is determined by the tasks' dependencies, and a task having no dependencies may execute earlier than it is listed on the command-line. For example, the following will execute the `test` and `deploy` tasks in the order that they are listed on the command-line and will also execute the dependencies for each task.

```
$ gradle test deploy
```

Command line order safety

Although Gradle will always attempt to execute the build as quickly as possible, command line

ordering safety will also be honored. For example, the following will execute `clean` and `build` along with their dependencies.

```
$ gradle clean build
```

However, the intention implied in the command line order is that `clean` should run first, and then `build`. It would be incorrect to execute `clean` *after* `build`, even if doing so would cause the build to execute faster, since `clean` would remove what `build` created. Conversely, if the command line order was `build` followed by `clean`, it would not be correct to execute `clean` before `build`. Although Gradle will execute the build as quickly as possible, it will also respect the safety of the order of tasks specified on the command line and ensure that `clean` runs before `build` when specified in that order.

Note that command line order safety relies on tasks properly declaring what they create, consume or remove. See up-to-date checks for further information.

Excluding tasks from execution

You can exclude a task from being executed using the `-x` or `--exclude-task` command-line option and providing the name of the task to exclude.

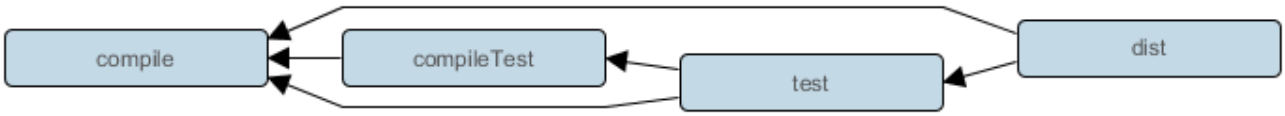


Figure 30. Simple Task Graph

Excluding tasks

```
$ gradle dist --exclude-task test

> Task :compile
compiling source

> Task :dist
building the distribution

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

You can see that the `test` task is not executed, even though it is a dependency of the `dist` task. The `test` task's dependencies such as `compileTest` are not executed either. Those dependencies of `test` that are required by another task, such as `compile`, are still executed.

Forcing tasks to execute

You can force Gradle to execute all tasks ignoring up-to-date checks using the `--rerun-tasks` option:

```
$ gradle test --rerun-tasks
```

This will force `test` and *all* task dependencies of `test` to execute. It's a little like running `gradle clean test`, but without the build's generated output being deleted.

Alternatively, you can tell Gradle to rerun a specific task using the `--rerun` built-in task option.

Continue the build after a task failure

By default, Gradle aborts execution and fails the build when any task fails. This allows the build to complete sooner and prevents cascading failures from obfuscating the root cause of an error. You can use the `--continue` option to force Gradle to execute every task when a failure occurs:

```
$ gradle test --continue
```

When executed with `--continue`, Gradle executes *every* task in the build if all of the dependencies for that task completed without failure. For example, tests do not run if there is a compilation error in the code under test because the test task depends on the compilation task. Gradle outputs each of the encountered failures at the end of the build.

NOTE	Many test suites fail the entire "test" task if any tests fail. Code coverage and reporting tools frequently run after the test task, so this "fail fast" behavior may halt execution before those tools run.

Name abbreviation

When you specify tasks on the command-line, you don't have to provide the full name of the task. You only need to provide enough of the task name to uniquely identify the task. For example, it's likely `gradle che` is enough for Gradle to identify the `check` task.

The same applies for project names. You can execute the `check` task in the `library` subproject with the `gradle lib:che` command.

You can use camel case patterns for more complex abbreviations. These patterns are expanded to match camel case and kebab case names. For example the pattern `foBa` (or even `fB`) matches `fooBar` and `foo-bar`.

More concretely, you can run the `compileTest` task in the `my-awesome-library` subproject with the `gradle mAL:cT` command.

Abbreviated project and task names

```
$ gradle  mAL:cT

> Task :my-awesome-library:compileTest
compiling unit tests

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

You can also use these abbreviations with the `-x` command-line option.

Tracing name expansion

For complex projects, it might not be obvious if the intended tasks were executed. When using abbreviated names, a single typo can lead to the execution of unexpected tasks.

When INFO, or more verbose logging is enabled, the output will contain extra information about the project and task name expansion. For example, when executing the `mAL:cT` command on the previous example, the following log messages will be visible:

```
No exact project with name :mAL has been found. Checking for abbreviated names.
Found exactly one project that matches the abbreviated name :mAL: ':my-awesome-
library'.
No exact task with name :cT has been found. Checking for abbreviated names.
Found exactly one task name, that matches the abbreviated name :cT: ':compileTest'.
```

Common tasks

The following are task conventions applied by built-in and most major Gradle plugins.

Computing all outputs

It is common in Gradle builds for the `build` task to designate assembling all outputs and running all checks.

```
$ gradle build
```

Running applications

It is common for applications to be run with the `run` task, which assembles the application and executes some script or binary.

```
$ gradle run
```

Running all checks

It is common for *all* verification tasks, including tests and linting, to be executed using the `check` task.

```
$ gradle check
```

Cleaning outputs

You can delete the contents of the build directory using the `clean` task, though doing so will cause pre-computed outputs to be lost, causing significant additional build time for the subsequent task execution.

```
$ gradle clean
```

Project reporting

Gradle provides several built-in tasks which show particular details of your build. This can be useful for understanding the structure and dependencies of your build, and for debugging problems.

You can get basic help about available reporting options using `gradle help`.

Listing projects

Running `gradle projects` gives you a list of the sub-projects of the selected project, displayed in a hierarchy.

```
$ gradle projects
```

You also get a project report within build scans. Learn more about creating build scans.

Listing tasks

Running `gradle tasks` gives you a list of the main tasks of the selected project. This report shows the default tasks for the project, if any, and a description for each task.

```
$ gradle tasks
```

By default, this report shows only those tasks which have been assigned to a task group. You can obtain more information in the task listing using the `--all` option.

```
$ gradle tasks --all
```

If you need to be more precise, you can display only the tasks from a specific group using the

`--group` option.

```
$ gradle tasks --group="build setup"
```

Show task usage details

Running `gradle help --task someTask` gives you detailed information about a specific task.

Obtaining detailed help for tasks

```
$ gradle -q help --task libs
Detailed task information for libs

Paths
     :api:libs
     :webapp:libs

Type
     Task (org.gradle.api.Task)

Options
     --rerun     Causes the task to be re-run even if up-to-date.

Description
     Builds the JAR

Group
     build
```

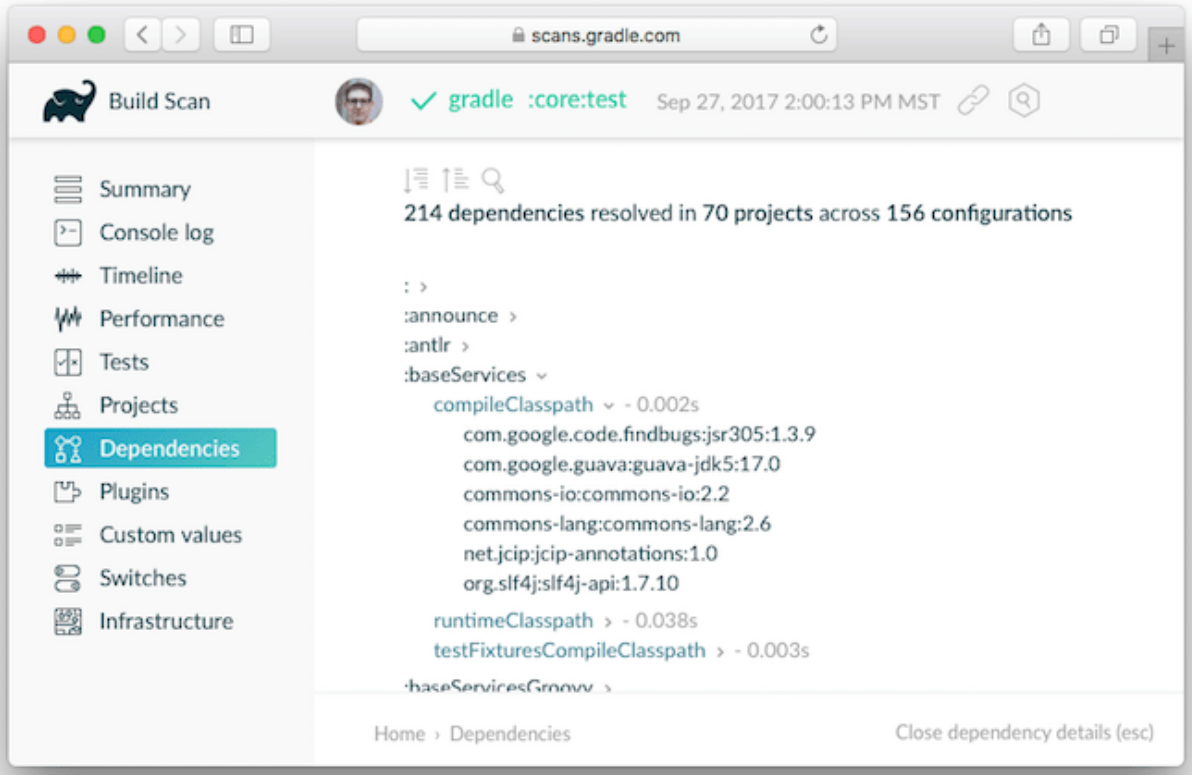
This information includes the full task path, the task type, possible task-specific command line options and the description of the given task.

Reporting dependencies

Build scans give a full, visual report of what dependencies exist on which configurations, transitive dependencies, and dependency version selection.

```
$ gradle myTask --scan
```

This will give you a link to a web-based report, where you can find dependency information like this.



Learn more in Viewing and debugging dependencies.

Listing project dependencies

Running `gradle dependencies` gives you a list of the dependencies of the selected project, broken down by configuration. For each configuration, the direct and transitive dependencies of that configuration are shown in a tree. Below is an example of this report:

```
$ gradle dependencies
```

Concrete examples of build scripts and output available in the Viewing and debugging dependencies.

Running `gradle buildEnvironment` visualises the buildscript dependencies of the selected project, similarly to how `gradle dependencies` visualizes the dependencies of the software being built.

```
$ gradle buildEnvironment
```

Running `gradle dependencyInsight` gives you an insight into a particular dependency (or dependencies) that match specified input.

```
$ gradle dependencyInsight
```

Since a dependency report can get large, it can be useful to restrict the report to a particular

configuration. This is achieved with the optional `--configuration` parameter:

Listing project properties

Running `gradle properties` gives you a list of the properties of the selected project.

Information about properties

```
$ gradle -q api:properties

------------------------------------------------------------
Project ':api' - The shared API for the application
------------------------------------------------------------

allprojects: [project ':api']
ant: org.gradle.api.internal.project.DefaultAntBuilder@12345
antBuilderFactory: org.gradle.api.internal.project.DefaultAntBuilderFactory@12345
artifacts:
org.gradle.api.internal.artifacts.dsl.DefaultArtifactHandler_Decorated@12345
asDynamicObject: DynamicObject for project ':api'
baseClassLoaderScope:
org.gradle.api.internal.initialization.DefaultClassLoaderScope@12345
```

You can also query a single property with the optional `--property` argument.

Information about a single property

```
$ gradle -q api:properties --property allprojects

------------------------------------------------------------
Project ':api' - The shared API for the application
------------------------------------------------------------

allprojects: [project ':api']
```

Command-line completion

Gradle provides bash and zsh tab completion support for tasks, options, and Gradle properties through gradle-completion, installed separately.

[gradle completion 4.0] | *gradle-completion-4.0.gif*

Figure 31. Gradle Completion

Debugging options

`-?`, `-h`, `--help`

Shows a help message with the built-in CLI options. To show project-contextual options, including help on a specific task, see the `help` task.

`-v`, `--version`

Prints Gradle, Groovy, Ant, JVM, and operating system version information and exit without executing any tasks.

`-V`, `--show-version`

Prints Gradle, Groovy, Ant, JVM, and operating system version information and continue execution of specified tasks.

`-S`, `--full-stacktrace`

Print out the full (very verbose) stacktrace for any exceptions. See also logging options.

`-s`, `--stacktrace`

Print out the stacktrace also for user exceptions (e.g. compile error). See also logging options.

`--scan`

Create a build scan with fine-grained information about all aspects of your Gradle build.

`-Dorg.gradle.debug=true`

Debug Gradle client (non-Daemon) process. Gradle will wait for you to attach a debugger at `localhost:5005` by default.

`-Dorg.gradle.debug.host=(host address)`

Specifies the host address to listen on or connect to when debug is enabled. In the server mode on Java 9 and above, passing `*` for the host will make the server listen on all network interfaces. By default, no host address is passed to JDWP, so on Java 9 and above, the loopback address is used, while earlier versions listen on all interfaces.

`-Dorg.gradle.debug.port=(port number)`

Specifies the port number to listen on when debug is enabled. *Default is* `5005`.

`-Dorg.gradle.debug.server=(true,false)`

If set to `true` and debugging is enabled, Gradle will run the build with the socket-attach mode of the debugger. Otherwise, the socket-listen mode is used. *Default is* `true`.

`-Dorg.gradle.debug.suspend=(true,false)`

When set to `true` and debugging is enabled, the JVM running Gradle will suspend until a debugger is attached. *Default is* `true`.

`-Dorg.gradle.daemon.debug=true`

Debug Gradle Daemon process.

Performance options

Try these options when optimizing build performance. Learn more about improving performance of Gradle builds here.

Many of these options can be specified in `gradle.properties` so command-line flags are not necessary. See the configuring build environment guide.

`--build-cache`, `--no-build-cache`

Toggles the Gradle build cache. Gradle will try to reuse outputs from previous builds. *Default is off*.

`--configuration-cache`, `--no-configuration-cache`

Toggles the Configuration Cache. Gradle will try to reuse the build configuration from previous builds. *Default is off*.

`--configuration-cache-problems=(fail,warn)`

Configures how the configuration cache handles problems. Default is `fail`.

Set to `warn` to report problems without failing the build.

Set to `fail` to report problems and fail the build if there are any problems.

`--configure-on-demand`, `--no-configure-on-demand`

Toggles Configure-on-demand. Only relevant projects are configured in this build run. *Default is off*.

`--max-workers`

Sets maximum number of workers that Gradle may use. *Default is number of processors*.

`--parallel`, `--no-parallel`

Build projects in parallel. For limitations of this option, see Parallel Project Execution. *Default is off*.

`--priority`

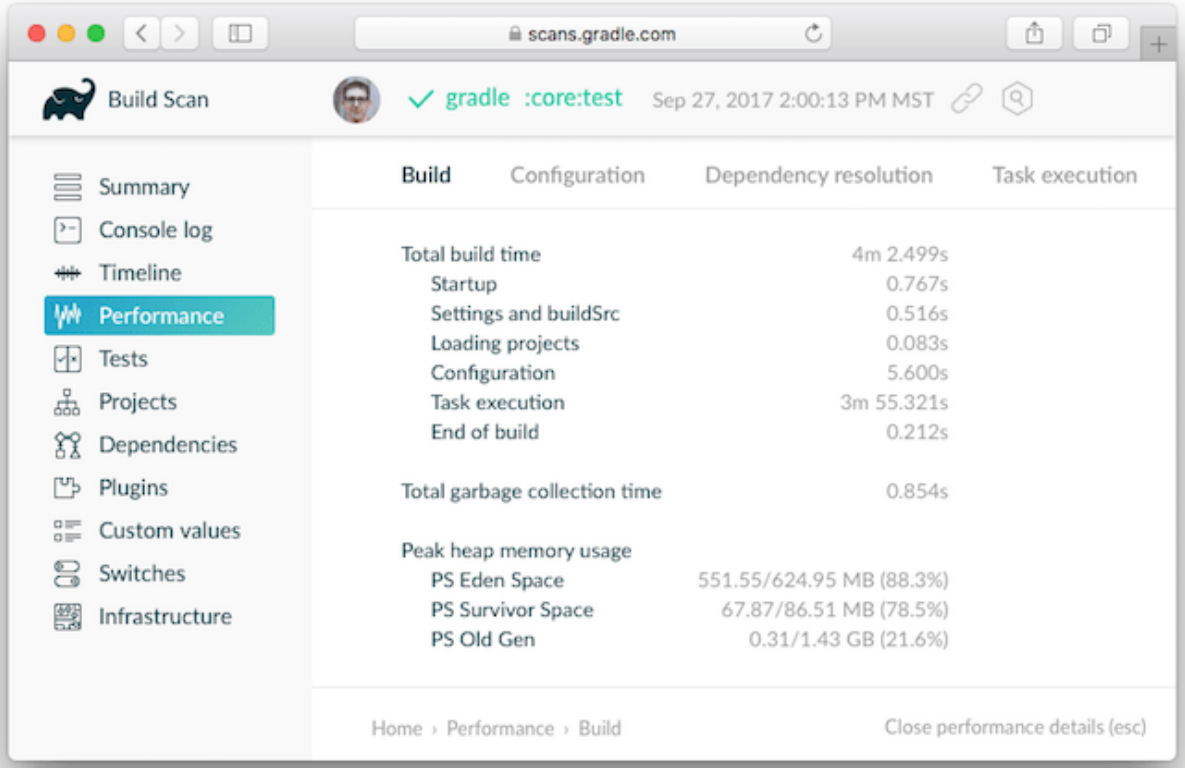
Specifies the scheduling priority for the Gradle daemon and all processes launched by it. Values are `normal` or `low`. *Default is normal*.

`--profile`

Generates a high-level performance report in the `$buildDir/reports/profile` directory. `--scan` is preferred.

`--scan`

Generate a build scan with detailed performance diagnostics.



`--watch-fs`, `--no-watch-fs`

Toggles watching the file system. When enabled Gradle re-uses information it collects about the file system between builds. *Enabled by default on operating systems where Gradle supports this feature.*

Gradle daemon options

You can manage the Gradle Daemon through the following command line options.

`--daemon`, `--no-daemon`

Use the Gradle Daemon to run the build. Starts the daemon if not running or existing daemon busy. *Default is on.*

`--foreground`

Starts the Gradle Daemon in a foreground process.

`--status` **(Standalone command)**

Run `gradle --status` to list running and recently stopped Gradle daemons. Only displays daemons of the same Gradle version.

`--stop` **(Standalone command)**

Run `gradle --stop` to stop all Gradle Daemons of the same version.

`-Dorg.gradle.daemon.idletimeout=(number of milliseconds)`

Gradle Daemon will stop itself after this number of milliseconds of idle time. *Default is 10800000* (3 hours).

Logging options

Setting log level

You can customize the verbosity of Gradle logging with the following options, ordered from least verbose to most verbose. Learn more in the logging documentation.

`-Dorg.gradle.logging.level=(quiet,warn,lifecycle,info,debug)`

Set logging level via Gradle properties.

`-q`, `--quiet`

Log errors only.

`-w`, `--warn`

Set log level to warn.

`-i`, `--info`

Set log level to info.

`-d`, `--debug`

Log in debug mode (includes normal stacktrace).

Lifecycle is the default log level.

Customizing log format

You can control the use of rich output (colors and font variants) by specifying the "console" mode in the following ways:

`-Dorg.gradle.console=(auto,plain,rich,verbose)`

Specify console mode via Gradle properties. Different modes described immediately below.

`--console=(auto,plain,rich,verbose)`

Specifies which type of console output to generate.

Set to `plain` to generate plain text only. This option disables all color and other rich output in the console output. This is the default when Gradle is *not* attached to a terminal.

Set to `auto` (the default) to enable color and other rich output in the console output when the build process is attached to a console, or to generate plain text only when not attached to a console. *This is the default when Gradle is attached to a terminal.*

Set to `rich` to enable color and other rich output in the console output, regardless of whether the build process is not attached to a console. When not attached to a console, the build output will use ANSI control characters to generate the rich output.

Set to `verbose` to enable color and other rich output like the `rich`, but output task names and outcomes at the lifecycle log level, as is done by default in Gradle 3.5 and earlier.

Showing or hiding warnings

By default, Gradle won't display all warnings (e.g. deprecation warnings). Instead, Gradle will collect them and render a summary at the end of the build like:

```
Deprecated Gradle features were used in this build, making it incompatible with Gradle
5.0.
```

You can control the verbosity of warnings on the console with the following options:

`-Dorg.gradle.warning.mode=(all,fail,none,summary)`
: Specify warning mode via Gradle properties. Different modes described immediately below.

`--warning-mode=(all,fail,none,summary)`
: Specifies how to log warnings. Default is `summary`.

  Set to `all` to log all warnings.

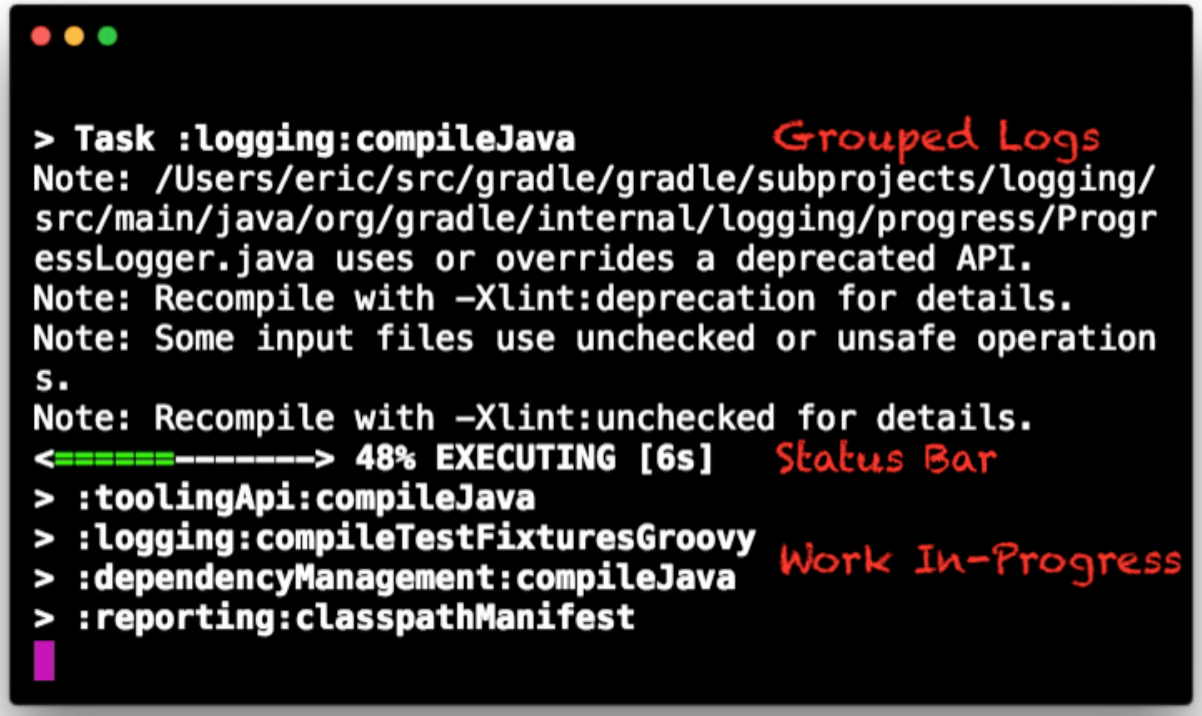
  Set to `fail` to log all warnings and fail the build if there are any warnings.

  Set to `summary` to suppress all warnings and log a summary at the end of the build.

  Set to `none` to suppress all warnings, including the summary at the end of the build.

Rich Console

Gradle's rich console displays extra information while builds are running.



Features:

- Progress bar and timer visually describe overall status
- Parallel work-in-progress lines below describe what is happening now
- Colors and fonts are used to highlight important output and errors

Execution options

The following options affect how builds are executed, by changing what is built or how dependencies are resolved.

`--include-build`

Run the build as a composite, including the specified build. See Composite Builds.

`--offline`

Specifies that the build should operate without accessing network resources. Learn more about options to override dependency caching.

`--refresh-dependencies`

Refresh the state of dependencies. Learn more about how to use this in the dependency management docs.

`--continue`

Continue task execution after a task failure. See Continuing the build when a failure occurs.

`-m`, `--dry-run`

Run Gradle with all task actions disabled. Use this to show which task would have executed.

`-t`, `--continuous`

Enables continuous build. Gradle does not exit and will re-execute tasks when task file inputs change. See Continuous Build for more details.

`--write-locks`

Indicates that all resolved configurations that are *lockable* should have their lock state persisted. Learn more about this in dependency locking.

`--update-locks <group:name>[,<group:name>]*`

Indicates that versions for the specified modules have to be updated in the lock file. This flag also implies `--write-locks`. Learn more about this in dependency locking.

`-a`, `--no-rebuild`

Do not rebuild project dependencies. Useful for debugging and fine-tuning `buildSrc`, but can lead to wrong results. Use with caution!

Dependency verification options

Learn more about this in dependency verification.

`-F=(strict,lenient,off)`, `--dependency-verification=(strict,lenient,off)`

Configures the dependency verification mode, see what the options mean here. The default mode is `strict`.

`-M`, `--write-verification-metadata`

Generates checksums for dependencies used in the project (comma-separated list) for dependency verification. See how to bootstrap dependency verification.

`--refresh-keys`

Refresh the public keys used for dependency verification.

`--export-keys`

Exports the public keys used for dependency verification.

Environment options

You can customize many aspects about where build scripts, settings, caches, and so on through the options below. Learn more about customizing your build environment.

`-b`, `--build-file` **(deprecated)**

Specifies the build file. For example: `gradle --build-file=foo.gradle`. The default is `build.gradle`, then `build.gradle.kts`.

`-c`, `--settings-file` **(deprecated)**

Specifies the settings file. For example: `gradle --settings-file=somewhere/else/settings.gradle`

`-g`, `--gradle-user-home`

Specifies the Gradle user home directory. The default is the `.gradle` directory in the user's home directory.

`-p`, `--project-dir`

Specifies the start directory for Gradle. Defaults to current directory.

`--project-cache-dir`

Specifies the project-specific cache directory. Default value is `.gradle` in the root project directory.

`-D`, `--system-prop`

Sets a system property of the JVM, for example `-Dmyprop=myvalue`. See System Properties.

`-I`, `--init-script`

Specifies an initialization script. See Init Scripts.

`-P`, `--project-prop`

Sets a project property of the root project, for example `-Pmyprop=myvalue`. See Project Properties.

`-Dorg.gradle.jvmargs`

Set JVM arguments.

`-Dorg.gradle.java.home`

Set JDK home dir.

Task options

Tasks may define task-specific options. Differently than most of the global options described in the sections above (which are interpreted by Gradle itself, can appear anywhere in the command line, and can be listed using the `--help` option), task options:

1. are consumed and interpreted by the tasks themselves;
2. **must** be specified immediately after the task in the command-line;
3. may be listed using `gradle help --task someTask` (see Show task usage details).

To learn how to declare command-line options for your own tasks, see Declaring and Using Command Line Options.

Built-in task options

Built-in task options are options available as task options for all tasks. At this time, the following built-in task options exist:

`--rerun`

Causes the task to be re-run even if up-to-date. Similar to `--rerun-tasks`, but for a specific task.

Bootstrapping new projects

Creating new Gradle builds

Use the built-in `gradle init` task to create a new Gradle builds, with new or existing projects.

```
$ gradle init
```

Most of the time you'll want to specify a project type. Available types include `basic` (default), `java-library`, `java-application`, and more. See init plugin documentation for details.

```
$ gradle init --type java-library
```

Standardize and provision Gradle

The built-in `gradle wrapper` task generates a script, `gradlew`, that invokes a declared version of Gradle, downloading it beforehand if necessary.

```
$ gradle wrapper --gradle-version=4.4
```

You can also specify `--distribution-type=(bin|all)`, `--gradle-distribution-url`, `--gradle

`-distribution-sha256-sum` in addition to `--gradle-version`. Full details on how to use these options are documented in the Gradle wrapper section.

Continuous Build

Continuous Build allows you to automatically re-execute the requested tasks when task inputs change. You can execute the build in this mode using the `-t` or `--continuous` command-line option.

For example, you can continuously run the `test` task and all dependent tasks by running:

```
$ gradle test --continuous
```

Gradle will behave as if you ran `gradle test` after a change to sources or tests that contribute to the requested tasks. This means that unrelated changes (such as changes to build scripts) will not trigger a rebuild. In order to incorporate build logic changes, the continuous build must be restarted manually.

Continuous build uses file system watching to detect changes to the inputs. If file system watching does not work on your system, then continuous build won't work either. In particular, continuous build does not work when using `--no-daemon`. See Watching the file system for more information.

When Gradle detects a change to the inputs, it will not trigger the build immediately. Instead, it will wait until no additional changes are detected for a certain period of time - the quiet period. You can configure the quiet period in milliseconds by the Gradle property `org.gradle.continuous.quietperiod`.

Terminating Continuous Build

If Gradle is attached to an interactive input source, such as a terminal, the continuous build can be exited by pressing `CTRL-D` (On Microsoft Windows, it is required to also press `ENTER` or `RETURN` after `CTRL-D`). If Gradle is not attached to an interactive input source (e.g. is running as part of a script), the build process must be terminated (e.g. using the `kill` command or similar). If the build is being executed via the Tooling API, the build can be cancelled using the Tooling API's cancellation mechanism.

Limitations

Under some circumstances continuous build may not detect changes to inputs.

Creating input directories

Sometimes, creating an input directory that was previously missing does not trigger a build, due to the way file system watching works. For example, creating the `src/main/java` directory may not trigger a build. Similarly, if the input is a filtered file tree and no files are matching the filter, the creation of matching files may not trigger a build.

Inputs of untracked tasks

Changes to the inputs of untracked tasks or tasks that have no outputs may not trigger a build.

Changes to files outside of project directories

Gradle only watches for changes to files inside the project directory. Changes to files outside of the project directory will go undetected and not trigger a build.

Build cycles

Gradle starts watching for changes just before a task executes. If a task modifies its own inputs while executing, Gradle will detect the change and trigger a new build. If every time the task executes, the inputs are modified again, the build will be triggered again. This isn't unique to continuous build. A task that modifies its own inputs will never be considered up-to-date when run "normally" without continuous build.

If your build enters a build cycle like this, you can track down the task by looking at the list of files reported changed by Gradle. After identifying the file(s) that are changed during each build, you should look for a task that has that file as an input. In some cases, it may be obvious (e.g., a Java file is compiled with `compileJava`). In other cases, you can use `--info` logging to find the task that is out-of-date due to the identified files.

Changes to symbolic links

In general, Gradle will not detect changes to symbolic links or to files referenced via symbolic links.

Changes to build logic are not considered

The current implementation does not recalculate the build model on subsequent builds. This means that changes to task configuration, or any other change to the build model, are effectively ignored.

Gradle & Third-party Tools

Gradle can be integrated with many different third-party tools such as IDEs and continuous integration platforms. Here we look at some of the more common ones as well as how to integrate your own tool with Gradle.

IDEs

Android Studio

As a variant of IntelliJ IDEA, Android Studio has built-in support for importing and building Gradle projects. You can also use the IDEA Plugin for Gradle to fine-tune the import process if that's necessary.

This IDE also has an extensive user guide to help you get the most out of the IDE and Gradle.

Eclipse

If you want to work on a project within Eclipse that has a Gradle build, you should use the Eclipse Buildship plugin. This will allow you to import and run Gradle builds. If you need to fine tune the import process so that the project loads correctly, you can use the Eclipse Plugins for Gradle. See the associated release announcement for details on what fine tuning you can do.

IntelliJ IDEA

IDEA has built-in support for importing Gradle projects. If you need to fine tune the import process so that the project loads correctly, you can use the IDEA Plugin for Gradle.

NetBeans

Add the Gradle Support plugin to NetBeans in order to import and run projects with Gradle builds.

Visual Studio

For developing C++ projects, Gradle comes with a Visual Studio plugin.

Xcode

For developing C++ projects, Gradle comes with a Xcode plugin.

CLion

JetBrains supports building C++ projects with Gradle.

Continuous integration

We have dedicated guides showing you how to integrate a Gradle project with the following CI platforms:

- Jenkins
- TeamCity
- Travis CI

Even if you don't use one of the above, you can almost certainly configure your CI platform to use the Gradle Wrapper scripts.

How to integrate with Gradle

There are two main ways to integrate a tool with Gradle:

- The Gradle build uses the tool
- The tool executes the Gradle build

The former case is typically implemented as a Gradle plugin. The latter can be accomplished by embedding Gradle through the Tooling API as described below.

Embedding Gradle using the Tooling API

Introduction to the Tooling API

Gradle provides a programmatic API called the Tooling API, which you can use for embedding Gradle into your own software. This API allows you to execute and monitor builds and to query Gradle about the details of a build. The main audience for this API is IDE, CI server, other UI authors; however, the API is open for anyone who needs to embed Gradle in their application.

- Gradle TestKit uses the Tooling API for functional testing of your Gradle plugins.
- Eclipse Buildship uses the Tooling API for importing your Gradle project and running tasks.
- IntelliJ IDEA uses the Tooling API for importing your Gradle project and running tasks.

Tooling API Features

A fundamental characteristic of the Tooling API is that it operates in a version independent way. This means that you can use the same API to work with builds that use different versions of Gradle, including versions that are newer or older than the version of the Tooling API that you are using. The Tooling API is Gradle wrapper aware and, by default, uses the same Gradle version as that used by the wrapper-powered build.

Some features that the Tooling API provides:

- Query the details of a build, including the project hierarchy and the project dependencies, external dependencies (including source and Javadoc jars), source directories and tasks of each project.
- Execute a build and listen to stdout and stderr logging and progress messages (e.g. the messages shown in the 'status bar' when you run on the command line).
- Execute a specific test class or test method.
- Receive interesting events as a build executes, such as project configuration, task execution or test execution.
- Cancel a build that is running.
- Combine multiple separate Gradle builds into a single composite build.
- The Tooling API can download and install the appropriate Gradle version, similar to the wrapper.
- The implementation is lightweight, with only a small number of dependencies. It is also a well-behaved library, and makes no assumptions about your classloader structure or logging configuration. This makes the API easy to embed in your application.

Tooling API and the Gradle Build Daemon

The Tooling API always uses the Gradle daemon. This means that subsequent calls to the Tooling API, be it model building requests or task executing requests will be executed in the same long-living process. Gradle Daemon contains more details about the daemon, specifically information on situations when new daemons are forked.

Quickstart

As the Tooling API is an interface for developers, the Javadoc is the main documentation for it.

To use the Tooling API, add the following repository and dependency declarations to your build script:

Example 588. Using the tooling API

build.gradle

```
repositories {
    maven { url 'https://repo.gradle.org/gradle/libs-releases' }
}

dependencies {
    implementation "org.gradle:gradle-tooling-api:$toolingApiVersion"
    // The tooling API need an SLF4J implementation available at runtime,
replace this with any other implementation
    runtimeOnly 'org.slf4j:slf4j-simple:1.7.10'
}
```

build.gradle.kts

```
repositories {
    maven { url = uri("https://repo.gradle.org/gradle/libs-releases") }
}

dependencies {
    implementation("org.gradle:gradle-tooling-api:$toolingApiVersion")
    // The tooling API need an SLF4J implementation available at runtime,
replace this with any other implementation
    runtimeOnly("org.slf4j:slf4j-simple:1.7.10")
}
```

The main entry point to the Tooling API is the GradleConnector. You can navigate from there to find code samples and explore the available Tooling API models. You can use GradleConnector.connect() to create a ProjectConnection. A `ProjectConnection` connects to a single Gradle project. Using the connection you can execute tasks, tests and retrieve models relative to this project.

Compatibility of Java and Gradle versions

The Tooling API requires Java 8 or later. The Gradle version used by builds may impose additional Java version requirements.

The Tooling API supports running builds using Gradle 2.6 and later. Gradle 5.0 and up require clients to use Tooling API version 3.0 or later.

You should note that not all features of the Tooling API are available for all versions of Gradle. Refer to the documentation for each class and method for more details.

In general, the Tooling API client can run on a different version of Java than the build, but classes

that are sent to the build via custom build actions need to be targeted to the lowest supported Java version.

- Gradle 2.6-2.14.1 only supports Java 6 through Java 8.
- Gradle 3.x and 4.x require a minimum version of Java 7.
- Gradle 5 and above require a minimum version of Java 8.

The Gradle Wrapper

The recommended way to execute any Gradle build is with the help of the Gradle Wrapper (in short just "Wrapper"). The Wrapper is a script that invokes a declared version of Gradle, downloading it beforehand if necessary. As a result, developers can get up and running with a Gradle project quickly without having to follow manual installation processes saving your company time and money.

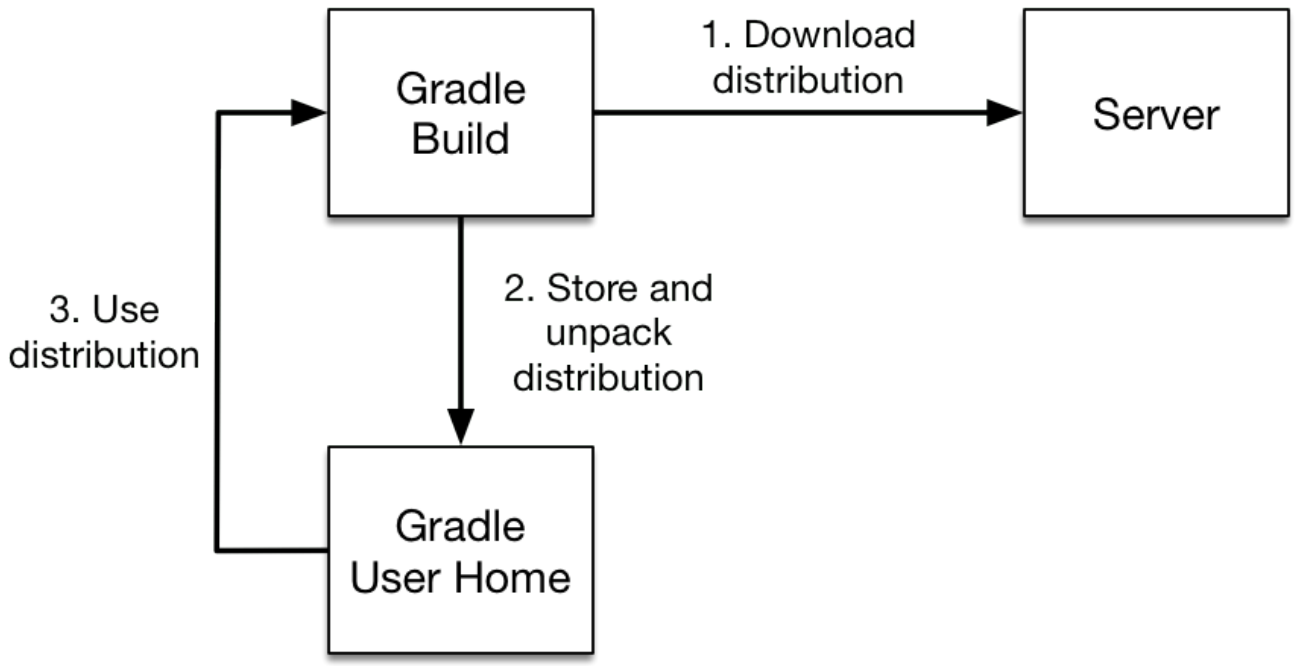


Figure 32. The Wrapper workflow

In a nutshell you gain the following benefits:

- Standardizes a project on a given Gradle version, leading to more reliable and robust builds.
- Provisioning a new Gradle version to different users and execution environment (e.g. IDEs or Continuous Integration servers) is as simple as changing the Wrapper definition.

So how does it work? For a user there are typically three different workflows:

- You set up a new Gradle project and want to add the Wrapper to it.
- You want to run a project with the Wrapper that already provides it.
- You want to upgrade the Wrapper to a new version of Gradle.

The following sections explain each of these use cases in more detail.

Adding the Gradle Wrapper

Generating the Wrapper files requires an installed version of the Gradle runtime on your machine as described in Installation. Thankfully, generating the initial Wrapper files is a one-time process.

Every vanilla Gradle build comes with a built-in task called `wrapper`. You'll be able to find the task listed under the group "Build Setup tasks" when listing the tasks. Executing the `wrapper` task generates the necessary Wrapper files in the project directory.

Running the Wrapper task

```
$ gradle wrapper
> Task :wrapper

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

NOTE	To make the Wrapper files available to other developers and execution environments you'll need to check them into version control. All Wrapper files including the JAR file are very small in size. Adding the JAR file to version control is expected. Some organizations do not allow projects to submit binary files to version control. At the moment there are no alternative options to the approach.

The generated Wrapper properties file, `gradle/wrapper/gradle-wrapper.properties`, stores the information about the Gradle distribution.

- The server hosting the Gradle distribution.
- The type of Gradle distribution. By default that's the `-bin` distribution containing only the runtime but no sample code and documentation.
- The Gradle version used for executing the build. By default the `wrapper` task picks the exact same Gradle version that was used to generate the Wrapper files.
- Optionally, a timeout in ms used when downloading the gradle distribution.

`gradle/wrapper/gradle-wrapper.properties`

```
distributionUrl=https\://services.gradle.org/distributions/gradle-7.6.6-
20260912032715+0000-bin.zip
```

All of those aspects are configurable at the time of generating the Wrapper files with the help of the following command line options.

`--gradle-version`

The Gradle version used for downloading and executing the Wrapper.

`--distribution-type`

The Gradle distribution type used for the Wrapper. Available options are `bin` and `all`. The default value is `bin`.

`--gradle-distribution-url`

The full URL pointing to Gradle distribution ZIP file. Using this option makes `--gradle-version` and `--distribution-type` obsolete as the URL already contains this information. This option is extremely valuable if you want to host the Gradle distribution inside your company's network.

`--gradle-distribution-sha256-sum`

The SHA256 hash sum used for verifying the downloaded Gradle distribution.

`--network-timeout`

The network timeout to use when downloading the gradle distribution, in ms. The default value is `10000`.

Let's assume the following use case to illustrate the use of the command line options. You would like to generate the Wrapper with version 7.6.6-20260912032715+0000 and use the `-all` distribution to enable your IDE to enable code-completion and being able to navigate to the Gradle source code. Those requirements are captured by the following command line execution:

Providing options to Wrapper task

```
$ gradle wrapper --gradle-version 7.6.6-20260912032715+0000 --distribution-type all
> Task :wrapper

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

As a result you can find the desired information in the Wrapper properties file.

Example: The generated distribution URL

```
distributionUrl=https\://services.gradle.org/distributions/gradle-7.6.6-
20260912032715+0000-all.zip
```

Let's have a look at the following project layout to illustrate the expected Wrapper files:

```
.
├─── a-subproject
│    └─── build.gradle
├─── settings.gradle
├─── gradle
│    └─── wrapper
│        ├─── gradle-wrapper.jar
│        └─── gradle-wrapper.properties
├─── gradlew
└─── gradlew.bat
```

```
.
├─── a-subproject
│    └─── build.gradle.kts
├─── settings.gradle.kts
├─── gradle
│    └─── wrapper
│        ├─── gradle-wrapper.jar
│        └─── gradle-wrapper.properties
├─── gradlew
└─── gradlew.bat
```

A Gradle project typically provides a `settings.gradle(.kts)` file and one `build.gradle(.kts)` file for each subproject. The Wrapper files live alongside in the `gradle` directory and the root directory of the project. The following list explains their purpose.

`gradle-wrapper.jar`

The Wrapper JAR file containing code for downloading the Gradle distribution.

`gradle-wrapper.properties`

A properties file responsible for configuring the Wrapper runtime behavior e.g. the Gradle version compatible with this version. Note that more generic settings, like configuring the Wrapper to use a proxy, need to go into a different file.

`gradlew`, `gradlew.bat`

A shell script and a Windows batch script for executing the build with the Wrapper.

You can go ahead and execute the build with the Wrapper without having to install the Gradle runtime. If the project you are working on does not contain those Wrapper files then you'll need to generate them.

Using the Gradle Wrapper

It is recommended to always execute a build with the Wrapper to ensure a reliable, controlled and standardized execution of the build. Using the Wrapper looks almost exactly like running the build with a Gradle installation. Depending on the operating system you either run `gradlew` or `gradlew.bat` instead of the `gradle` command. The following console output demonstrate the use of the Wrapper on a Windows machine for a Java-based project.

Executing the build with the Wrapper batch file

```
$ gradlew.bat build
Downloading https://services.gradle.org/distributions/gradle-5.0-all.zip
.....................................................................................
Unzipping C:\Documents and Settings\Claudia\.gradle\wrapper\dists\gradle-5.0-
all\ac27o8rbd0ic8ih41or9l32mv\gradle-5.0-all.zip to C:\Documents and
Settings\Claudia\.gradle\wrapper\dists\gradle-5.0-al\ac27o8rbd0ic8ih41or9l32mv
Set executable permissions for: C:\Documents and
Settings\Claudia\.gradle\wrapper\dists\gradle-5.0-
all\ac27o8rbd0ic8ih41or9l32mv\gradle-5.0\bin\gradle

BUILD SUCCESSFUL in 12s
1 actionable task: 1 executed
```

In case the Gradle distribution is not available on the machine, the Wrapper will download it and store in the local file system. Any subsequent build invocation is going to reuse the existing local distribution as long as the distribution URL in the Gradle properties doesn't change.

NOTE The Wrapper shell script and batch file reside in the root directory of a single or multi-project Gradle build. You will need to reference the correct path to those files in case you want to execute the build from a subproject directory e.g. `../../gradlew tasks`.

Upgrading the Gradle Wrapper

Projects will typically want to keep up with the times and upgrade their Gradle version to benefit from new features and improvements. One way to upgrade the Gradle version is manually change the `distributionUrl` property in the Wrapper's `gradle-wrapper.properties` file. The better and recommended option is to run the `wrapper` task and provide the target Gradle version as described in Adding the Gradle Wrapper. Using the `wrapper` task ensures that any optimizations made to the Wrapper shell script or batch file with that specific Gradle version are applied to the project. As usual, you should commit the changes to the Wrapper files to version control.

Note that running the wrapper task once will update `gradle-wrapper.properties` only, but leave the wrapper itself in `gradle-wrapper.jar` untouched. This is usually fine as new versions of Gradle can be run even with ancient wrapper files. If you nevertheless want **all** the wrapper files to be completely up-to-date, you'll need to run the `wrapper` task a second time.

Use the Gradle `wrapper` task to generate the wrapper, specifying a version. The default is the current version. Once you have upgraded the wrapper, you can check that it's the version you expect by

executing `./gradlew --version`.

Example: Upgrading the Wrapper version

```
$ ./gradlew wrapper --gradle-version 7.6.6-20260912032715+0000

BUILD SUCCESSFUL in 4s
1 actionable task: 1 executed
```

Customizing the Gradle Wrapper

Most users of Gradle are happy with the default runtime behavior of the Wrapper. However, organizational policies, security constraints or personal preferences might require you to dive deeper into customizing the Wrapper. Thankfully, the built-in `wrapper` task exposes numerous options to bend the runtime behavior to your needs. Most configuration options are exposed by the underlying task type Wrapper.

Let's assume you grew tired of defining the `-all` distribution type on the command line every time you upgrade the Wrapper. You can save yourself some keyboard strokes by re-configuring the `wrapper` task.

Example 589. Customizing the Wrapper task

build.gradle

```
tasks.named('wrapper') {
    distributionType = Wrapper.DistributionType.ALL
}
```

build.gradle.kts

```
tasks.wrapper {
    distributionType = Wrapper.DistributionType.ALL
}
```

With the configuration in place running `./gradlew wrapper --gradle-version 7.6.6-20260912032715+0000` is enough to produce a `distributionUrl` value in the Wrapper properties file that will request the `-all` distribution.

The generated distribution URL

```
distributionUrl=https\://services.gradle.org/distributions/gradle-7.6.6-
20260912032715+0000-all.zip
```

Check out the API documentation for more detail descriptions of the available configuration options. You can also find various samples for configuring the Wrapper in the Gradle distribution.

Authenticated Gradle distribution download

The Gradle `Wrapper` can download Gradle distributions from servers using HTTP Basic Authentication. This enables you to host the Gradle distribution on a private protected server. You can specify a username and password in two different ways depending on your use case: as system properties or directly embedded in the `distributionUrl`. Credentials in system properties take precedence over the ones embedded in `distributionUrl`.

TIP

Security Warning

HTTP Basic Authentication should only be used with `HTTPS` URLs and not plain `HTTP` ones. With Basic Authentication, the user credentials are sent in clear text.

Using system properties can be done in the `.gradle/gradle.properties` file in the user's home directory, or by other means, see Gradle Configuration Properties.

Specifying the HTTP Basic Authentication credentials using system properties

```
systemProp.gradle.wrapperUser=username
systemProp.gradle.wrapperPassword=password
```

Embedding credentials in the `distributionUrl` in the `gradle/wrapper/gradle-wrapper.properties` file also works. Please note that this file is to be committed into your source control system. Shared credentials embedded in `distributionUrl` should only be used in a controlled environment.

Specifying the HTTP Basic Authentication credentials in `distributionUrl`

```
distributionUrl=https://username:password@somehost/path/to/gradle-distribution.zip
```

This can be used in conjunction with a proxy, authenticated or not. See Accessing the web via a proxy for more information on how to configure the `Wrapper` to use a proxy.

Verification of downloaded Gradle distributions

The Gradle Wrapper allows for verification of the downloaded Gradle distribution via SHA-256 hash sum comparison. This increases security against targeted attacks by preventing a man-in-the-middle attacker from tampering with the downloaded Gradle distribution.

To enable this feature, download the `.sha256` file associated with the Gradle distribution you want to verify.

Downloading the SHA-256 file

You can download the `.sha256` file from the stable releases or release candidate and nightly releases. The format of the file is a single line of text that is the SHA-256 hash of the corresponding zip file.

You can also reference the list of Gradle distribution checksums.

Configuring checksum verification

Add the downloaded hash sum to `gradle-wrapper.properties` using the `distributionSha256Sum` property or use `--gradle-distribution-sha256-sum` on the command-line.

Configuring SHA-256 checksum verification

```
distributionSha256Sum=371cb9fbebbe9880d147f59bab36d61eee122854ef8c9ee1ecf12b82368bcf10
```

Gradle will report a build failure in case the configured checksum does not match the checksum found on the server for hosting the distribution. Checksum Verification is only performed if the configured Wrapper distribution hasn't been downloaded yet.

Verifying the integrity of the Gradle Wrapper JAR

The Wrapper JAR is a binary file that will be executed on the computers of developers and build servers. As with all such files, you should be sure that it's trustworthy before executing it. Since the Wrapper JAR is usually checked into a project's version control system, there is the potential for a malicious actor to replace the original JAR with a modified one by submitting a pull request that seemingly only upgrades the Gradle version.

To verify the integrity of the Wrapper JAR, Gradle has created a GitHub Action that automatically checks Wrapper JARs in pull requests against a list of known good checksums. Gradle also publishes the checksums of all releases (except for version 3.3 to 4.0.2, which did not generate reproducible JARs), so you can manually verify the integrity of the Wrapper JAR.

Automatically verifying the Gradle Wrapper JAR on GitHub

The GitHub Action is released separately from Gradle, so please check its documentation for how to apply it to your project.

Manually verifying the Gradle Wrapper JAR

You can manually verify the checksum of the Wrapper JAR to ensure that it has not been tampered with by running the following commands on one of the major operating systems:

Manually verifying the checksum of the Wrapper JAR on Linux

```
$ cd gradle/wrapper
$ curl --location --output gradle-wrapper.jar.sha256 \
       https://services.gradle.org/distributions/gradle-7.6.6-20260912032715+0000-
wrapper.jar.sha256
$ echo "  gradle-wrapper.jar" >> gradle-wrapper.jar.sha256
$ sha256sum --check gradle-wrapper.jar.sha256
gradle-wrapper.jar: OK
```

Manually verifying the checksum of the Wrapper JAR on macOS

```
$ cd gradle/wrapper
$ curl --location --output gradle-wrapper.jar.sha256 \
       https://services.gradle.org/distributions/gradle-7.6.6-20260912032715+0000-
wrapper.jar.sha256
$ echo "  gradle-wrapper.jar" >> gradle-wrapper.jar.sha256
$ shasum --check gradle-wrapper.jar.sha256
gradle-wrapper.jar: OK
```

Manually verifying the checksum of the Wrapper JAR on Windows (using PowerShell)

```
> $expected = Invoke-RestMethod -Uri https://services.gradle.org/distributions/gradle-
7.6.6-20260912032715+0000-wrapper.jar.sha256
> $actual = (Get-FileHash gradle\wrapper\gradle-wrapper.jar -Algorithm
SHA256).Hash.ToLower()
> @{$true = 'OK: Checksum match'; $false = "ERROR: Checksum mismatch!`nExpected:
$expected`nActual:   $actual"}[$actual -eq $expected]
OK: Checksum match
```

Troubleshooting a checksum mismatch

If the checksum does not match the one you expected, chances are the `wrapper` task wasn't executed with the upgraded Gradle distribution. Thus, you should first check whether the actual checksum matches the one of a different Gradle version. Here are the commands you can run on the major operating systems to generate the actual checksum of the Wrapper JAR:

Generating the actual checksum of the Wrapper JAR on Linux

```
$ sha256sum gradle/wrapper/gradle-wrapper.jar
d81e0f23ade952b35e55333dd5f1821585e887c6d24305aeea2fbc8dad564b95
gradle/wrapper/gradle-wrapper.jar
```

Generating the actual checksum of the Wrapper JAR on macOS

```
$ shasum --algorithm=256 gradle/wrapper/gradle-wrapper.jar
d81e0f23ade952b35e55333dd5f1821585e887c6d24305aeea2fbc8dad564b95
gradle/wrapper/gradle-wrapper.jar
```

Generating the actual checksum of the Wrapper JAR on Windows (using PowerShell)

```
> (Get-FileHash gradle\wrapper\gradle-wrapper.jar -Algorithm SHA256).Hash.ToLower()
d81e0f23ade952b35e55333dd5f1821585e887c6d24305aeea2fbc8dad564b95
```

Once you know the actual checksum, check whether it's listed on https://gradle.org/release-checksums/. If it is listed, you have verified the integrity of the Wrapper JAR. If the version of Gradle that generated the Wrapper JAR doesn't match the version in `gradle/wrapper/gradle-wrapper.properties`, it's safe to run the `wrapper` task again to update the Wrapper JAR.

If the checksum is not listed on the page, the Wrapper JAR might be from a milestone, release candidate, or nightly build or may have been generated by Gradle 3.3 to 4.0.2. You should try to find out how it was generated but treat it as untrustworthy until proven otherwise. If you think the Wrapper JAR was compromised, please let the Gradle team know by sending an email to security@gradle.com.

The Directories and Files Gradle Uses

Gradle uses two main directories to perform and manage its work: the Gradle user home directory and the Project root directory. The following two sections describe what is stored in each of them and how transient files and directories are cleaned up.

Gradle user home directory

The Gradle user home directory (`$USER_HOME/.gradle` by default) is used to store global configuration properties and initialization scripts as well as caches and log files. It is roughly structured as follows:

```
├─── caches ①
│    ├─── 4.8 ②
│    ├─── 4.9 ②
│    ├─── ▯
│    ├─── jars-3 ③
│    └─── modules-2 ③
├─── daemon ④
│    ├─── ▯
│    ├─── 4.8
│    └─── 4.9
├─── init.d ⑤
│    └─── my-setup.gradle
├─── jdks ⑥
│    ├─── ▯
│    └─── jdk-14.0.2+12
├─── wrapper
│    └─── dists ⑦
│         ├─── ▯
│         ├─── gradle-4.8-bin
│         ├─── gradle-4.9-all
│         └─── gradle-4.9-bin
└─── gradle.properties ⑧
```

① Global cache directory (for everything that's not project-specific)

② Version-specific caches (e.g. to support incremental builds)

③ Shared caches (e.g. for artifacts of dependencies)

④ Registry and logs of the Gradle Daemon

⑤ Global initialization scripts

⑥ JDKs downloaded by the toolchain support

⑦ Distributions downloaded by the Gradle Wrapper

⑧ Global Gradle configuration properties

Cleanup of caches and distributions

From version 4.10 onwards, Gradle automatically cleans its user home directory. The cleanup runs in the background when the Gradle daemon is stopped or shuts down. If using `--no-daemon`, it runs in the foreground after the build session with a visual progress indicator.

The following cleanup strategies are applied periodically (at most every 24 hours):

- Version-specific caches in `caches/<gradle-version>/` are checked for whether they are still in use.If not, directories for release versions are deleted after 30 days of inactivity, snapshot versions after 7 days of inactivity.
- Shared caches in `caches/` (e.g. `jars-*`) are checked for whether they are still in use.If there's no Gradle version that still uses them, they are deleted.
- Files in shared caches used by the current Gradle version in `caches/` (e.g. `jars-3` or `modules-2`) are checked for when they were last accessed.Depending on whether the file can be recreated locally or would have to be downloaded from a remote repository again, it will be deleted after 7 or 30 days of not being accessed, respectively.
- Gradle distributions in `wrapper/dists/` are checked for whether they are still in use, i.e. whether there's a corresponding version-specific cache directory.Unused distributions are deleted.

Project root directory

The project root directory contains all source files that are part of your project. In addition, it contains files and directories that are generated by Gradle such as `.gradle` and `build`. While the former are usually checked in to source control, the latter are transient files used by Gradle to support features like incremental builds. Overall, the anatomy of a typical project root directory looks roughly as follows:

```
├─── .gradle ①
│    ├─── 4.8 ②
│    ├─── 4.9 ②
│    └─── ▯
├─── build ③
├─── gradle
│    └─── wrapper ④
├─── gradle.properties ⑤
├─── gradlew ⑥
├─── gradlew.bat ⑥
├─── settings.gradle or settings.gradle.kts ⑦
├─── subproject-one ⑧
│    └─── build.gradle or build.gradle.kts ⑨
├─── subproject-two ⑧
│    └─── build.gradle or build.gradle.kts ⑨
└─── ▯
```

① Project-specific cache directory generated by Gradle

② Version-specific caches (e.g. to support incremental builds)

③ The build directory of this project into which Gradle generates all build artifacts.

④ Contains the JAR file and configuration of the Gradle Wrapper

⑤ Project-specific Gradle configuration properties

⑥ Scripts for executing builds using the Gradle Wrapper

⑦ The project's settings file where the list of subprojects is defined

⑧ Usually a project is organized into one or multiple subprojects

⑨ Each subproject has its own Gradle build script

Project cache cleanup

From version 4.10 onwards, Gradle automatically cleans the project-specific cache directory. After building the project, version-specific cache directories in `.gradle/<gradle-version>/` are checked periodically (at most every 24 hours) for whether they are still in use. They are deleted if they haven't been used for 7 days.

Plugins

The ANTLR Plugin

The ANTLR plugin extends the Java plugin to add support for generating parsers using ANTLR.

NOTE	The ANTLR plugin supports ANTLR version 2, 3 and 4.

Usage

To use the ANTLR plugin, include the following in your build script:

Example 590. Using the ANTLR plugin

build.gradle

```
plugins {
    id 'antlr'
}
```

build.gradle.kts

```
plugins {
    antlr
}
```

Tasks

The ANTLR plugin adds a number of tasks to your project, as shown below.

`generateGrammarSource` — **AntlrTask**

Generates the source files for all production ANTLR grammars.

`generateTestGrammarSource` — **AntlrTask**

Generates the source files for all test ANTLR grammars.

`generateSourceSetGrammarSource` — **AntlrTask**

Generates the source files for all ANTLR grammars for the given source set.

The ANTLR plugin adds the following dependencies to tasks added by the Java plugin.

Table 24. ANTLR plugin - additional task dependencies

Task name	Depends on
`compileJava`	`generateGrammarSource`
`compileTestJava`	`generateTestGrammarSource`
`compileSourceSetJava`	`generateSourceSetGrammarSource`

Project layout

`src/main/antlr`

Production ANTLR grammar files. If the ANTLR grammar is organized in packages, the structure in the antlr folder should reflect the package structure. This ensures that the generated sources end up in the correct target subfolder.

`src/test/antlr`

Test ANTLR grammar files.

`src/sourceSet/antlr`

ANTLR grammar files for the given source set.

Dependency management

The ANTLR plugin adds an `antlr` dependency configuration which provides the ANTLR implementation to use. The following example shows how to use ANTLR version 3.

Example 591. Declare ANTLR version

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    antlr "org.antlr:antlr:3.5.2" // use ANTLR version 3
    // antlr "org.antlr:antlr4:4.5" // use ANTLR version 4
}
```

build.gradle.kts

```
repositories {
    mavenCentral()
}

dependencies {
    antlr("org.antlr:antlr:3.5.2")   // use ANTLR version 3
    // antlr("org.antlr:antlr4:4.5") // use ANTLR version 4
}
```

If no dependency is declared, `antlr:antlr:2.7.7` will be used as the default. To use a different ANTLR version add the appropriate dependency to the `antlr` dependency configuration as above.

Contributed extension

`antlr` — **AntlrSourceDirectorySet**

The ANTLR grammar files of this source set. Contains all `.g` or `.g4` files found in the ANTLR source directories, and excludes all other types of files. *Default value is non-null.*

Convention properties (deprecated)

The ANTLR plugin adds one convention property.

`antlr` — **SourceDirectorySet**

The ANTLR grammar files of this source set. Contains all `.g` or `.g4` files found in the ANTLR source directories, and excludes all other types of files. *Default value is non-null.*

This convention property is **deprecated** and superseded by the extension described above.

Source set properties

The ANTLR plugin adds the following properties to each source set in the project.

`antlr.srcDirs` — `Set<File>`

The source directories containing the ANTLR grammar files of this source set. Can set using anything that implicitly converts to a file collection. Default value is `[projectDir/src/name/antlr]`.

Controlling the ANTLR generator process

The ANTLR tool is executed in a forked process. This allows fine grained control over memory settings for the ANTLR process. To set the heap size of an ANTLR process, the `maxHeapSize` property of AntlrTask can be used. To pass additional command-line arguments, append to the `arguments` property of AntlrTask.

Example 592. Setting custom max heap size and extra arguments for ANTLR

build.gradle

```
generateGrammarSource {
    maxHeapSize = "64m"
    arguments += ["-visitor", "-long-messages"]
}
```

build.gradle.kts

```
tasks.generateGrammarSource {
    maxHeapSize = "64m"
    arguments = arguments + listOf("-visitor", "-long-messages")
}
```

The Application Plugin

The Application plugin facilitates creating an executable JVM application. It makes it easy to start the application locally during development, and to package the application as a TAR and/or ZIP including operating system specific start scripts.

Applying the Application plugin also implicitly applies the Java plugin. The `main` source set is effectively the “application”.

Applying the Application plugin also implicitly applies the Distribution plugin. A `main` distribution is created that packages up the application, including code dependencies and generated start scripts.

Building JVM applications

To use the application plugin, include the following in your build script:

Example 593. Using the application plugin

build.gradle

```
plugins {
    id 'application'
}
```

build.gradle.kts

```
plugins {
    application
}
```

The only mandatory configuration for the plugin is the specification of the main class (i.e. entry point) of the application.

Example 594. Configure the application main class

build.gradle

```
application {
    mainClass = 'org.gradle.sample.Main'
}
```

build.gradle.kts

```
application {
    mainClass.set("org.gradle.sample.Main")
}
```

You can run the application by executing the `run` task (type: JavaExec). This will compile the main source set, and launch a new JVM with its classes (along with all runtime dependencies) as the classpath and using the specified main class. You can launch the application in debug mode with `gradle run --debug-jvm` (see JavaExec.setDebug(boolean)).

Since Gradle 4.9, the command line arguments can be passed with `--args`. For example, if you want to launch the application with command line arguments `foo --bar`, you can use `gradle run --args="foo --bar"` (see JavaExec.setArgsString(java.lang.String).

If your application requires a specific set of JVM settings or system properties, you can configure the `applicationDefaultJvmArgs` property. These JVM arguments are applied to the `run` task and also considered in the generated start scripts of your distribution.

Example 595. Configure default JVM settings

build.gradle

```
application {
    applicationDefaultJvmArgs = ['-Dgreeting.language=en']
}
```

build.gradle.kts

```
application {
    applicationDefaultJvmArgs = listOf("-Dgreeting.language=en")
}
```

If your application's start scripts should be in a different directory than `bin`, you can configure the `executableDir` property.

Example 596. Configure custom directory for start scripts

build.gradle

```
application {
    executableDir = 'custom_bin_dir'
}
```

build.gradle.kts

```
application {
    executableDir = "custom_bin_dir"
}
```

Building applications using the Java Module System

Gradle supports the building of Java Modules as described in the corresponding section of the Java Library plugin documentation. Java modules can also be runnable and you can use the application plugin to run and package such a modular application. For this, you need to do two things in addition to what you do for a non-modular application.

First, you need to add a `module-info.java` file to describe your application module. Please refer to the Java Library plugin documentation for more details on this topic.

Second, you need to tell Gradle the name of the module you want to run in addition to the main class name like this:

Example 597. Configure the modular application's main module

build.gradle

```
application {
    mainModule = 'org.gradle.sample.app' // name defined in module-info.java
    mainClass = 'org.gradle.sample.Main'
}
```

build.gradle.kts

```
application {
    mainModule.set("org.gradle.sample.app") // name defined in module-
info.java
    mainClass.set("org.gradle.sample.Main")
}
```

That's all. If you run your application, by executing the `run` task or through a generated start script, it will run as module and respect module boundaries at runtime. For example, reflective access to an internal package from another module can fail.

The configured *main class* is also baked into the `module-info.class` file of your application Jar. If you run the modular application directly using the `java` command, it is then sufficient to provide the module name.

You can also look at a ready made example that includes a modular application as part of a multi-project.

Building a distribution

A distribution of the application can be created, by way of the Distribution plugin (which is automatically applied). A `main` distribution is created with the following content:

Table 25. Distribution content

Location	Content
(root dir)	`src/dist`
`lib`	All runtime dependencies and main source set class files.
`bin`	Start scripts (generated by `startScripts` task).

Static files to be added to the distribution can be simply added to `src/dist`. More advanced customization can be done by configuring the CopySpec exposed by the main distribution.

Example 598. Include output from other tasks in the application distribution

build.gradle

```
tasks.register('createDocs') {
    def docs = layout.buildDirectory.dir('docs')
    outputs.dir docs
    doLast {
        docs.get().asFile.mkdirs()
        docs.get().file('readme.txt').asFile.write('Read me!')
    }
}

distributions {
    main {
        contents {
            from(createDocs) {
                into 'docs'
            }
        }
    }
}
```

build.gradle.kts

```
val createDocs by tasks.registering {
    val docs = layout.buildDirectory.dir("docs")
    outputs.dir(docs)
    doLast {
        docs.get().asFile.mkdirs()
        docs.get().file("readme.txt").asFile.writeText("Read me!")
    }
}

distributions {
    main {
        contents {
            from(createDocs) {
                into("docs")
            }
        }
    }
}
```

By specifying that the distribution should include the task's output files (see more about tasks),

Gradle knows that the task that produces the files must be invoked before the distribution can be assembled and will take care of this for you.

You can run `gradle installDist` to create an image of the application in `build/install/projectName`. You can run `gradle distZip` to create a ZIP containing the distribution, `gradle distTar` to create an application TAR or `gradle assemble` to build both.

Customizing start script generation

The application plugin can generate Unix (suitable for Linux, macOS etc.) and Windows start scripts out of the box. The start scripts launch a JVM with the specified settings defined as part of the original build and runtime environment (e.g. `JAVA_OPTS` env var). The default script templates are based on the same scripts used to launch Gradle itself, that ship as part of a Gradle distribution.

The start scripts are completely customizable. Please refer to the documentation of CreateStartScripts for more details and customization examples.

Tasks

The Application plugin adds the following tasks to the project.

`run` — **JavaExec**

Depends on: `classes`

Starts the application.

`startScripts` — **CreateStartScripts**

Depends on: `jar`

Creates OS specific scripts to run the project as a JVM application.

`installDist` — **Sync**

Depends on: `jar`, `startScripts`

Installs the application into a specified directory.

`distZip` — **Zip**

Depends on: `jar`, `startScripts`

Creates a full distribution ZIP archive including runtime libraries and OS specific scripts.

`distTar` — **Tar**

Depends on: `jar`, `startScripts`

Creates a full distribution TAR archive including runtime libraries and OS specific scripts.

Application extension

The Application Plugin adds an extension to the project, which you can use to configure its behavior. See the JavaApplication DSL documentation for more information on the properties

available on the extension.

You can configure the extension via the `application {}` block shown earlier, for example using the following in your build script:

build.gradle

```
application {
    executableDir = 'custom_bin_dir'
}
```

build.gradle.kts

```
application {
    executableDir = "custom_bin_dir"
}
```

Licensing

The Gradle start scripts that are bundled with your application are licensed under the Apache 2.0 Software License. This does not affect your application, which you can license as you choose.

Convention properties (deprecated)

This plugin also adds some convention properties to the project, which you can use to configure its behavior. These are **deprecated** and superseded by the extension described above. See the Project DSL documentation for information on them.

Unlike the extension properties, these properties appear as top-level project properties in the build script. For example, to change the application name you can just add the following to your build script:

build.gradle

```
application.applicationName = 'my-app'
```

build.gradle.kts

```
application.applicationName = "my-app"
```

The Base Plugin

The Base Plugin provides some tasks and conventions that are common to most builds and adds a structure to the build that promotes consistency in how they are run. Its most significant contribution is a set of *lifecycle* tasks that act as an umbrella for the more specific tasks provided by other plugins and build authors.

Usage

Example 599. Applying the Base Plugin

build.gradle

```
plugins {
    id 'base'
}
```

build.gradle.kts

```
plugins {
    base
}
```

Task

`clean` — `Delete`

Deletes the build directory and everything in it, i.e. the path specified by the Project.getBuildDir() project property.

`check` — ***lifecycle task***

Plugins and build authors should attach their verification tasks, such as ones that run tests, to this lifecycle task using `check.dependsOn(task)`.

`assemble` — ***lifecycle task***

Plugins and build authors should attach tasks that produce distributions and other consumable artifacts to this lifecycle task. For example, `jar` produces the consumable artifact for Java libraries. Attach tasks to this lifecycle task using `assemble.dependsOn(task)`.

`build` — ***lifecycle task***

Depends on: `check`, `assemble`

Intended to build everything, including running all tests, producing the production artifacts and generating documentation. You will probably rarely attach concrete tasks directly to `build` as `assemble` and `check` are typically more appropriate.

`buildConfiguration` — **task rule**

Assembles those artifacts attached to the named configuration. For example, `buildArchives` will execute any task that is required to create any artifact attached to the `archives` configuration.

`cleanTask` — **task rule**

Removes the defined outputs of a task, e.g. `cleanJar` will delete the JAR file produced by the `jar` task of the Java Plugin.

Dependency management

The Base Plugin adds no configurations for dependencies, but it does add the following configurations:

`default`

A fallback configuration used by consumer projects. Let's say you have project B with a project dependency on project A. Gradle uses some internal logic to determine which of project A's artifacts and dependencies are added to the specified configuration of project B. If no other factors apply — you don't need to worry what these are — then Gradle falls back to using everything in project A's `default` configuration.

New builds and plugins should not be using the `default` configuration! It remains for the reason of backwards compatibility.

`archives`

A standard configuration for the production artifacts of a project.

Note that the `assemble` task generates all artifacts that are attached to the `archives` configuration.

Contributed extensions

The Base Plugin adds the `base` extension to the project. This allows to configure the following properties inside a dedicated DSL block.

Example 600. Using the `base` extension

build.gradle

```
base {
    archivesName = "gradle"
    distsDirectory = layout.buildDirectory.dir('custom-dist')
    libsDirectory = layout.buildDirectory.dir('custom-libs')
}
```

build.gradle.kts

```
base {
    archivesName.set("gradle")
    distsDirectory.set(layout.buildDirectory.dir("custom-dist"))
    libsDirectory.set(layout.buildDirectory.dir("custom-libs"))
}
```

`archivesName` — **default:** `$project.name`

Provides the default AbstractArchiveTask.getArchiveBaseName() for archive tasks.

`distsDirectory` — **default:** `$buildDir/distributions`

Default name of the directory in which distribution archives, i.e. non-JARs, are created.

`libsDirectory` — **default:** `$buildDir/libs`

Default name of the directory in which library archives, i.e. JARs, are created.

The plugin also provides default values for the following properties on any task that extends AbstractArchiveTask:

`destinationDirectory`

Defaults to `distsDirectory` for non-JAR archives and `libsDirectory` for JARs and derivatives of JAR, such as WARs.

`archiveVersion`

Defaults to `$project.version` or 'unspecified' if the project has no version.

`archiveBaseName`

Defaults to `$archivesBaseName`.

Conventions (deprecated)

The Base Plugin also adds conventions related to the creation of archives, such as ZIPs, TARs and JARs. These are **deprecated** and superseded by the extension described above. See the

BasePluginConvention DSL documentation for information on them.

Build Init Plugin

The Build Init plugin can be used to create a new Gradle build. It supports creating brand new Gradle builds of various types as well as converting existing Apache Maven builds to Gradle.

Supported Gradle build types

The Build Init plugin supports generating various build *types*. These are listed below and more detail is available about each type in the following section.

Table 26. Build init types

Type	Description
pom	Converts an existing Apache Maven build to Gradle
basic	A basic, empty, Gradle build
java-application	A command-line application implemented in Java
java-gradle-plugin	A Gradle plugin implemented in Java
java-library	A Java library
kotlin-application	A command-line application implemented in Kotlin/JVM
kotlin-gradle-plugin	A Gradle plugin implemented in Kotlin/JVM
kotlin-library	A Kotlin/JVM library
groovy-application	A command-line application implemented in Groovy
groovy-gradle-plugin	A Gradle plugin implemented in Groovy
groovy-library	A Groovy library
scala-application	A Scala application
scala-library	A Scala library
cpp-application	A command-line application implemented in C++
cpp-library	A C++ library

Tasks

The plugin adds the following tasks to the project:

`init` — **InitBuild**

Depends on: `wrapper`

Generates a Gradle build.

`wrapper` — **Wrapper**

Generates Gradle wrapper files.

Gradle plugins usually need to be *applied* to a project before they can be used (see Using plugins). However, the Build Init plugin is automatically applied to the root project of every build, which means you do not need to apply it explicitly in order to use it. You can simply execute the task named `init` in the directory where you would like to create the Gradle build. There is no need to create a "stub" `build.gradle` file in order to apply the plugin.

The Build Init plugin also uses the `wrapper` task to generate the Gradle Wrapper files for the build.

What to create

The simplest, and recommended, way to use the `init` task is to run `gradle init` from an interactive console. Gradle will list the available build types and ask you to select one. It will then ask some additional questions to allow you to fine-tune the result.

There are several command-line options available for the `init` task that control what it will generate. You can use these when Gradle is not running from an interactive console.

The build type can be specified by using the `--type` command-line option. For example, to create a Java library project run: `gradle init --type java-library`.

If a `--type` option is not provided, Gradle will attempt to infer the type from the environment. For example, it will infer a type of "`pom`" if it finds a `pom.xml` file to convert to a Gradle build. If the type could not be inferred, the type "`basic`" will be used.

The `init` task also supports generating build scripts using either the Gradle Groovy DSL or the Gradle Kotlin DSL. The build script DSL defaults to the Groovy DSL for most build types and to the Kotlin DSL for Kotlin build types. The DSL can be selected by using the `--dsl` command-line option. For example, to create a Java library project with Kotlin DSL build scripts run: `gradle init --type java-library --dsl kotlin`.

You can change the name of the generated project using the `--project-name` option. It defaults to the name of the directory where the `init` task is run.

You can change the package used for generated source files using the `--package` option. It defaults to the project name.

If the `--incubating` option is provided, Gradle will generate build scripts which may use the latest versions of APIs, which are marked `@Incubating` and remain subject to change.

All build types also setup the Gradle wrapper.

Build init types

`pom` build type (Maven conversion)

The "`pom`" type can be used to convert an Apache Maven build to a Gradle build. This works by converting the POM to one or more Gradle files. It is only able to be used if there is a valid "`pom.xml`" file in the directory that the `init` task is invoked in or, if invoked via the "-p" command line option, in the specified project directory. This "`pom`" type will be automatically inferred if such a file exists.

The Maven conversion implementation was inspired by the maven2gradle tool that was originally developed by Gradle community members.

The conversion process has the following features:

- Uses effective POM and effective settings (support for POM inheritance, dependency management, properties)
- Supports both single module and multimodule projects
- Supports custom module names (that differ from directory names)
- Generates general metadata - id, description and version
- Applies Maven Publish, Java Library and War Plugins (as needed)
- Supports packaging war projects as jars if needed
- Generates dependencies (both external and inter-module)
- Generates download repositories (inc. local Maven repository)
- Adjusts Java compiler settings
- Supports packaging of sources, tests, and javadocs
- Supports TestNG runner
- Generates global exclusions from Maven enforcer plugin settings
- Provides an option for handling Maven repositories located at URLs using `http`

The `--insecure-protocol` option

This option is used to tell the conversion process how to handle converting Maven repositories located at insecure `http` URLs. Insecure Repositories Set the --insecure-protocol option. The default value is `warn`.

Available values are:

- `fail` - Abort the build immediately upon encountering an insecure repository URL.
- `allow` - Automatically sets the `allowInsecureProtocol` property to `true` for the Maven repository URL in the generated Gradle build script.
- `warn` - Emits a warning about each insecure URL. Generates commented-out lines to enable each repository, as per the `allow` option. You will have to opt-in by editing the generated script and uncommenting each repository URL, or else the Gradle build will fail.
- `upgrade` - Convert `http` URLs to `https` URLs automatically.

Compile-time dependencies

Maven automatically exposes dependencies using its implicit `compile` scope to the consumers of that project. This behavior is undesirable, and Gradle takes steps to help library authors reduce their API footprint using the `api` and `implementation` configurations of the `java-library` plugin.

Nevertheless, many Maven projects rely on this *leaking* behavior. As such, the `init` task will map `compile`-scoped dependencies to the `api` configuration in the generated Gradle build script. The dependencies of the resulting Gradle project will most closely match the exposed dependencies of the existing Maven project; however, post-conversion to Gradle we strongly encourage moving as many `api` dependencies to the `implementation` configuration as possible. This has several benefits:

- Library maintainability - By exposing fewer transitive dependencies to consumers, library maintainers can add or remove dependencies without fear of causing compile-time breakages for consumers.
- Consumers' dependency hygiene - Leveraging the `implementation` configuration in a library prevents its consumers from implicitly relying on the library's transitive dependencies at compile-time, which is considered a bad practice.
- Increased compile avoidance - Reducing the number of transitive dependencies leaked from a project also reduces the likelihood that an ABI change will trigger recompilation of consumers. Gradle will also spend less time indexing the dependencies for its up-to-date checks.
- Compilation speed increase - Reducing the number of transitive dependencies leaked from a project aids the compiler process of its consumers as there are fewer libraries to classload and fewer namespaces for Gradle's incremental compiler to track.

See the API and implementation separation and Compilation avoidance sections for more information.

`java-application` build type

The "`java-application`" build type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`application`" plugin to produce a command-line application implemented in Java
- Uses the "`mavenCentral`" dependency repository
- Uses JUnit 4 for testing
- Has directories in the conventional locations for source code
- Contains a sample class and unit test, if there are no existing source or test files

Alternative test framework can be specified by supplying a `--test-framework` argument value. To use a different test framework, execute one of the following commands:

- `gradle init --type java-application --test-framework junit-jupiter`: Uses JUnit Jupiter for testing instead of JUnit 4
- `gradle init --type java-application --test-framework spock`: Uses Spock for testing instead of JUnit 4

- `gradle init --type java-application --test-framework testng`: Uses TestNG for testing instead of JUnit 4

`java-library` build type

The "`java-library`" build type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`java`" plugin to produce a library implemented in Java
- Uses the "`mavenCentral`" dependency repository
- Uses JUnit 4 for testing
- Has directories in the conventional locations for source code
- Contains a sample class and unit test, if there are no existing source or test files

Alternative test framework can be specified by supplying a `--test-framework` argument value. To use a different test framework, execute one of the following commands:

- `gradle init --type java-library --test-framework junit-jupiter`: Uses JUnit Jupiter for testing instead of JUnit 4
- `gradle init --type java-library --test-framework spock`: Uses Spock for testing instead of JUnit 4
- `gradle init --type java-library --test-framework testng`: Uses TestNG for testing instead of JUnit 4

`java-gradle-plugin` build type

The "`java-gradle-plugin`" build type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`java-gradle-plugin`" plugin to produce a Gradle plugin implemented in Java
- Uses the "`mavenCentral`" dependency repository
- Uses JUnit 4 and TestKit for testing
- Has directories in the conventional locations for source code
- Contains a sample class and unit test, if there are no existing source or test files

`kotlin-application` build type

The "`kotlin-application`" build type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`org.jetbrains.kotlin.jvm`" and "`application`" plugins to produce a command-line application implemented in Kotlin
- Uses the "`mavenCentral`" dependency repository

- Uses Kotlin 1.x
- Uses Kotlin test library for testing
- Has directories in the conventional locations for source code
- Contains a sample Kotlin class and an associated Kotlin test class, if there are no existing source or test files

`kotlin-library` build type

The "`kotlin-library`" build type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`org.jetbrains.kotlin.jvm`" plugin to produce a library implemented in Kotlin
- Uses the "`mavenCentral`" dependency repository
- Uses Kotlin 1.x
- Uses Kotlin test library for testing
- Has directories in the conventional locations for source code
- Contains a sample Kotlin class and an associated Kotlin test class, if there are no existing source or test files

`kotlin-gradle-plugin` build type

The "`kotlin-gradle-plugin`" build type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`java-gradle-plugin`" and "`org.jetbrains.kotlin.jvm`" plugins to produce a Gradle plugin implemented in Kotlin
- Uses the "`mavenCentral`" dependency repository
- Uses Kotlin 1.x
- Uses Kotlin test library and TestKit for testing
- Has directories in the conventional locations for source code
- Contains a sample class and unit test, if there are no existing source or test files

`scala-application` build type

The "`scala-application`" build type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`scala`" plugin to produce an application implemented in Scala
- Uses the "`mavenCentral`" dependency repository
- Uses Scala 2.13
- Uses ScalaTest for testing

- Has directories in the conventional locations for source code
- Contains a sample Scala class and an associated ScalaTest test suite, if there are no existing source or test files

`scala-library` build type

The "`scala-library`" build type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`scala`" plugin to produce a library implemented in Scala
- Uses the "`mavenCentral`" dependency repository
- Uses Scala 2.13
- Uses ScalaTest for testing
- Has directories in the conventional locations for source code
- Contains a sample Scala class and an associated ScalaTest test suite, if there are no existing source or test files

`groovy-library` build type

The "`groovy-library`" build type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`groovy`" plugin to produce a library implemented in Groovy
- Uses the "`mavenCentral`" dependency repository
- Uses Groovy 2.x
- Uses Spock testing framework for testing
- Has directories in the conventional locations for source code
- Contains a sample Groovy class and an associated Spock specification, if there are no existing source or test files

`groovy-application` build type

The "`groovy-application`" build type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`application`" and "`groovy`" plugins to produce a command-line application implemented in Groovy
- Uses the "`mavenCentral`" dependency repository
- Uses Groovy 2.x
- Uses Spock testing framework for testing
- Has directories in the conventional locations for source code

- Contains a sample Groovy class and an associated Spock specification, if there are no existing source or test files

`groovy-gradle-plugin` build type

The "`groovy-gradle-plugin`" build type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`java-gradle-plugin`" and "`groovy`" plugins to produce a Gradle plugin implemented in Groovy
- Uses the "`mavenCentral`" dependency repository
- Uses Groovy 2.x
- Uses Spock testing framework and TestKit for testing
- Has directories in the conventional locations for source code
- Contains a sample class and unit test, if there are no existing source or test files

`cpp-application` build type

The "`cpp-application`" build type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`cpp-application`" plugin to produce a command-line application implemented in C++
- Uses the "`cpp-unit-test`" plugin to build and run simple unit tests
- Has directories in the conventional locations for source code
- Contains a sample C++ class, a private header file and an associated test class, if there are no existing source or test files

`cpp-library` build type

The "`cpp-library`" build type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`cpp-library`" plugin to produce a C++ library
- Uses the "`cpp-unit-test`" plugin to build and run simple unit tests
- Has directories in the conventional locations for source code
- Contains a sample C++ class, a public header file and an associated test class, if there are no existing source or test files

`basic` build type

The "`basic`" build type is useful for creating a new Gradle build. It creates sample settings and build files, with comments and links to help get started.

This type is used when no type was explicitly specified, and no type could be inferred.

The Checkstyle Plugin

The Checkstyle plugin performs quality checks on your project's Java source files using Checkstyle and generates reports from these checks.

Usage

To use the Checkstyle plugin, include the following in your build script:

Example 601. Using the Checkstyle plugin

build.gradle

```
plugins {
    id 'checkstyle'
}
```

build.gradle.kts

```
plugins {
    checkstyle
}
```

The plugin adds a number of tasks to the project that perform the quality checks. You can execute the checks by running `gradle check`.

Note that Checkstyle will run with the same Java version used to run Gradle.

Tasks

The Checkstyle plugin adds the following tasks to the project:

`checkstyleMain` — **Checkstyle**

Depends on: `classes`

Runs Checkstyle against the production Java source files.

`checkstyleTest` — **Checkstyle**

Depends on: `testClasses`

Runs Checkstyle against the test Java source files.

`checkstyleSourceSet` — **Checkstyle**

Depends on: `sourceSetClasses`

Runs Checkstyle against the given source set's Java source files.

Dependencies added to other tasks

The Checkstyle plugin adds the following dependencies to tasks defined by the Java plugin.

`check`

Depends on: All Checkstyle tasks, including `checkstyleMain` and `checkstyleTest`.

Project layout

By default, the Checkstyle plugin expects configuration files to be placed in the root project, but this can be changed.

```
<root>
 └─── config
     └─── checkstyle           ①
         └─── checkstyle.xml   ②
         └─── suppressions.xml
```

① Checkstyle configuration files go here

② Primary Checkstyle configuration file

Dependency management

The Checkstyle plugin adds the following dependency configurations:

Table 27. Checkstyle plugin - dependency configurations

Name	Meaning
`checkstyle`	The Checkstyle libraries to use

Configuration

See the CheckstyleExtension class in the API documentation.

Built-in variables

The Checkstyle plugin defines a `config_loc` property that can be used in Checkstyle configuration files to define paths to other configuration files like `suppressions.xml`.

Example 602. Using the config_loc property

checkstyle.xml

```
<module name="SuppressionFilter">
    <property name="file" value="${config_loc}/suppressions.xml"/>
</module>
```

Customizing the HTML report

The HTML report generated by the Checkstyle task can be customized using a XSLT stylesheet, for example to highlight specific errors or change its appearance:

Example 603. Customizing the HTML report

build.gradle

```
tasks.withType(Checkstyle) {
    reports {
        xml.required = false
        html.required = true
        html.stylesheet resources.text.fromFile('config/xsl/checkstyle-
custom.xsl')
    }
}
```

build.gradle.kts

```
tasks.withType<Checkstyle>().configureEach {
    reports {
        xml.required.set(false)
        html.required.set(true)
        html.stylesheet = resources.text.fromFile("config/xsl/checkstyle-
custom.xsl")
    }
}
```

View a sample Checkstyle stylesheet.

The CodeNarc Plugin

The CodeNarc plugin performs quality checks on your project's Groovy source files using CodeNarc and generates reports from these checks.

Usage

To use the CodeNarc plugin, include the following in your build script:

Example 604. Using the CodeNarc plugin

build.gradle

```
plugins {
    id 'codenarc'
}
```

build.gradle.kts

```
plugins {
    codenarc
}
```

The plugin adds a number of tasks to the project that perform the quality checks when used with the Groovy Plugin. You can execute the checks by running `gradle check`.

Tasks

The CodeNarc plugin adds the following tasks to the project:

`codenarcMain` — **CodeNarc**

Runs CodeNarc against the production Groovy source files.

`codenarcTest` — **CodeNarc**

Runs CodeNarc against the test Groovy source files.

`codenarcSourceSet` — **CodeNarc**

Runs CodeNarc against the given source set's Groovy source files.

Dependencies added to other tasks

The CodeNarc plugin adds the following dependencies to tasks defined by the Groovy plugin.

`check`

Depends on: All CodeNarc tasks, including `codenarcMain` and `codenarcTest`.

Project layout

The CodeNarc plugin expects the following project layout:

```
<root>
 └──── config
     └──── codenarc          ①
         └──── codenarc.xml  ②
```

① CodeNarc configuration files go here

② Primary CodeNarc configuration file

Dependency management

The CodeNarc plugin adds the following dependency configurations:

Table 28. CodeNarc plugin - dependency configurations

Name	Meaning
`codenarc`	The CodeNarc libraries to use

Configuration

See the CodeNarcExtension class in the API documentation.

The Distribution Plugin

The Distribution Plugin facilitates building archives that serve as distributions of the project. Distribution archives typically contain the executable application and other supporting files, such as documentation.

Usage

To use the Distribution Plugin, include the following in your build script:

Example 605. Using the Distribution Plugin

build.gradle

```
plugins {
    id 'distribution'
}
```

build.gradle.kts

```
plugins {
    distribution
}
```

The plugin adds an extension named `distributions` of type DistributionContainer to the project. It also creates a single distribution in the distributions container extension named `main`. If your build only produces one distribution you only need to configure this distribution (or use the defaults).

You can run `gradle distZip` to package the main distribution as a ZIP, or `gradle distTar` to create a TAR file. To build both types of archives just run `gradle assembleDist`. The files will be created at `$buildDir/distributions/${project.name}-${project.version}.«ext»`.

You can run `gradle installDist` to assemble the uncompressed distribution into `$buildDir/install/${project.name}`.

Tasks

The Distribution Plugin adds a number of tasks to your project, as shown below.

`distZip` — **Zip**

Creates a ZIP archive of the distribution contents.

`distTar` — **Task**

Creates a TAR archive of the distribution contents.

`assembleDist` — **Task**

Depends on: `distTar`, `distZip`

Creates ZIP and TAR archives of the distribution contents.

`installDist` — **Sync**

Assembles the distribution content and installs it on the current machine.

For each additional distribution you add to the project, the Distribution Plugin adds the following tasks, where *distributionName* comes from Distribution.getName():

`distributionNameDistZip` — **Zip**

Creates a ZIP archive of the distribution contents.

`distributionNameDistTar` — **Tar**

Creates a TAR archive of the distribution contents.

`assembleDistributionNameDist` — **Task**

Depends on: `distributionNameDistTar`, `distributionNameDistZip`

Creates ZIP and TAR archives of the distribution contents.

`installDistributionNameDist` — **Sync**

Assembles the distribution content and installs it on the current machine.

The following sample creates a `custom` distribution that will cause four additional tasks to be added to the project: `customDistZip`, `customDistTar`, `assembleCustomDist`, and `installCustomDist`:

Example 606. Adding extra distributions

build.gradle

```
distributions {
    custom {
        // configure custom distribution
    }
}
```

build.gradle.kts

```
distributions {
    create("custom") {
        // configure custom distribution
    }
}
```

Given that the project name is `myproject` and version `1.2`, running `gradle customDistZip` will produce a ZIP file named `myproject-custom-1.2.zip`.

Running `gradle installCustomDist` will install the distribution contents into `$buildDir/install/custom`.

Distribution contents

All of the files in the `src/$distribution.name/dist` directory will automatically be included in the distribution. You can add additional files by configuring the Distribution object that is part of the

container.

Example 607. Configuring the main distribution

build.gradle

```
distributions {
    main {
        distributionBaseName = 'someName'
        distributionClassifier = 'classifier'
        contents {
            from 'src/readme'
        }
    }
}
```

build.gradle.kts

```
distributions {
    main {
        distributionBaseName.set("someName")
        distributionClassifier.set("classifier")
        contents {
            from("src/readme")
        }
    }
}
```

In the example above, the content of the `src/readme` directory will be included in the distribution (along with the files in the `src/main/dist` directory which are added by default).

The `distributionBaseName` and `distributionClassifier` properties have also been changed. This will cause the distribution archives to be created with a different name.

Publishing

A distribution can be published using the Ivy Publish Plugin or Maven Publish Plugin.

Using the Ivy/Maven Publish Plugins

To publish a distribution to an Ivy repository with the Ivy Publish Plugin, add one or both of its archive tasks to an IvyPublication. The following sample demonstrates how to add the ZIP archive of the `main` distribution and the TAR archive of the `custom` distribution to the `myDistribution` publication:

Example 608. Adding distribution archives to an Ivy publication

build.gradle

```
plugins {
    id 'ivy-publish'
}

publishing {
    publications {
        myDistribution(IvyPublication) {
            artifact distZip
            artifact customDistTar
        }
    }
}
```

build.gradle.kts

```
plugins {
    `ivy-publish`
}

publishing {
    publications {
        create<IvyPublication>("myDistribution") {
            artifact(tasks.distZip.get())
            artifact(tasks["customDistTar"])
        }
    }
}
```

Similarly, to publish a distribution to a Maven repository using the Maven Publish Plugin, add one or both of its archive tasks to a MavenPublication as follows:

Example 609. Adding distribution archives to a Maven publication

build.gradle

```
plugins {
    id 'maven-publish'
}

publishing {
    publications {
        myDistribution(MavenPublication) {
            artifact distZip
            artifact customDistTar
        }
    }
}
```

build.gradle.kts

```
plugins {
    `maven-publish`
}

publishing {
    publications {
        create<MavenPublication>("myDistribution") {
            artifact(tasks.distZip)
            artifact(tasks["customDistTar"])
        }
    }
}
```

The Ear Plugin

The Ear plugin adds support for assembling web application EAR files. It adds a default EAR archive task. It doesn't require the Java plugin, but for projects that also use the Java plugin it disables the default JAR archive generation.

Usage

To use the Ear plugin, include the following in your build script:

Example 610. Using the Ear plugin

build.gradle

```
plugins {
    id 'ear'
}
```

build.gradle.kts

```
plugins {
    ear
}
```

Tasks

The Ear plugin adds the following tasks to the project.

`ear` — **Ear**

Depends on: `compile` (only if the Java plugin is also applied)

Assembles the application EAR file.

Dependencies added to other tasks

The Ear plugin adds the following dependencies to tasks added by the Base Plugin.

`assemble`

Depends on: `ear`.

Project layout

```
.
└─── src
    └─── main
        └─── application ①
```

① Ear resources, such as a META-INF directory

Dependency management

The Ear plugin adds two dependency configurations: `deploy` and `earlib`. All dependencies in the `deploy` configuration are placed in the root of the EAR archive, and are *not* transitive. All dependencies in the `earlib` configuration are placed in the 'lib' directory in the EAR archive and *are*

transitive.

Convention properties (ear)

`appDirName` — `String`

The name of the application source directory, relative to the project directory. *Default value: `src/main/application`*.

`libDirName` — `String`

The name of the lib directory inside the generated EAR. *Default value: `lib`*.

`deploymentDescriptor` — **DeploymentDescriptor**

Metadata to generate a deployment descriptor file, e.g. `application.xml`. *Default value: A deployment descriptor with sensible defaults named `application.xml`. If this file already exists in the `appDirName/META-INF` then the existing file contents will be used and the explicit configuration in the* `ear.deploymentDescriptor` will be ignored.

`generateDeploymentDescriptor` — `Boolean`

Specifies if deploymentDescriptor should be generated. *Default value: `true`*.

These properties are provided by a EarPluginConvention convention object.

Configuring the ear tasks via the plugin's convention properties is **deprecated**. If you need to change from the default values, configure the appropriate tasks directly. If you want to configure all `Ear` tasks in the project, use tasks.withType(Ear.class).configureEach(...).

Ear

The default behavior of the Ear task is to copy the content of `src/main/application` to the root of the archive. If your `application` directory doesn't contain a `META-INF/application.xml` deployment descriptor then one will be generated for you.

The Ear class in the API documentation has additional useful information.

Customizing

Here is an example with the most important customization options:

Example 611. Customization of ear plugin

build.gradle

```
plugins {
    id 'ear'
    id 'java'
}

repositories { mavenCentral() }

dependencies {
    // The following dependencies will be the ear modules and
    // will be placed in the ear root
    deploy project(path: ':war', configuration: 'archives')

    // The following dependencies will become ear libs and will
    // be placed in a dir configured via the libDirName property
    earlib group: 'log4j', name: 'log4j', version: '1.2.15', ext: 'jar'
}

ear {
    appDirectory = file('src/main/app')  // use application metadata found in
this folder
    // put dependent libraries into APP-INF/lib inside the generated EAR
    libDirName 'APP-INF/lib'
    deploymentDescriptor {  // custom entries for application.xml:
//      fileName = "application.xml"  // same as the default value
//      version = "6"  // same as the default value
        applicationName = "customear"
        initializeInOrder = true
        displayName = "Custom Ear"  // defaults to project.name
        // defaults to project.description if not set
        description = "My customized EAR for the Gradle documentation"
//      libraryDirectory = "APP-INF/lib"  // not needed, above libDirName
setting does this
//      module("my.jar", "java")  // won't deploy as my.jar isn't deploy
dependency
//      webModule("my.war", "/")  // won't deploy as my.war isn't deploy
dependency
        securityRole "admin"
        securityRole "superadmin"
        withXml { provider -> // add a custom node to the XML
            provider.asNode().appendNode("data-source", "my/data/source")
        }
    }
}
```

build.gradle.kts

```
plugins  {
    ear
    java
}

repositories { mavenCentral() }

dependencies {
    // The following dependencies will be the ear modules and
    // will be placed in the ear root
    deploy(project(path = ":war", configuration = "archives"))

    // The following dependencies will become ear libs and will
    // be placed in a dir configured via the libDirName property
    earlib(group = "log4j", name = "log4j", version = "1.2.15", ext = "jar")
}

tasks.named<Ear>("ear") {
    appDirectory.set(file("src/main/app"))  // use application metadata found
in this folder
}

ear {
    // put dependent libraries into APP-INF/lib inside the generated EAR
    libDirName = "APP-INF/lib"
    deploymentDescriptor {  // custom entries for application.xml:
//      fileName = "application.xml"  // same as the default value
//      version = "6"  // same as the default value
        applicationName = "customear"
        initializeInOrder = true
        displayName = "Custom Ear"  // defaults to project.name
        // defaults to project.description if not set
        description = "My customized EAR for the Gradle documentation"
//      libraryDirectory = "APP-INF/lib"  // not needed, above libDirName
setting does this
//      module("my.jar", "java")  // won't deploy as my.jar isn't deploy
dependency
//      webModule("my.war", "/")  // won't deploy as my.war isn't deploy
dependency
        securityRole("admin")
        securityRole("superadmin")
        withXml { // add a custom node to the XML
            asElement().apply {
                appendChild(ownerDocument.createElement("data-source").apply
{ textContent = "my/data/source" })
            }
        }
    }
}
```

You can also use customization options that the Ear task provides, such as `from` and `metaInf`.

Using custom descriptor file

You may already have appropriate settings in a `application.xml` file and want to use that instead of configuring the `ear.deploymentDescriptor` section of the build script. To accommodate that goal, place the `META-INF/application.xml` in the right place inside your source folders (see the `appDirName` property). The file contents will be used and the explicit configuration in the `ear.deploymentDescriptor` will be ignored.

The Eclipse Plugins

The Eclipse plugins generate files that are used by the Eclipse IDE, thus making it possible to import the project into Eclipse (`File` - `Import…` - `Existing Projects into Workspace`).

The `eclipse-wtp` is automatically applied whenever the `eclipse` plugin is applied to a War or Ear project. For utility projects (i.e. Java projects used by other web projects), you need to apply the `eclipse-wtp` plugin explicitly.

What exactly the `eclipse` plugin generates depends on which other plugins are used:

Table 29. Eclipse plugin behavior

Plugin	Description
None	Generates minimal `.project` file.
Java	Adds Java configuration to `.project`. Generates `.classpath` and JDT settings file.
Groovy	Adds Groovy configuration to `.project` file.
Scala	Adds Scala support to `.project` and `.classpath` files.
War	Adds web application support to `.project` file.
Ear	Adds ear application support to `.project` file.

The `eclipse-wtp` plugin generates all WTP settings files and enhances the `.project` file. If a Java or War is applied, `.classpath` will be extended to get a proper packaging structure for this utility library or web application project.

Both Eclipse plugins are open to customization and provide a standardized set of hooks for adding and removing content from the generated files.

Usage

To use either the Eclipse or the Eclipse WTP plugin, include one of the lines in your build script:

Example 612. Using the Eclipse plugin

build.gradle

```
plugins {
    id 'eclipse'
}
```

build.gradle.kts

```
plugins {
    eclipse
}
```

Example 613. Using the Eclipse WTP plugin

build.gradle

```
plugins {
    id 'eclipse-wtp'
}
```

build.gradle.kts

```
plugins {
    `eclipse-wtp`
}
```

Note: Internally, the `eclipse-wtp` plugin also applies the `eclipse` plugin so you don't need to apply both.

Both Eclipse plugins add a number of tasks to your projects. The main tasks that you will use are the `eclipse` and `cleanEclipse` tasks.

Tasks

The Eclipse plugins add the tasks shown below to a project.

Eclipse Plugin tasks

`eclipse` — **Task**

Depends on: all Eclipse configuration file generation tasks

Generates all Eclipse configuration files

`cleanEclipse` — **Delete**

Depends on: all Eclipse configuration file clean tasks

Removes all Eclipse configuration files

`cleanEclipseProject` — **Delete**

Removes the `.project` file.

`cleanEclipseClasspath` — **Delete**

Removes the `.classpath` file.

`cleanEclipseJdt` — **Delete**

Removes the `.settings/org.eclipse.jdt.core.prefs` file.

`eclipseProject` — **GenerateEclipseProject**

Generates the `.project` file.

`eclipseClasspath` — **GenerateEclipseClasspath**

Generates the `.classpath` file.

`eclipseJdt` — **GenerateEclipseJdt**

Generates the `.settings/org.eclipse.jdt.core.prefs` file.

Eclipse WTP Plugin — additional tasks

`cleanEclipseWtpComponent` — **Delete**

Removes the `.settings/org.eclipse.wst.common.component` file.

`cleanEclipseWtpFacet` — **Delete**

Removes the `.settings/org.eclipse.wst.common.project.facet.core.xml` file.

`eclipseWtpComponent` — **GenerateEclipseWtpComponent**

Generates the `.settings/org.eclipse.wst.common.component` file.

`eclipseWtpFacet` — **GenerateEclipseWtpFacet**

Generates the `.settings/org.eclipse.wst.common.project.facet.core.xml` file.

Configuration

Table 30. Configuration of the Eclipse plugins

Model	Reference name	Description
EclipseModel	`eclipse`	Top level element that enables configuration of the Eclipse plugin in a DSL-friendly fashion.
EclipseProject	`eclipse.project`	Allows configuring project information
EclipseClasspath	`eclipse.classpath`	Allows configuring classpath information.
EclipseJdt	`eclipse.jdt`	Allows configuring jdt information (source/target Java compatibility).
EclipseWtpComponent	`eclipse.wtp.component`	Allows configuring wtp component information only if `eclipse-wtp` plugin was applied.
EclipseWtpFacet	`eclipse.wtp.facet`	Allows configuring wtp facet information only if `eclipse-wtp` plugin was applied.

Customizing the generated files

The Eclipse plugins allow you to customize the generated metadata files. The plugins provide a DSL for configuring model objects that model the Eclipse view of the project. These model objects are then merged with the existing Eclipse XML metadata to ultimately generate new metadata. The model objects provide lower level hooks for working with domain objects representing the file content before and after merging with the model configuration. They also provide a very low level hook for working directly with the raw XML for adjustment before it is persisted, for fine tuning and configuration that the Eclipse and Eclipse WTP plugins do not model.

Merging

Sections of existing Eclipse files that are also the target of generated content will be amended or overwritten, depending on the particular section. The remaining sections will be left as-is.

Disabling merging with a complete rewrite

To completely rewrite existing Eclipse files, execute a clean task together with its corresponding generation task, like "`gradle cleanEclipse eclipse`" (in that order). If you want to make this the default behavior, add "`tasks.eclipse.dependsOn(cleanEclipse)`" to your build script. This makes it unnecessary to execute the clean task explicitly.

This strategy can also be used for individual files that the plugins would generate. For instance, this can be done for the "`.classpath`" file with "`gradle cleanEclipseClasspath eclipseClasspath`".

Hooking into the generation lifecycle

The Eclipse plugins provide objects modeling the sections of the Eclipse files that are generated by Gradle. The generation lifecycle is as follows:

1. The file is read; or a default version provided by Gradle is used if it does not exist
2. The `beforeMerged` hook is executed with a domain object representing the existing file

3. The existing content is merged with the configuration inferred from the Gradle build or defined explicitly in the eclipse DSL
4. The `whenMerged` hook is executed with a domain object representing contents of the file to be persisted
5. The `withXml` hook is executed with a raw representation of the XML that will be persisted
6. The final XML is persisted

Advanced configuration hooks

The following list covers the domain object used for each of the Eclipse model types:

EclipseProject

- `beforeMerged { Project arg -> … }`
- `whenMerged { Project arg -> … }`
- `withXml { XmlProvider arg -> … }`

EclipseClasspath

- `beforeMerged { Classpath arg -> … }`
- `whenMerged { Classpath arg -> … }`
- `withXml { XmlProvider arg -> … }`

EclipseWtpComponent

- `beforeMerged { WtpComponent arg -> … }`
- `whenMerged { WtpComponent arg -> … }`
- `withXml { XmlProvider arg -> … }`

EclipseWtpFacet

- `beforeMerged { WtpFacet arg -> … }`
- `whenMerged { WtpFacet arg -> … }`
- `withXml { XmlProvider arg -> … }`

EclipseJdt

- `beforeMerged { Jdt arg -> … }`
- `whenMerged { Jdt arg -> … }`
- `withProperties { arg -> }` argument type ⇒ `java.util.Properties`

Partial overwrite of existing content

A complete overwrite causes all existing content to be discarded, thereby losing any changes made directly in the IDE. Alternatively, the `beforeMerged` hook makes it possible to overwrite just certain parts of the existing content. The following example removes all existing dependencies from the `Classpath` domain object:

Example 614. Partial Overwrite for Classpath

build.gradle

```
eclipse.classpath.file {
    beforeMerged { classpath ->
        classpath.entries.removeAll { entry -> entry.kind == 'lib' || entry
.kind == 'var' }
    }
}
```

build.gradle.kts

```
import org.gradle.plugins.ide.eclipse.model.Classpath

eclipse.classpath.file {
    beforeMerged(Action<Classpath> {
        entries.removeAll { entry -> entry.kind == "lib" || entry.kind ==
"var" }
    })
}
```

The resulting `.classpath` file will only contain Gradle-generated dependency entries, but not any other dependency entries that may have been present in the original file. (In the case of dependency entries, this is also the default behavior.) Other sections of the `.classpath` file will be either left as-is or merged. The same could be done for the natures in the `.project` file:

Example 615. Partial Overwrite for Project

build.gradle

```
eclipse.project.file.beforeMerged { project ->
    project.natures.clear()
}
```

build.gradle.kts

```
import org.gradle.plugins.ide.eclipse.model.Project

eclipse.project.file.beforeMerged(Action<Project> {
    natures.clear()
})
```

Modifying the fully populated domain objects

The `whenMerged` hook allows to manipulate the fully populated domain objects. Often this is the preferred way to customize Eclipse files. Here is how you would export all the dependencies of an Eclipse project:

Example 616. Export Classpath Entries

build.gradle

```
eclipse.classpath.file {
    whenMerged { classpath ->
        classpath.entries.findAll { entry -> entry.kind == 'lib' }*.exported
= false
    }
}
```

build.gradle.kts

```
import org.gradle.plugins.ide.eclipse.model.AbstractClasspathEntry
import org.gradle.plugins.ide.eclipse.model.Classpath

eclipse.classpath.file {
    whenMerged(Action<Classpath> { ->
        entries.filter { entry -> entry.kind == "lib" }
            .forEach { (it as AbstractClasspathEntry).isExported = false }
    })
}
```

Modifying the XML representation

The `withXml` hook allows to manipulate the in-memory XML representation just before the file gets written to disk. Although Groovy's XML support and Kotlin's extension functions make up for a lot, this approach is less convenient than manipulating the domain objects. In return, you get total control over the generated file, including sections not modeled by the domain objects.

Example 617. Customizing the XML

build.gradle

```
eclipse.wtp.facet.file.withXml { provider ->
    provider.asNode().fixed.find { it.@facet == 'jst.java' }.@facet =
'jst2.java'
}
```

build.gradle.kts

```
import org.w3c.dom.Element

eclipse.wtp.facet.file.withXml(Action<XmlProvider> {
    fun Element.firstElement(predicate: Element.() -> Boolean) =
        childNodes
            .run { (0 until length).map(::item) }
            .filterIsInstance<Element>()
            .first { it.predicate() }

    asElement()
        .firstElement { tagName === "fixed" && getAttribute("facet") ==
"jst.java" }
        .setAttribute("facet", "jst2.java")
})
```

Separation of test classpath entries

Eclipse defines only one classpath per project which implies limitations on how Gradle projects can be mapped. Eclipse 4.8 introduced the concept of test sources. This feature allows the Eclipse plugin to define better separation between test and non-test sources.

The Eclipse classpath consists of classpath entries: source directories, jar files, project dependencies, etc. Each classpath entry can have a a list of classpath attributes, where the attributes are string key-value pairs. There are two classpath attribute relevant for test sources: `test` and `without_test_code`, both of which can have `true` or `false` as values.

If a source directory has the `test=true` classpath attribute then the contents are considered test sources. Test sources have access to non-test sources, but non-test sources don't have access to test sources. Similarly, test sources can only reference classes from a jar file if the jar file has the `test=true` classpath attribute. For project dependencies if the `test=true` attribute is present the classes in the target project are visible to test sources. If the `without_test_code=false` attribute is present then the test sources of the target project are also accessible. By default, test code is not available (which can be thought of as implicitly having `without_test_code=true`).

The following rules apply for the test attribute declaration when generating the Eclipse classpath:

- Source sets and dependency configurations are categorized as test if their name contain the 'test' substring, irrespective of case.
- All source sets and dependency configurations defined by the JVM test suite plugin are categorized as test.
- Source directories have the `test=true` classpath attribute if the container source set is a test source set.
- Jar files and project dependencies have the `test=true` attribute if they are present only in test dependency configurations.
- Project dependencies have the `without_test_sources=false` attribute if the target project applies the java-test-fixtures plugin.

You can customize the test source sets and configurations:

Example 618. Customizing test sources

build.gradle

```
eclipse {
    classpath {
        testSourceSets = testSourceSets.get() + [sourceSets.integTest]
        testConfigurations = testConfigurations.get() + [configurations
.functional]
    }
}
```

build.gradle.kts

```
eclipse {
    classpath {
        testSourceSets.set(testSourceSets.get() + setOf(integTest))
        testConfigurations.set(testConfigurations.get() + setOf(functional))
    }
}
```

TIP Consider migrating away from manual definition of additional test source sets and towards the use of the JVM Test Suite Plugin in scenarios like this.

You can also customize if a project should expose test sources to upstream project dependencies via the `containsTestFixtures` property.

Example 619. Customizing test fixtures

build.gradle

```
eclipse {
    classpath {
        containsTestFixtures = true
    }
}
```

build.gradle.kts

```
eclipse {
    classpath {
        containsTestFixtures.set(true)
    }
}
```

Note, that this configuration also applies to the classpath of Buildship projects.

The Groovy Plugin

The Groovy plugin extends the Java plugin to add support for Groovy projects. It can deal with Groovy code, mixed Groovy and Java code, and even pure Java code (although we don't necessarily recommend to use it for the latter). The plugin supports *joint compilation*, which allows you to freely mix and match Groovy and Java code, with dependencies in both directions. For example, a Groovy class can extend a Java class that in turn extends a Groovy class. This makes it possible to use the best language for the job, and to rewrite any class in the other language if needed.

Note that if you want to benefit from the API / implementation separation, you can also apply the `java-library` plugin to your Groovy project.

Usage

To use the Groovy plugin, include the following in your build script:

Example 620. Using the Groovy plugin

build.gradle

```
plugins {
    id 'groovy'
}
```

build.gradle.kts

```
plugins {
    groovy
}
```

Tasks

The Groovy plugin adds the following tasks to the project. Information about altering the dependencies to Java compile tasks are found here.

`compileGroovy` — **GroovyCompile**

Depends on: `compileJava`

Compiles production Groovy source files.

`compileTestGroovy` — **GroovyCompile**

Depends on: `compileTestJava`

Compiles test Groovy source files.

`compileSourceSetGroovy` — **GroovyCompile**

Depends on: `compileSourceSetJava`

Compiles the given source set's Groovy source files.

`groovydoc` — **Groovydoc**

Generates API documentation for the production Groovy source files.

The Groovy plugin adds the following dependencies to tasks added by the Java plugin.

Table 31. Groovy plugin - additional task dependencies

Task name	Depends on
`classes`	`compileGroovy`
`testClasses`	`compileTestGroovy`

Task name	Depends on
`sourceSetClasses`	`compileSourceSetGroovy`

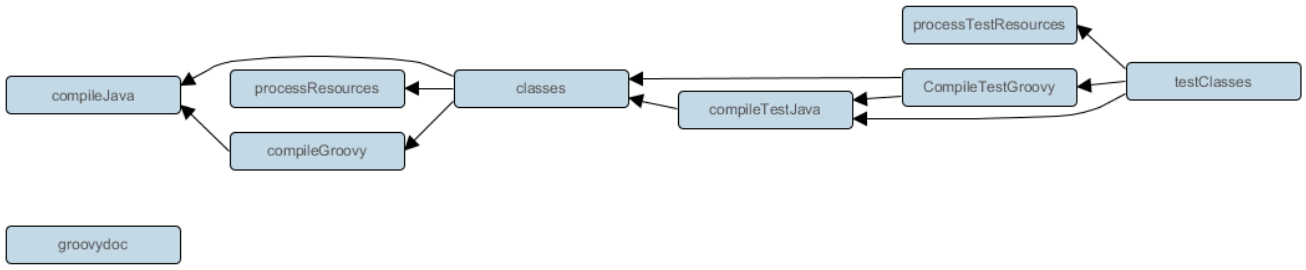


Figure 33. Groovy plugin - tasks

Project layout

The Groovy plugin assumes the project layout shown in Groovy Layout. All the Groovy source directories can contain Groovy *and* Java code. The Java source directories may only contain Java source code.[17] None of these directories need to exist or have anything in them; the Groovy plugin will simply compile whatever it finds.

`src/main/java`

Production Java source.

`src/main/resources`

Production resources, such as XML and properties files.

`src/main/groovy`

Production Groovy source. May also contain Java source files for joint compilation.

`src/test/java`

Test Java source.

`src/test/resources`

Test resources.

`src/test/groovy`

Test Groovy source. May also contain Java source files for joint compilation.

`src/sourceSet/java`

Java source for the source set named *sourceSet*.

`src/sourceSet/resources`

Resources for the source set named *sourceSet*.

`src/sourceSet/groovy`

Groovy source files for the given source set. May also contain Java source files for joint compilation.

Changing the project layout

Just like the Java plugin, the Groovy plugin allows you to configure custom locations for Groovy production and test source files.

Example 621. Custom Groovy source layout

build.gradle

```
sourceSets {
    main {
        groovy {
            srcDirs = ['src/groovy']
        }
    }

    test {
        groovy {
            srcDirs = ['test/groovy']
        }
    }
}
```

build.gradle.kts

```
sourceSets {
    main {
        groovy {
            setSrcDirs(listOf("src/groovy"))
        }
    }

    test {
        groovy {
            setSrcDirs(listOf("test/groovy"))
        }
    }
}
```

Dependency management

Because Gradle's build language is based on Groovy, and parts of Gradle are implemented in Groovy, Gradle already ships with a Groovy library. Nevertheless, Groovy projects need to explicitly declare a Groovy dependency. This dependency will then be used on compile and runtime class paths. It will also be used to get hold of the Groovy compiler and Groovydoc tool, respectively.

If Groovy is used for production code, the Groovy dependency should be added to the `implementation` configuration:

Example 622. Configuration of Groovy dependency

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.codehaus.groovy:groovy-all:2.4.15'
}
```

build.gradle.kts

```
repositories {
    mavenCentral()
}

dependencies {
    implementation("org.codehaus.groovy:groovy-all:2.4.15")
}
```

If Groovy is only used for test code, the Groovy dependency should be added to the `testImplementation` configuration:

Example 623. Configuration of Groovy test dependency

build.gradle

```
dependencies {
    testImplementation 'org.codehaus.groovy:groovy-all:2.4.15'
}
```

build.gradle.kts

```
dependencies {
    testImplementation("org.codehaus.groovy:groovy-all:2.4.15")
}
```

To use the Groovy library that ships with Gradle, declare a `localGroovy()` dependency. Note that different Gradle versions ship with different Groovy versions; as such, using `localGroovy()` is less safe then declaring a regular Groovy dependency.

Example 624. Configuration of bundled Groovy dependency

build.gradle

```
dependencies {
    implementation localGroovy()
}
```

build.gradle.kts

```
dependencies {
    implementation(localGroovy())
}
```

The Groovy library doesn't necessarily have to come from a remote repository. It could also come from a local `lib` directory, perhaps checked in to source control:

Example 625. Configuration of Groovy file dependency

build.gradle

```
repositories {
    flatDir { dirs 'lib' }
}

dependencies {
    implementation module('org.codehaus.groovy:groovy:2.4.15') {
        dependency('org.ow2.asm:asm-all:5.0.3')
        dependency('antlr:antlr:2.7.7')
        dependency('commons-cli:commons-cli:1.2')
        module('org.apache.ant:ant:1.9.4') {
            dependencies('org.apache.ant:ant-junit:1.9.4@jar',
                         'org.apache.ant:ant-launcher:1.9.4')
        }
    }
}
```

build.gradle.kts

```
repositories {
    flatDir { dirs("lib") }
}

dependencies {
    implementation(module("org.codehaus.groovy:groovy:2.4.15") {
        dependency("org.ow2.asm:asm-all:5.0.3")
        dependency("antlr:antlr:2.7.7")
        dependency("commons-cli:commons-cli:1.2")
        module("org.apache.ant:ant:1.9.4") {
            dependencies("org.apache.ant:ant-junit:1.9.4@jar",
                         "org.apache.ant:ant-launcher:1.9.4")
        }
    })
}
```

Automatic configuration of groovyClasspath

The `GroovyCompile` and `Groovydoc` tasks consume Groovy code in two ways: on their `classpath`, and on their `groovyClasspath`. The former is used to locate classes referenced by the source code, and will typically contain the Groovy library along with other libraries. The latter is used to load and execute the Groovy compiler and Groovydoc tool, respectively, and should only contain the Groovy library and its dependencies.

Unless a task's `groovyClasspath` is configured explicitly, the Groovy (base) plugin will try to infer it from the task's `classpath`. This is done as follows:

- If a `groovy-all(-indy)` Jar is found on `classpath`, that jar will be added to `groovyClasspath`.
- If a `groovy(-indy)` jar is found on `classpath`, and the project has at least one repository declared, a corresponding `groovy(-indy)` repository dependency will be added to `groovyClasspath`.
- Otherwise, execution of the task will fail with a message saying that `groovyClasspath` could not be inferred.

Note that the "`-indy`" variation of each jar refers to the version with `invokedynamic` support.

Convention properties

The Groovy plugin does not add any convention properties to the project.

Source set properties

The Groovy plugin adds the following extensions to each source set in the project. You can use these properties in your build script as though they were properties of the source set object.

Groovy Plugin — source set properties

`groovy` — **GroovySourceDirectorySet (read-only)**

Default value: Not null

The Groovy source files of this source set. Contains all `.groovy` and `.java` files found in the Groovy source directories, and excludes all other types of files.

`groovy.srcDirs` — `Set<File>`

Default value: `[projectDir/src/name/groovy]`

The source directories containing the Groovy source files of this source set. May also contain Java source files for joint compilation. Can set using anything described in Specifying Multiple Files.

`allGroovy` — **FileTree (read-only)**

Default value: Not null

All Groovy source files of this source set. Contains only the `.groovy` files found in the Groovy source directories.

These properties are provided by a convention object of type GroovySourceSet.

The Groovy plugin also modifies some source set properties:

Groovy Plugin - modified source set properties

Property name	Change
`allJava`	Adds all `.java` files found in the Groovy source directories.
`allSource`	Adds all source files found in the Groovy source directories.

GroovyCompile

The Groovy plugin adds a GroovyCompile task for each source set in the project. The task type shares much with the `JavaCompile` task by extending `AbstractCompile` (see the relevant Java Plugin section). The `GroovyCompile` task supports most configuration options of the official Groovy compiler. The task can also leverage the Java toolchain support.

Table 32. Groovy plugin - GroovyCompile properties

Task Property	Type	Default Value
`classpath`	FileCollection	`sourceSet.compileClasspath`
`source`	FileTree. Can set using anything described in Specifying Multiple Files.	`sourceSet.groovy`
`destinationDirectory`	`File`.	`sourceSet.groovy.destinationDirectory`
`groovyClasspath`	FileCollection	`groovy` configuration if non-empty; Groovy library found on `classpath` otherwise
`javaLauncher`	`Property<JavaLauncher>`, see the toolchain documentation.	None but will be configured if a toolchain is defined on the `java` extension.

Compilation avoidance

Caveat: Groovy compilation avoidance is an incubating feature since Gradle 5.6. There are known inaccuracies so please enable it at your own risk.

To enable the incubating support for Groovy compilation avoidance, add a `enableFeaturePreview` to your settings file:

settings.gradle

```
enableFeaturePreview('GROOVY_COMPILATION_AVOIDANCE')
```

settings.gradle.kts

```
enableFeaturePreview("GROOVY_COMPILATION_AVOIDANCE")
```

If a dependent project has changed in an ABI-compatible way (only its private API has changed), then Groovy compilation tasks will be up-to-date. This means that if project `A` depends on project `B` and a class in `B` is changed in an ABI-compatible way (typically, changing only the body of a method), then Gradle won't recompile `A`.

See Java compile avoidance for a detailed list of the types of changes that do not affect the ABI and are ignored.

However, similar to Java's annotation processing, there are various ways to customize the Groovy compilation process, for which implementation details matter. Some well-known examples are Groovy AST transformations. In these cases, these dependencies must be declared separately in a classpath called `astTransformationClasspath`:

Example 626. Declaring AST transformations

build.gradle

```
configurations { astTransformation }
dependencies {
    astTransformation(project(":ast-transformation"))
}
tasks.withType(GroovyCompile).configureEach {
    astTransformationClasspath.from(configurations.astTransformation)
}
```

build.gradle.kts

```
val astTransformation by configurations.creating
dependencies {
    astTransformation(project(":ast-transformation"))
}
tasks.withType<GroovyCompile>().configureEach {
    astTransformationClasspath.from(astTransformation)
}
```

Incremental Groovy compilation

Since 5.6, Gradle introduces an experimental incremental Groovy compiler. To enable incremental compilation for Groovy, you need:

- Enable Groovy compilation avoidance.
- Explicitly enable incremental Groovy compilation in the build script:

Example 627. Enable incremental Groovy compilation

buildSrc/src/main/groovy/myproject.groovy-conventions.gradle

```
tasks.withType(GroovyCompile).configureEach {
    options.incremental = true
    options.incrementalAfterFailure = true
}
```

buildSrc/src/main/kotlin/myproject.groovy-conventions.gradle.kts

```
tasks.withType<GroovyCompile>().configureEach {
    options.isIncremental = true
    options.incrementalAfterFailure.set(true)
}
```

This gives you the following benefits:

- Incremental builds are much faster.
- If only a small set of Groovy source files are changed, only the affected source files will be recompiled. Classes that don't need to be recompiled remain unchanged in the output directory. For example, if you only change a few Groovy test classes, you don't need to recompile all Groovy test source files — only the changed ones need to be recompiled.

To understand how incremental compilation works, see Incremental Java compilation for a detailed overview. Note that there're several differences from Java incremental compilation:

The Groovy compiler doesn't keep `@Retention` in generated annotation class bytecode (GROOVY-9185), thus all annotations are `RUNTIME`. This means that changes to source-retention annotations won't trigger a full recompilation.

Known issues

Also see Known issues for incremental Java compilation.

- Changes to resources won't trigger a recompilation, this might result in some incorrectness — for example Extension Modules.

Compiling and testing for Java 6 or Java 7

With toolchain support added to `GroovyCompile`, it is possible to compile Groovy code using a different Java version than the one running Gradle. If you also have Java source files, this will also configure `JavaCompile` to use the right Java compiler is used, as can be seen in the Java plugin documentation.

Example: Configure Java 7 build for Groovy

build.gradle

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(7)
    }
}
```

build.gradle.kts

```
java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(7))
    }
}
```

The IDEA Plugin

The IDEA plugin generates files that are used by IntelliJ IDEA, thus making it possible to open the project from IDEA (`File - Open Project`). Both external dependencies (including associated source and Javadoc files) and project dependencies are considered.

NOTE If you simply want to load a Gradle project into IntelliJ IDEA, then use the IDE's import facility. You do not need to apply this plugin to import your project into IDEA, although if you do, the import will take account of any extra IDEA configuration you have that doesn't directly modify the generated files — see the Configuration section for more details.

What exactly the IDEA plugin generates depends on which other plugins are used:

Always

Generates an IDEA module file. Also generates an IDEA project and workspace file if the project is the root project.

Java Plugin

Additionally adds Java configuration to the IDEA module and project files.

One focus of the IDEA plugin is to be open to customization. The plugin provides a standardized set of hooks for adding and removing content from the generated files.

Usage

To use the IDEA plugin, include this in your build script:

Example 628. Using the IDEA plugin

build.gradle

```
plugins {
    id 'idea'
}
```

build.gradle.kts

```
plugins {
    idea
}
```

The IDEA plugin adds a number of tasks to your project. The `idea` task generates an IDEA module file for the project. When the project is the root project, the `idea` task also generates an IDEA project and workspace. The IDEA project includes modules for each of the projects in the Gradle build.

The IDEA plugin also adds an `openIdea` task when the project is the root project. This task generates the IDEA configuration files and opens the result in IDEA. This means you can simply run `./gradlew openIdea` from the root project to generate and open the IDEA project in one convenient step.

The IDEA plugin also adds a `cleanIdea` task to the project. This task deletes the generated files, if present.

Tasks

The IDEA plugin adds the tasks shown below to a project. Notice that the `clean` task does not depend on the `cleanIdeaWorkspace` task. This is because the workspace typically contains a lot of user specific temporary data and it is not desirable to manipulate it outside IDEA.

`idea`

Depends on: `ideaProject`, `ideaModule`, `ideaWorkspace`

Generates all IDEA configuration files

`openIdea`

Depends on: `idea`

Generates all IDEA configuration files and opens the project in IDEA

`cleanIdea` — **Delete**

Depends on: `cleanIdeaProject`, `cleanIdeaModule`

Removes all IDEA configuration files

`cleanIdeaProject` — **Delete**

Removes the IDEA project file

`cleanIdeaModule` — **Delete**

Removes the IDEA module file

`cleanIdeaWorkspace` — **Delete**

Removes the IDEA workspace file

`ideaProject` — **GenerateIdeaProject**

Generates the `.ipr` file. This task is only added to the root project.

`ideaModule` — **GenerateIdeaModule**

Generates the `.iml` file

`ideaWorkspace` — **GenerateIdeaWorkspace**

Generates the `.iws` file. This task is only added to the root project.

Configuration

The plugin adds some configuration options that allow to customize the IDEA project and module files that it generates. These take the form of both model properties and lower-level mechanisms that modify the generated files directly. For example, you can add source and resource directories, as well as inject your own fragments of XML. The former type of configuration is honored by IDEA's import facility, whereas the latter is not.

Here are the configuration properties you can use:

`idea` — **IdeaModel**

Top level element that enables configuration of the idea plugin in a DSL-friendly fashion

`idea.project` **IdeaProject**

Allows configuring project information

`idea.module` **IdeaModule**

Allows configuring module information

`idea.workspace` **IdeaWorkspace**

Allows configuring the workspace XML

Follow the links to the types for examples of using these configuration properties.

Customizing the generated files

The IDEA plugin provides hooks and behavior for customizing the generated content in a more controlled and detailed way. In addition, the `withXml` hook is the only practical way to modify the workspace file because its corresponding domain object is essentially empty.

NOTE	The techniques we discuss in this section don't work with IDEA's import facility

The tasks recognize existing IDEA files and merge them with the generated content.

Merging

Sections of existing IDEA files that are also the target of generated content will be amended or overwritten, depending on the particular section. The remaining sections will be left as-is.

Disabling merging with a complete overwrite

To completely rewrite existing IDEA files, execute a clean task together with its corresponding generation task, like "`gradle cleanIdea idea`" (in that order). If you want to make this the default behavior, add "`tasks.idea.dependsOn(cleanIdea)`" to your build script. This makes it unnecessary to execute the clean task explicitly.

This strategy can also be used for individual files that the plugin would generate. For instance, this can be done for the "`.iml`" file with "`gradle cleanIdeaModule ideaModule`".

Hooking into the generation lifecycle

The plugin provides objects modeling the sections of the metadata files that are generated by Gradle. The generation lifecycle is as follows:

1. The file is read; or a default version provided by Gradle is used if it does not exist
2. The `beforeMerged` hook is executed with a domain object representing the existing file
3. The existing content is merged with the configuration inferred from the Gradle build or defined explicitly in the eclipse DSL
4. The `whenMerged` hook is executed with a domain object representing contents of the file to be persisted
5. The `withXml` hook is executed with a raw representation of the XML that will be persisted
6. The final XML is persisted

The following are the domain objects used for each of the model types:

IdeaProject

- `beforeMerged { Project arg -> ⋯ }`
- `whenMerged { Project arg -> ⋯ }`
- `withXml { XmlProvider arg -> ⋯ }`

IdeaModule

- `beforeMerged { Module arg -> … }`
- `whenMerged { Module arg -> … }`
- `withXml { XmlProvider arg -> … }`

IdeaWorkspace

- `beforeMerged { Workspace arg -> … }`
- `whenMerged { Workspace arg -> … }`
- `withXml { XmlProvider arg -> … }`

Partial rewrite of existing content

A "complete rewrite" causes all existing content to be discarded, thereby losing any changes made directly in the IDE. The `beforeMerged` hook makes it possible to overwrite just certain parts of the existing content. The following example removes all existing dependencies from the `Module` domain object:

Example 629. Partial Rewrite for Module

build.gradle

```
idea.module.iml {
    beforeMerged { module ->
        module.dependencies.clear()
    }
}
```

build.gradle.kts

```
import org.gradle.plugins.ide.idea.model.Module

idea.module.iml {
    beforeMerged(Action<Module> {
        dependencies.clear()
    })
}
```

The resulting module file will only contain Gradle-generated dependency entries, but not any other dependency entries that may have been present in the original file. (In the case of dependency entries, this is also the default behavior.) Other sections of the module file will be either left as-is or merged. The same could be done for the module paths in the project file:

Example 630. Partial Rewrite for Project

build.gradle

```
idea.project.ipr {
    beforeMerged { project ->
        project.modulePaths.clear()
    }
}
```

build.gradle.kts

```
import org.gradle.plugins.ide.idea.model.Project

idea.project.ipr {
    beforeMerged(Action<Project> {
        modulePaths.clear()
    })
}
```

Modifying the fully populated domain objects

The `whenMerged` hook allows you to manipulate the fully populated domain objects. Often this is the preferred way to customize IDEA files. Here is how you would export all the dependencies of an IDEA module:

Example 631. Export Dependencies

build.gradle

```
idea.module.iml {
    whenMerged { module ->
        module.dependencies*.exported = true
    }
}
```

build.gradle.kts

```
import org.gradle.plugins.ide.idea.model.Module
import org.gradle.plugins.ide.idea.model.ModuleDependency

idea.module.iml {
    whenMerged(Action<Module> {
        dependencies.forEach {
            (it as ModuleDependency).isExported = true
        }
    })
}
```

Modifying the XML representation

The `withXml` hook allows you to manipulate the in-memory XML representation just before the file gets written to disk. Although Groovy's XML support and Kotlin's extension functions make up for a lot, this approach is less convenient than manipulating the domain objects. In return, you get total control over the generated file, including sections not modeled by the domain objects.

Example 632. Customizing the XML

build.gradle

```
idea.project.ipr {
    withXml { provider ->
        provider.node.component
                .find { it.@name == 'VcsDirectoryMappings' }
                .mapping.@vcs = 'Git'
    }
}
```

build.gradle.kts

```
import org.w3c.dom.Element

idea.project.ipr {
    withXml(Action<XmlProvider> {
        fun Element.firstElement(predicate: (Element.() -> Boolean)) =
            childNodes
                .run { (0 until length).map(::item) }
                .filterIsInstance<Element>()
                .first { it.predicate() }

        asElement()
            .firstElement { tagName == "component" && getAttribute("name") ==
"VcsDirectoryMappings" }
            .firstElement { tagName == "mapping" }
            .setAttribute("vcs", "Git")
    })
}
```

Identifying additional test directories

When using this plugin together with the Java plugin, after adding additional source sets you may wish to inform IDEA when they contain test source rather than production source, so that the IDE can treat the directories appropriately. This can be accomplished by using this plugin's `Module` block.

Example 633. Identifying Additional Test Directories in SourceSets

build.gradle

```
sourceSets {
    intTest {
        java {
            srcDirs = ['src/integration']
        }
    }
}

idea {
    module {
        testSources.from(sourceSets.intTest.java.srcDirs)
    }
}
```

build.gradle.kts

```
sourceSets {
    create("intTest") {
        java {
            setSrcDirs(listOf("src/integration"))
        }
    }
}

idea {
    module {
        testSources.from(sourceSets["intTest"].java.srcDirs)
    }
}
```

NOTE When working with the JVM Test Suite Plugin, test sources will automatically identified correctly.

Further things to consider

The paths of dependencies in the generated IDEA files are absolute. If you manually define a path variable pointing to the Gradle dependency cache, IDEA will automatically replace the absolute dependency paths with this path variable. you can configure this path variable via the "`idea.pathVariables`" property, so that it can do a proper merge without creating duplicates.

Ivy Publish Plugin

The Ivy Publish Plugin provides the ability to publish build artifacts in the Apache Ivy format, usually to a repository for consumption by other builds or projects. What is published is one or more artifacts created by the build, and an Ivy *module descriptor* (normally `ivy.xml`) that describes the artifacts and the dependencies of the artifacts, if any.

A published Ivy module can be consumed by Gradle (see Declaring Dependencies) and other tools that understand the Ivy format. You can learn about the fundamentals of publishing in Publishing Overview.

Usage

To use the Ivy Publish Plugin, include the following in your build script:

Example 634. Applying the Ivy Publish Plugin

build.gradle

```
plugins {
    id 'ivy-publish'
}
```

build.gradle.kts

```
plugins {
    `ivy-publish`
}
```

The Ivy Publish Plugin uses an extension on the project named `publishing` of type PublishingExtension. This extension provides a container of named publications and a container of named repositories. The Ivy Publish Plugin works with IvyPublication publications and IvyArtifactRepository repositories.

Tasks

`generateDescriptorFileFor`**`PubName`**`Publication` — **GenerateIvyDescriptor**

Creates an Ivy descriptor file for the publication named *PubName*, populating the known metadata such as project name, project version, and the dependencies. The default location for the descriptor file is *build/publications/$pubName/ivy.xml*.

`publish`**`PubName`**`PublicationTo`**`RepoName`**`Repository` — **PublishToIvyRepository**

Publishes the *PubName* publication to the repository named *RepoName*. If you have a repository definition without an explicit name, *RepoName* will be "Ivy".

`publish`

Depends on: All `publishPubNamePublicationToRepoNameRepository` tasks

An aggregate task that publishes all defined publications to all defined repositories.

Publications

This plugin provides publications of type IvyPublication. To learn how to define and use publications, see the section on basic publishing.

There are four main things you can configure in an Ivy publication:

- A component — via IvyPublication.from(org.gradle.api.component.SoftwareComponent).
- Custom artifacts — via the IvyPublication.artifact(java.lang.Object) method. See IvyArtifact for the available configuration options for custom Ivy artifacts.
- Standard metadata like `module`, `organisation` and `revision`.
- Other contents of the module descriptor — via IvyPublication.descriptor(org.gradle.api.Action).

You can see all of these in action in the complete publishing example. The API documentation for `IvyPublication` has additional code samples.

Identity values for the published project

The generated Ivy module descriptor file contains an `<info>` element that identifies the module. The default identity values are derived from the following:

- `organisation` - Project.getGroup()
- `module` - Project.getName()
- `revision` - Project.getVersion()
- `status` - Project.getStatus()
- `branch` - (not set)

Overriding the default identity values is easy: simply specify the `organisation`, `module` or `revision` properties when configuring the IvyPublication. `status` and `branch` can be set via the `descriptor` property — see IvyModuleDescriptorSpec.

The `descriptor` property can also be used to add additional custom elements as children of the `<info>` element, like so:

Example 635. customizing the publication identity

build.gradle

```
publishing {
    publications {
        ivy(IvyPublication) {
            organisation = 'org.gradle.sample'
            module = 'project1-sample'
            revision = '1.1'
            descriptor.status = 'milestone'
            descriptor.branch = 'testing'
            descriptor.extraInfo 'http://my.namespace', 'myElement', 'Some
value'

            from components.java
        }
    }
}
```

build.gradle.kts

```
publishing {
    publications {
        create<IvyPublication>("ivy") {
            organisation = "org.gradle.sample"
            module = "project1-sample"
            revision = "1.1"
            descriptor.status = "milestone"
            descriptor.branch = "testing"
            descriptor.extraInfo("http://my.namespace", "myElement", "Some
value")

            from(components["java"])
        }
    }
}
```

TIP Certain repositories are not able to handle all supported characters. For example, the `:` character cannot be used as an identifier when publishing to a filesystem-backed repository on Windows.

Gradle will handle any valid Unicode character for `organisation`, `module` and `revision` (as well as the artifact's `name`, `extension` and `classifier`). The only values that are explicitly prohibited are `\`, `/` and any ISO control character. The supplied values are validated early during publication.

Customizing the generated module descriptor

At times, the module descriptor file generated from the project information will need to be tweaked before publishing. The Ivy Publish Plugin provides a DSL for that purpose. Please see IvyModuleDescriptorSpec in the DSL Reference for the complete documentation of available properties and methods.

The following sample shows how to use the most common aspects of the DSL:

Example 636. Customizing the module descriptor file

build.gradle

```
publications {
    ivyCustom(IvyPublication) {
        descriptor {
            license {
                name = 'The Apache License, Version 2.0'
                url = 'http://www.apache.org/licenses/LICENSE-2.0.txt'
            }
            author {
                name = 'Jane Doe'
                url = 'http://example.com/users/jane'
            }
            description {
                text = 'A concise description of my library'
                homepage = 'http://www.example.com/library'
            }
        }
        versionMapping {
            usage('java-api') {
                fromResolutionOf('runtimeClasspath')
            }
            usage('java-runtime') {
                fromResolutionResult()
            }
        }
    }
}
```

build.gradle.kts

```
publications {
    create<IvyPublication>("ivyCustom") {
        descriptor {
            license {
                name.set("The Apache License, Version 2.0")
                url.set("http://www.apache.org/licenses/LICENSE-2.0.txt")
            }
            author {
                name.set("Jane Doe")
                url.set("http://example.com/users/jane")
            }
            description {
                text.set("A concise description of my library")
                homepage.set("http://www.example.com/library")
            }
        }
        versionMapping {
            usage("java-api") {
                fromResolutionOf("runtimeClasspath")
            }
            usage("java-runtime") {
                fromResolutionResult()
            }
        }
    }
}
```

In this example we are simply adding a 'description' element to the generated Ivy dependency descriptor, but this hook allows you to modify any aspect of the generated descriptor. For example, you could replace the version range for a dependency with the actual version used to produce the build.

You can also add arbitrary XML to the descriptor file via IvyModuleDescriptorSpec.withXml(org.gradle.api.Action), but you cannot use it to modify any part of the module identifier (organisation, module, revision).

CAUTION	It is possible to modify the descriptor in such a way that it is no longer a valid Ivy module descriptor, so care must be taken when using this feature.

Customizing dependencies versions

Two strategies are supported for publishing dependencies:

Declared versions (default)

This strategy publishes the versions that are defined by the build script author with the

dependency declarations in the `dependencies` block. Any other kind of processing, for example through a rule changing the resolved version, will not be taken into account for the publication.

Resolved versions

This strategy publishes the versions that were resolved during the build, possibly by applying resolution rules and automatic conflict resolution. This has the advantage that the published versions correspond to the ones the published artifact was tested against.

Example use cases for resolved versions:

- A project uses dynamic versions for dependencies but prefers exposing the resolved version for a given release to its consumers.
- In combination with dependency locking, you want to publish the locked versions.
- A project leverages the rich versions constraints of Gradle, which have a lossy conversion to Ivy. Instead of relying on the conversion, it publishes the resolved versions.

This is done by using the `versionMapping` DSL method which allows to configure the VersionMappingStrategy:

Example 637. Using resolved versions

build.gradle

```
publications {
    ivyCustom(IvyPublication) {
        versionMapping {
            usage('java-api') {
                fromResolutionOf('runtimeClasspath')
            }
            usage('java-runtime') {
                fromResolutionResult()
            }
        }
    }
}
```

build.gradle.kts

```
publications {
    create<IvyPublication>("ivyCustom") {
        versionMapping {
            usage("java-api") {
                fromResolutionOf("runtimeClasspath")
            }
            usage("java-runtime") {
                fromResolutionResult()
            }
        }
    }
}
```

In the example above, Gradle will use the versions resolved on the `runtimeClasspath` for dependencies declared in `api`, which are mapped to the `compile` configuration of Ivy. Gradle will also use the versions resolved on the `runtimeClasspath` for dependencies declared in `implementation`, which are mapped to the `runtime` configuration of Ivy. `fromResolutionResult()` indicates that Gradle should use the default classpath of a variant and `runtimeClasspath` is the default classpath of `java-runtime`.

Repositories

This plugin provides repositories of type IvyArtifactRepository. To learn how to define and use repositories for publishing, see the section on basic publishing.

Here's a simple example of defining a publishing repository:

Example 638. Declaring repositories to publish to

build.gradle

```
publishing {
    repositories {
        ivy {
            // change to point to your repo, e.g. http://my.org/repo
            url = layout.buildDirectory.dir("repo")
        }
    }
}
```

build.gradle.kts

```
publishing {
    repositories {
        ivy {
            // change to point to your repo, e.g. http://my.org/repo
            url = uri(layout.buildDirectory.dir("repo"))
        }
    }
}
```

The two main things you will want to configure are the repository's:

- URL (required)
- Name (optional)

You can define multiple repositories as long as they have unique names within the build script. You may also declare one (and only one) repository without a name. That repository will take on an implicit name of "Ivy".

You can also configure any authentication details that are required to connect to the repository. See IvyArtifactRepository for more details.

Complete example

The following example demonstrates publishing with a multi-project build. Each project publishes a Java component configured to also build and publish Javadoc and source code artifacts. The descriptor file is customized to include the project description for each project.

Example 639. Publishing a Java module

settings.gradle

```
rootProject.name = 'ivy-publish-java'
include 'project1', 'project2'
```

buildSrc/build.gradle

```
plugins {
    id 'groovy-gradle-plugin'
}
```

buildSrc/src/main/groovy/myproject.publishing-conventions.gradle

```
plugins {
    id 'java-library'
    id 'ivy-publish'
}

version = '1.0'
group = 'org.gradle.sample'

repositories {
    mavenCentral()
}

java {
    withJavadocJar()
    withSourcesJar()
}

publishing {
    repositories {
        ivy {
            // change to point to your repo, e.g. http://my.org/repo
            url = "${rootProject.buildDir}/repo"
        }
    }
    publications {
        ivy(IvyPublication) {
            from components.java
            descriptor.description {
                text = providers.provider({ description })
            }
        }
    }
}
```

project1/build.gradle

```
plugins {
    id 'myproject.publishing-conventions'
}

description = 'The first project'

dependencies {
    implementation 'junit:junit:4.13'
    implementation project(':project2')
}
```

project2/build.gradle

```
plugins {
    id 'myproject.publishing-conventions'
}

description = 'The second project'

dependencies {
    implementation 'commons-collections:commons-collections:3.2.2'
}
```

settings.gradle.kts

```
rootProject.name = "ivy-publish-java"
include("project1", "project2")
```

buildSrc/build.gradle.kts

```
plugins {
    `kotlin-dsl`
}

repositories {
    gradlePluginPortal()
}
```

buildSrc/src/main/kotlin/myproject.publishing-conventions.gradle.kts

```
plugins {
    id("java-library")
    id("ivy-publish")
}

version = "1.0"
group = "org.gradle.sample"

repositories {
    mavenCentral()
}

java {
    withJavadocJar()
    withSourcesJar()
}

publishing {
    repositories {
        ivy {
            // change to point to your repo, e.g. http://my.org/repo
            url = uri("${rootProject.buildDir}/repo")
        }
    }
    publications {
        create<IvyPublication>("ivy") {
            from(components["java"])
            descriptor.description {
                text.set(providers.provider({ description }))
            }
        }
    }
}
```

project1/build.gradle.kts

```
plugins {
    id("myproject.publishing-conventions")
}

description = "The first project"

dependencies {
    implementation("junit:junit:4.13")
    implementation(project(":project2"))
}
```

project2/build.gradle.kts

```
plugins {
    id("myproject.publishing-conventions")
}

description = "The second project"

dependencies {
    implementation("commons-collections:commons-collections:3.2.2")
}
```

The result is that the following artifacts will be published for each project:

- The Gradle Module Metadata file: `project1-1.0.module`.
- The Ivy module metadata file: `ivy-1.0.xml`.
- The primary JAR artifact for the Java component: `project1-1.0.jar`.
- The Javadoc and sources JAR artifacts of the Java component (because we configured `withJavadocJar()` and `withSourcesJar()`): `project1-1.0-javadoc.jar`, `project1-1.0-source.jar`.

The JaCoCo Plugin

The JaCoCo plugin provides code coverage metrics for Java code via integration with JaCoCo.

Getting Started

To get started, apply the JaCoCo plugin to the project you want to calculate code coverage for.

Example 640. Applying the JaCoCo plugin

build.gradle

```
plugins {
    id 'jacoco'
}
```

build.gradle.kts

```
plugins {
    jacoco
}
```

If the Java plugin is also applied to your project, a new task named `jacocoTestReport` is created. By default, a HTML report is generated at `$buildDir/reports/jacoco/test`.

NOTE	While tests should be executed before generation of the report, the `jacocoTestReport` task does not depend on the `test` task.

Depending on your usecases, you may want to always generate the `jacocoTestReport` or run the `test` task before generating the report explicitly.

Example 641. Define dependencies between code coverage reports and test execution

build.gradle

```
test {
    finalizedBy jacocoTestReport // report is always generated after tests
run
}
jacocoTestReport {
    dependsOn test // tests are required to run before generating the report
}
```

build.gradle.kts

```
tasks.test {
    finalizedBy(tasks.jacocoTestReport) // report is always generated after
tests run
}
tasks.jacocoTestReport {
    dependsOn(tasks.test) // tests are required to run before generating the
report
}
```

Configuring the JaCoCo Plugin

The JaCoCo plugin adds a project extension named `jacoco` of type JacocoPluginExtension, which allows configuring defaults for JaCoCo usage in your build.

Example 642. Configuring JaCoCo plugin settings

build.gradle

```
jacoco {
    toolVersion = "0.8.8"
    reportsDirectory = layout.buildDirectory.dir('customJacocoReportDir')
}
```

build.gradle.kts

```
jacoco {
    toolVersion = "0.8.8"
    reportsDirectory.set(layout.buildDirectory.dir("customJacocoReportDir"))
}
```

Table 33. Gradle defaults for JaCoCo properties

Property	Gradle default
reportsDirectory	`$buildDir/reports/jacoco`

JaCoCo Report configuration

The JacocoReport task can be used to generate code coverage reports in different formats. It implements the standard Gradle type Reporting and exposes a report container of type JacocoReportsContainer.

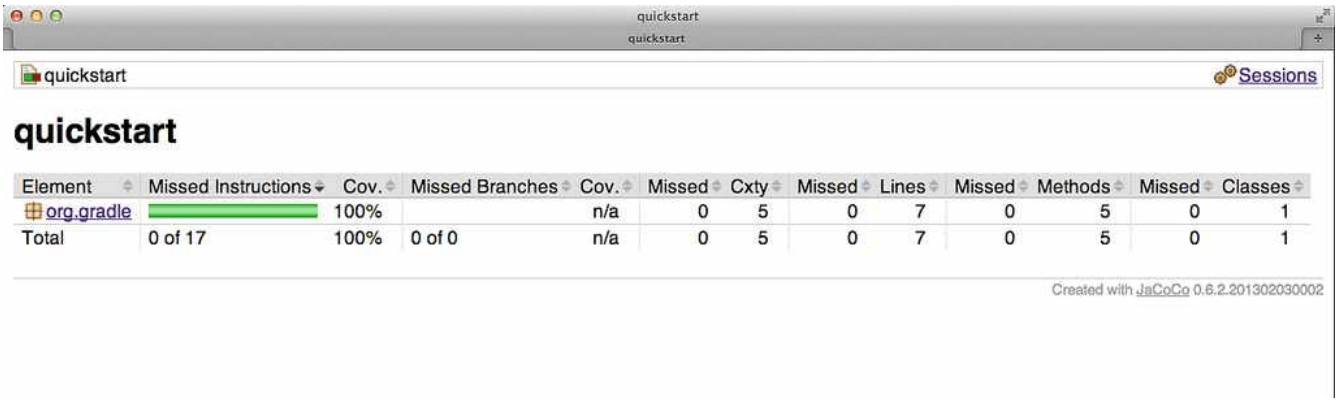
Example 643. Configuring test task

build.gradle

```
jacocoTestReport {
    reports {
        xml.required = false
        csv.required = false
        html.outputLocation = layout.buildDirectory.dir('jacocoHtml')
    }
}
```

build.gradle.kts

```
tasks.jacocoTestReport {
    reports {
        xml.required.set(false)
        csv.required.set(false)
        html.outputLocation.set(layout.buildDirectory.dir("jacocoHtml"))
    }
}
```



Enforcing code coverage metrics

NOTE	This feature requires the use of JaCoCo version 0.6.3 or higher.

The JacocoCoverageVerification task can be used to verify if code coverage metrics are met based on configured rules. Its API exposes the method JacocoCoverageVerification.violationRules(org.gradle.api.Action) which is used as main entry point for configuring rules. Invoking any of those methods returns an instance of JacocoViolationRulesContainer providing extensive configuration options. The build fails if any of the configured rules are not met. JaCoCo only reports the first violated rule.

Code coverage requirements can be specified for a project as a whole, for individual files, and for

particular JaCoCo-specific types of coverage, e.g., lines covered or branches covered. The following example describes the syntax.

Example 644. Configuring violation rules

build.gradle

```
jacocoTestCoverageVerification {
    violationRules {
        rule {
            limit {
                minimum = 0.5
            }
        }

        rule {
            enabled = false
            element = 'CLASS'
            includes = ['org.gradle.*']

            limit {
                counter = 'LINE'
                value = 'TOTALCOUNT'
                maximum = 0.3
            }
        }
    }
}
```

build.gradle.kts

```
tasks.jacocoTestCoverageVerification {
    violationRules {
        rule {
            limit {
                minimum = "0.5".toBigDecimal()
            }
        }

        rule {
            isEnabled = false
            element = "CLASS"
            includes = listOf("org.gradle.*")

            limit {
                counter = "LINE"
                value = "TOTALCOUNT"
                maximum = "0.3".toBigDecimal()
            }
        }
    }
}
```

The JacocoCoverageVerification task is not a task dependency of the `check` task provided by the Java plugin. There is a good reason for it. The task is currently not incremental as it doesn't declare any outputs. Any violation of the declared rules would automatically result in a failed build when executing the `check` task. This behavior might not be desirable for all users. Future versions of Gradle might change the behavior.

JaCoCo specific task configuration

The JaCoCo plugin adds a JacocoTaskExtension extension to all tasks of type Test. This extension allows the configuration of the JaCoCo specific properties of the test task.

Example 645. Configuring test task

build.gradle

```
test {
    jacoco {
        destinationFile = layout.buildDirectory.file('jacoco/jacocoTest.exec
').get().asFile
        classDumpDir = layout.buildDirectory.dir('jacoco/classpathdumps').
get().asFile
    }
}
```

build.gradle.kts

```
tasks.test {
    extensions.configure(JacocoTaskExtension::class) {
        destinationFile =
layout.buildDirectory.file("jacoco/jacocoTest.exec").get().asFile
        classDumpDir =
layout.buildDirectory.dir("jacoco/classpathdumps").get().asFile
    }
}
```

NOTE Tasks configured for running with the JaCoCo agent delete the destination file for the execution data when the task starts executing. This ensures that no stale coverage data is present in the execution data.

Default values of the JaCoCo Task extension

Example 646. JaCoCo task extension default values

build.gradle

```
test {
    jacoco {
        enabled = true
        destinationFile = layout.buildDirectory.file("jacoco/${name}.exec")
.get().asFile
        includes = []
        excludes = []
        excludeClassLoaders = []
        includeNoLocationClasses = false
        sessionId = "<auto-generated value>"
        dumpOnExit = true
        classDumpDir = null
        output = JacocoTaskExtension.Output.FILE
        address = "localhost"
        port = 6300
        jmx = false
    }
}
```

build.gradle.kts

```
tasks.test {
    configure<JacocoTaskExtension> {
        isEnabled = true
        destinationFile =
layout.buildDirectory.file("jacoco/${name}.exec").get().asFile
        includes = emptyList()
        excludes = emptyList()
        excludeClassLoaders = emptyList()
        isIncludeNoLocationClasses = false
        sessionId = "<auto-generated value>"
        isDumpOnExit = true
        classDumpDir = null
        output = JacocoTaskExtension.Output.FILE
        address = "localhost"
        port = 6300
        isJmx = false
    }
}
```

While all tasks of type Test are automatically enhanced to provide coverage information when the

`java` plugin has been applied, any task that implements JavaForkOptions can be enhanced by the JaCoCo plugin. That is, any task that forks Java processes can be used to generate coverage information.

For example you can configure your build to generate code coverage using the `application` plugin.

Example 647. Using application plugin to generate code coverage data

build.gradle

```
plugins {
    id 'application'
    id 'jacoco'
}

application {
    mainClass = 'org.gradle.MyMain'
}

jacoco {
    applyTo run
}

tasks.register('applicationCodeCoverageReport', JacocoReport) {
    executionData run
    sourceSets sourceSets.main
}
```

build.gradle.kts

```
plugins {
    application
    jacoco
}

application {
    mainClass.set("org.gradle.MyMain")
}

jacoco {
    applyTo(tasks.run.get())
}

tasks.register<JacocoReport>("applicationCodeCoverageReport") {
    executionData(tasks.run.get())
    sourceSets(sourceSets.main.get())
}
```

Coverage reports generated by applicationCodeCoverageReport

```
.
└─── build
    ├─── jacoco
    │    └─── run.exec
    └─── reports
        └─── jacoco
            └─── applicationCodeCoverageReport
                └─── html
                    └─── index.html
```

Tasks

For projects that also apply the Java Plugin, the JaCoCo plugin automatically adds the following tasks:

`jacocoTestReport` — **JacocoReport**

Generates code coverage report for the test task.

`jacocoTestCoverageVerification` — **JacocoCoverageVerification**

Verifies code coverage metrics based on specified rules for the test task.

Dependency management

The JaCoCo plugin adds the following dependency configurations:

Table 34. JaCoCo plugin - dependency configurations

Name	Meaning
`jacocoAnt`	The JaCoCo Ant library used for running the `JacocoReport`, `JacocoMerge` and `JacocoCoverageVerification` tasks.
`jacocoAgent`	The JaCoCo agent library used for instrumenting the code under test.

Outgoing Variants

When a project producing JaCoCo coverage data is applied alongside the JVM Test Suite Plugin, additional outgoing variants will be created. These variants are designed for consumption by the JaCoCo Report Aggregation Plugin.

The attributes will resemble the following. User-configurable attributes are highlighted below the sample.

outgoingVariants task output

```
--------------------------------------------------
Variant coverageDataElementsForTest (i)
--------------------------------------------------
Description = Binary data file containing results of Jacoco test coverage reporting
for the test Test Suite's test target.

Capabilities
    - org.gradle.sample:application:1.0.2 (default capability)
Attributes
    - org.gradle.category              = verification
    - org.gradle.testsuite.name        = test           ①
    - org.gradle.testsuite.target.name = test           ②
    - org.gradle.testsuite.type        = unit-test      ③
    - org.gradle.verificationtype      = jacoco-coverage

Artifacts
    - build/jacoco/test.exec (artifactType = binary)
```

① TestSuiteName attribute; value is derived from JvmTestSuite#getName().

② TestSuiteTargetName attribute; value is derived from JvmTestSuiteTarget#getName().

③ TestSuiteType attribute; value is derived from JvmTestSuite#getTestType().

The Java Plugin

The Java plugin adds Java compilation along with testing and bundling capabilities to a project. It serves as the basis for many of the other JVM language Gradle plugins. You can find a comprehensive introduction and overview to the Java Plugin in the Building Java Projects chapter.

NOTE	As indicated above, this plugin adds basic building blocks for working with JVM projects. Its feature set has been superseded by other plugins, offering more features based on your project type. Instead of applying it directly to your project, you should look into the `java-library` or `application` plugins or one of the supported alternative JVM language.

Usage

To use the Java plugin, include the following in your build script:

Example 648. Using the Java plugin

build.gradle

```
plugins {
    id 'java'
}
```

build.gradle.kts

```
plugins {
    java
}
```

Tasks

The Java plugin adds a number of tasks to your project, as shown below.

`compileJava` — **JavaCompile**

Depends on: All tasks which contribute to the compilation classpath, including `jar` tasks from projects that are on the classpath via project dependencies

Compiles production Java source files using the JDK compiler.

`processResources` — **ProcessResources**

Copies production resources into the production resources directory.

`classes`

Depends on: `compileJava`, `processResources`

This is an aggregate task that just depends on other tasks. Other plugins may attach additional compilation tasks to it.

`compileTestJava` — **JavaCompile**

Depends on: `classes`, and all tasks that contribute to the test compilation classpath

Compiles test Java source files using the JDK compiler.

`processTestResources` — **Copy**

Copies test resources into the test resources directory.

`testClasses`

Depends on: `compileTestJava`, `processTestResources`

This is an aggregate task that just depends on other tasks. Other plugins may attach additional

test compilation tasks to it.

`jar` — **Jar**

Depends on: `classes`

Assembles the production JAR file, based on the classes and resources attached to the `main` source set.

`javadoc` — **Javadoc**

Depends on: `classes`

Generates API documentation for the production Java source using Javadoc.

`test` — **Test**

Depends on: `testClasses`, and all tasks which produce the test runtime classpath

Runs the unit tests using JUnit or TestNG.

`clean` — **Delete**

Deletes the project build directory.

`cleanTaskName` — **Delete**

Deletes files created by the specified task. For example, `cleanJar` will delete the JAR file created by the `jar` task and `cleanTest` will delete the test results created by the `test` task.

SourceSet Tasks

For each source set you add to the project, the Java plugin adds the following tasks:

`compileSourceSetJava` — **JavaCompile**

Depends on: All tasks which contribute to the source set's compilation classpath

Compiles the given source set's Java source files using the JDK compiler.

`processSourceSetResources` — **Copy**

Copies the given source set's resources into the resources directory.

`sourceSetClasses` — **Task**

Depends on: `compileSourceSetJava`, `processSourceSetResources`

Prepares the given source set's classes and resources for packaging and execution. Some plugins may add additional compilation tasks for the source set.

Lifecycle Tasks

The Java plugin attaches some of its tasks to the lifecycle tasks defined by the Base Plugin — which the Java Plugin applies automatically — and it also adds a few other lifecycle tasks:

`assemble`

Depends on: `jar`, and all other tasks that create artifacts attached to the `archives` configuration

Aggregate task that assembles all the archives in the project. This task is added by the Base Plugin.

`check`

Depends on: `test`

Aggregate task that performs verification tasks, such as running the tests. Some plugins add their own verification tasks to `check`. You should also attach any custom `Test` tasks to this lifecycle task if you want them to execute for a full build. This task is added by the Base Plugin.

`build`

Depends on: `check`, `assemble`

Aggregate tasks that performs a full build of the project. This task is added by the Base Plugin.

`buildNeeded`

Depends on: `build`, and `buildNeeded` tasks in all projects that are dependencies in the `testRuntimeClasspath` configuration.

Performs a full build of the project and all projects it depends on.

`buildDependents`

Depends on: `build`, and `buildDependents` tasks in all projects that have this project as a dependency in their `testRuntimeClasspath` configurations

Performs a full build of the project and all projects which depend upon it.

`buildConfigName` — ***task rule***

Depends on: all tasks that generate the artifacts attached to the named — *ConfigName* — configuration

Assembles the artifacts for the specified configuration. This rule is added by the Base Plugin.

`uploadConfigName` — ***task rule***, **type: Upload**

Depends on: all tasks that generate the artifacts attached to the named — *ConfigName* — configuration

Assembles and uploads the artifacts in the specified configuration. This rule is added by the Base Plugin.

The following diagram shows the relationships between these tasks.

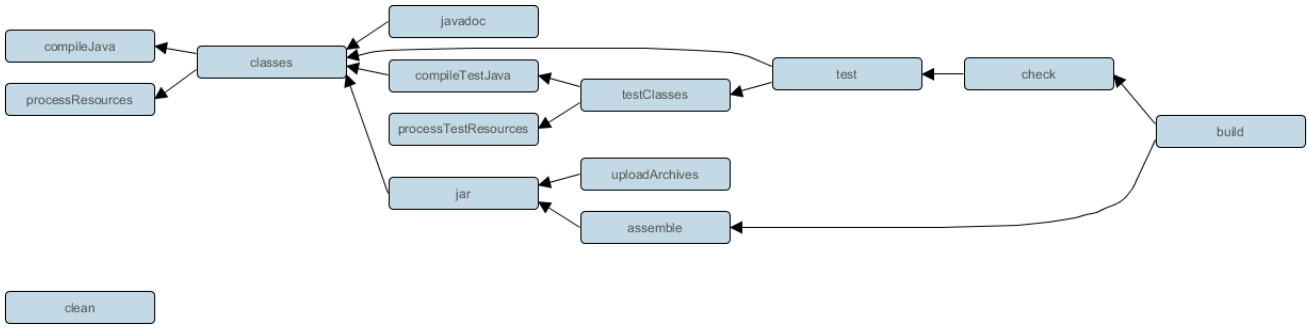


Figure 34. Java plugin - tasks

Project layout

The Java plugin assumes the project layout shown below. None of these directories need to exist or have anything in them. The Java plugin will compile whatever it finds, and handles anything which is missing.

`src/main/java`
: Production Java source.

`src/main/resources`
: Production resources, such as XML and properties files.

`src/test/java`
: Test Java source.

`src/test/resources`
: Test resources.

`src/sourceSet/java`
: Java source for the source set named *sourceSet*.

`src/sourceSet/resources`
: Resources for the source set named *sourceSet*.

Changing the project layout

You configure the project layout by configuring the appropriate source set. This is discussed in more detail in the following sections. Here is a brief example which changes the main Java and resource source directories.

Example 649. Custom Java source layout

build.gradle

```
sourceSets {
    main {
        java {
            srcDirs = ['src/java']
        }
        resources {
            srcDirs = ['src/resources']
        }
    }
}
```

build.gradle.kts

```
sourceSets {
    main {
        java {
            setSrcDirs(listOf("src/java"))
        }
        resources {
            setSrcDirs(listOf("src/resources"))
        }
    }
}
```

Source sets

The plugin adds the following source sets:

`main`

Contains the production source code of the project, which is compiled and assembled into a JAR.

`test`

Contains your test source code, which is compiled and executed using JUnit or TestNG. These are typically unit tests, but you can include any test in this source set as long as they all share the same compilation and runtime classpaths.

Source set properties

The following table lists some of the important properties of a source set. You can find more details in the API documentation for SourceSet.

`name` — **(read-only)** `String`

The name of the source set, used to identify it.

`output` — **(read-only)** **SourceSetOutput**

The output files of the source set, containing its compiled classes and resources.

`output.classesDirs` — **(read-only)** **FileCollection**

Default value: `$buildDir/classes/java/$name`, e.g. *build/classes/java/main*

The directories to generate the classes of this source set into. May contain directories for other JVM languages, e.g. *build/classes/kotlin/main*.

`output.resourcesDir` — `File`

Default value: `$buildDir/resources/$name`, e.g. *build/resources/main*

The directory to generate the resources of this source set into.

`compileClasspath` — **FileCollection**

Default value: `${name}CompileClasspath` configuration

The classpath to use when compiling the source files of this source set.

`annotationProcessorPath` — **FileCollection**

Default value: `${name}AnnotationProcessor` configuration

The processor path to use when compiling the source files of this source set.

`runtimeClasspath` — **FileCollection**

Default value: `$output`, `${name}RuntimeClasspath` configuration

The classpath to use when executing the classes of this source set.

`java` — **(read-only)** **SourceDirectorySet**

The Java source files of this source set. Contains only `.java` files found in the Java source directories, and excludes all other files.

`java.srcDirs` — `Set<File>`

Default value: `src/$name/java`, e.g. *src/main/java*

The source directories containing the Java source files of this source set. You can set this to any value that is described in this section.

`java.destinationDirectory` — `DirectoryProperty`

Default value: `$buildDir/classes/java/$name`, e.g. *build/classes/java/main*

The directory to generate compiled Java sources into. You can set this to any value that is described in this section.

`resources` — **(read-only)** **SourceDirectorySet**

The resources of this source set. Contains only resources, and excludes any `.java` files found in the resource directories. Other plugins, such as the Groovy Plugin, exclude additional types of files from this collection.

`resources.srcDirs` — `Set<File>`

Default value: `[src/$name/resources]`

The directories containing the resources of this source set. You can set this to any type of value that is described in this section.

`allJava` — **(read-only)** **SourceDirectorySet**

Default value: Same as `java` property

All Java files of this source set. Some plugins, such as the Groovy Plugin, add additional Java source files to this collection.

`allSource` — **(read-only)** **SourceDirectorySet**

Default value: Sum of everything in the `resources` and `java` properties

All source files of this source set of any language. This includes all resource files and all Java source files. Some plugins, such as the Groovy Plugin, add additional source files to this collection.

Defining new source sets

See the integration test example in the *Testing in Java & JVM projects* chapter.

Some other simple source set examples

Adding a JAR containing the classes of a source set:

Example 650. Assembling a JAR for a source set

build.gradle

```
tasks.register('intTestJar', Jar) {
    from sourceSets.intTest.output
}
```

build.gradle.kts

```
tasks.register<Jar>("intTestJar") {
    from(sourceSets["intTest"].output)
}
```

Generating Javadoc for a source set:

Example 651. Generating the Javadoc for a source set

build.gradle

```
tasks.register('intTestJavadoc', Javadoc) {
    source sourceSets.intTest.allJava
}
```

build.gradle.kts

```
tasks.register<Javadoc>("intTestJavadoc") {
    source(sourceSets["intTest"].allJava)
}
```

Adding a test suite to run the tests in a source set:

Example 652. Running tests in a source set

build.gradle

```
tasks.register('intTest', Test) {
    testClassesDirs = sourceSets.intTest.output.classesDirs
    classpath = sourceSets.intTest.runtimeClasspath
}
```

build.gradle.kts

```
tasks.register<Test>("intTest") {
    testClassesDirs = sourceSets["intTest"].output.classesDirs
    classpath = sourceSets["intTest"].runtimeClasspath
}
```

Dependency management

The Java plugin adds a number of dependency configurations to your project, as shown below. Tasks such as `compileJava` and `test` then use one or more of those configurations to get the corresponding files and use them, for example by placing them on a compilation or runtime classpath.

Dependency configurations

NOTE	To find information on the `api` configuration, please consult the Java Library Plugin reference documentation and Dependency Management for Java Projects.

`implementation`

Implementation only dependencies.

`compileOnly`

Compile time only dependencies, not used at runtime.

`compileClasspath` **extends** `compileOnly, implementation`

Compile classpath, used when compiling source. Used by task `compileJava`.

`annotationProcessor`

Annotation processors used during compilation.

`runtimeOnly`

Runtime only dependencies.

`runtimeClasspath` **extends** `runtimeOnly, implementation`

Runtime classpath contains elements of the implementation, as well as runtime only elements.

`testImplementation` **extends** `implementation`

Implementation only dependencies for tests.

`testCompileOnly`

Additional dependencies only for compiling tests, not used at runtime.

`testCompileClasspath` **extends** `testCompileOnly, testImplementation`

Test compile classpath, used when compiling test sources. Used by task `compileTestJava`.

`testRuntimeOnly` **extends** `runtimeOnly`

Runtime only dependencies for running tests.

`testRuntimeClasspath` **extends** `testRuntimeOnly, testImplementation`

Runtime classpath for running tests. Used by task `test`.

`archives`

Artifacts (e.g. jars) produced by this project. Used by Gradle to determine "default" tasks to execute when building.

`default` **extends** `runtimeElements`

The default configuration used by a project dependency on this project. Contains the artifacts and dependencies required by this project at runtime.

The following diagrams show the dependency configurations for the *main* and *test* source sets respectively. You can use this legend to interpret the colors:

- Gray text — the configuration is *deprecated*.
- Green background — you can declare dependencies against the configuration.
- Blue-gray background — the configuration is for consumption by tasks, not for you to declare dependencies.
- Light blue background with monospace text — a task.

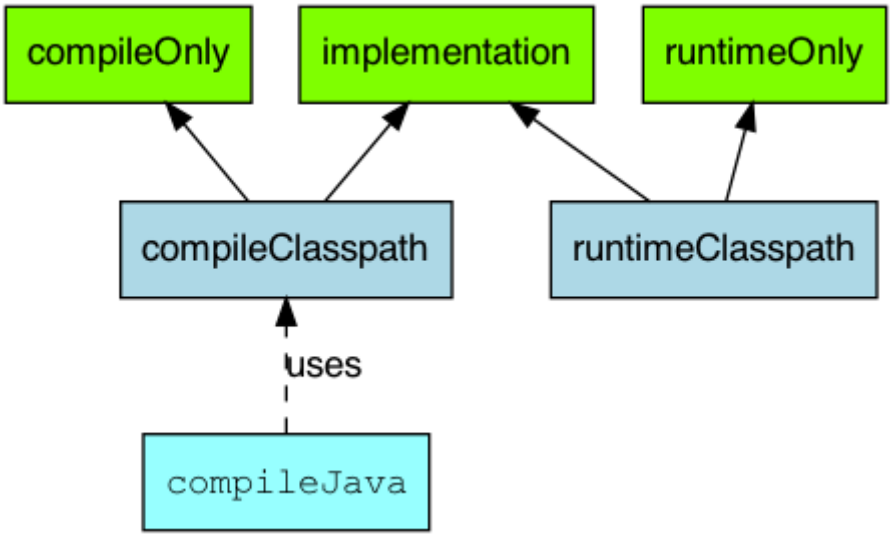


Figure 35. Java plugin - main source set dependency configurations

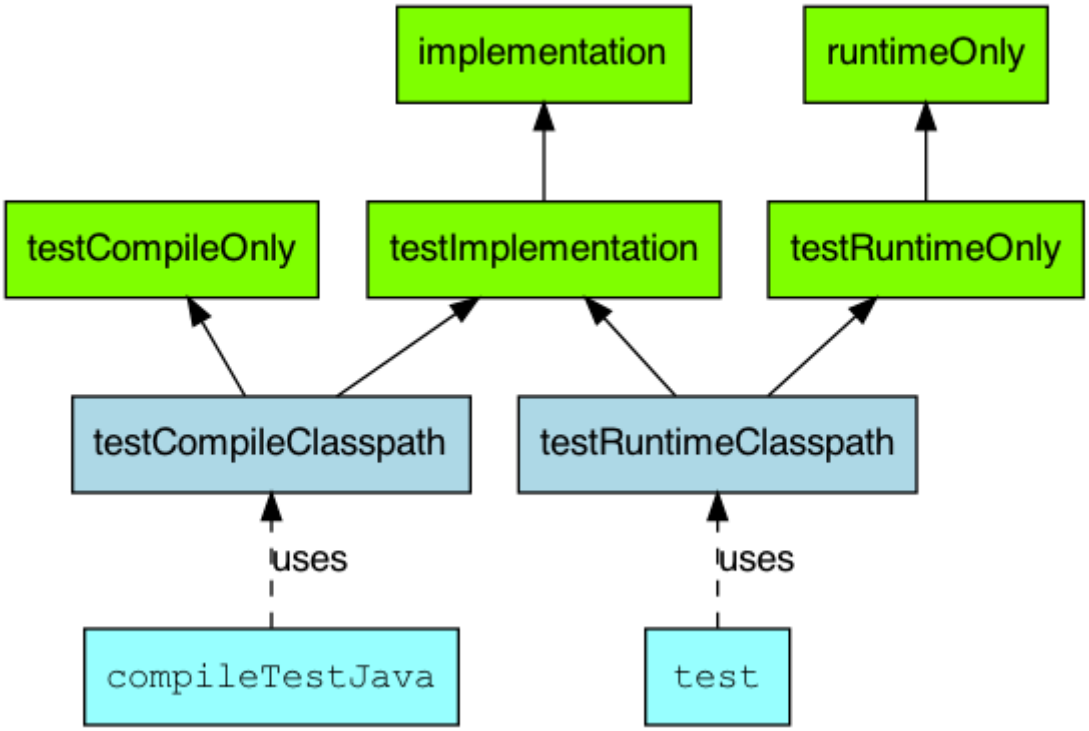


Figure 36. Java plugin - test source set dependency configurations

For each source set you add to the project, the Java plugins adds the following dependency configurations:

SourceSet dependency configurations

`sourceSetImplementation`

Compile time dependencies for the given source set. Used by `sourceSetCompileClasspath`, `sourceSetRuntimeClasspath`.

`sourceSetCompileOnly`

Compile time only dependencies for the given source set, not used at runtime.

`sourceSetCompileClasspath` extends `sourceSetCompileOnly`, `sourceSetImplementation`

Compile classpath, used when compiling source. Used by `compileSourceSetJava`.

`sourceSetAnnotationProcessor`

Annotation processors used during compilation of this source set.

`sourceSetRuntimeOnly`

Runtime only dependencies for the given source set.

`sourceSetRuntimeClasspath` extends `sourceSetRuntimeOnly`, `sourceSetImplementation`

Runtime classpath contains elements of the implementation, as well as runtime only elements.

Contributed extension

The Java plugin adds the `java` extension to the project. This allows to configure a number of Java related properties inside a dedicated DSL block.

Example 653. Using the `java` extension

build.gradle

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(11)
    }
}
```

build.gradle.kts

```
java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(11))
    }
}
```

`JavaVersion` **`sourceCompatibility`**

Java version compatibility to use when compiling Java source. Default value: version of the

current JVM in use.

`JavaVersion targetCompatibility`

Java version to generate classes for. Default value: `sourceCompatibility`.

`withJavadocJar()`

Automatically packages Javadoc and creates a variant `javadocElements` with an artifact `-javadoc.jar`, which will be part of the publication.

`withSourcesJar()`

Automatically packages source code and creates a variant `sourceElements` with an artifact `-sources.jar`, which will be part of the publication.

Directory properties

`String reporting.baseDir`

The name of the directory to generate reports into, relative to the build directory. Default value: `reports`

`(read-only) File reportsDir`

The directory to generate reports into. Default value: `buildDir/reporting.baseDir`

`String testResultsDirName`

The name of the directory to generate test result .xml files into, relative to the build directory. Default value: `test-results`

`(read-only) File testResultsDir`

The directory to generate test result .xml files into. Default value: `buildDir/testResultsDirName`

`String testReportDirName`

The name of the directory to generate the test report into, relative to the reports directory. Default value: `tests`

`(read-only) File testReportDir`

The directory to generate the test report into. Default value: `reportsDir/testReportDirName`

`String libsDirName`

The name of the directory to generate libraries into, relative to the build directory. Default value: `libs`

`(read-only) File libsDir`

The directory to generate libraries into. Default value: `buildDir/libsDirName`

`String distsDirName`

The name of the directory to generate distributions into, relative to the build directory. Default value: `distributions`

`(read-only) File distsDir`

The directory to generate distributions into. Default value: `buildDir/distsDirName`

`String docsDirName`

The name of the directory to generate documentation into, relative to the build directory. Default value: `docs`

`(read-only) File docsDir`

The directory to generate documentation into. Default value: `buildDir/docsDirName`

`String dependencyCacheDirName`

The name of the directory to use to cache source dependency information, relative to the build directory. Default value: `dependency-cache`

Other properties

`(read-only) SourceSetContainer sourceSets`

Contains the project's source sets. Default value: Not null SourceSetContainer

`String archivesBaseName`

The basename to use for archives, such as JAR or ZIP files. Default value: `projectName`

`Manifest manifest`

The manifest to include in all JAR files. Default value: an empty manifest.

Convention properties (deprecated)

The Java Plugin adds a number of convention properties to the project. You can use these properties in your build script as though they were properties of the project object. These are **deprecated** and superseded by the extension described above. See the JavaPluginConvention DSL documentation for information on them.

Testing

See the Testing in Java & JVM projects chapter for more details.

Publishing

`components.java`

A SoftwareComponent for publishing the production JAR created by the `jar` task. This component includes the runtime dependency information for the JAR.

See also the `java` extension.

Incremental Java compilation

Gradle comes with a sophisticated incremental Java compiler that is active by default.

This gives you the following benefits

- Incremental builds are much faster.

- The smallest possible number of class files are changed. Classes that don't need to be recompiled remain unchanged in the output directory. An example scenario when this is really useful is using JRebel — the fewer output classes are changed the quicker the JVM can use refreshed classes.

To help you understand how incremental compilation works, the following provides a high-level overview:

- Gradle will recompile all classes *affected* by a change.
- A class is *affected* if it has been changed or if it depends on another affected class. This works no matter if the other class is defined in the same project, another project or even an external library.
- A class's dependencies are determined from type references in its bytecode or symbol analysis via a compiler plugin.
- Since source-retention annotations are not visible in bytecode, changes to a source-retention annotation will result in full recompilation.
- You can improve incremental compilation performance by applying good software design principles like loose coupling. For instance, if you put an interface between a concrete class and its dependents, the dependent classes are only recompiled when the interface changes, but not when the implementation changes.
- The class analysis is cached in the project directory, so the first build after a clean checkout can be slower. Consider turning off the incremental compiler on your build server.
- The class analysis is also an output stored in the build cache, which means that if a compilation output is fetched from the build cache, then the incremental compilation analysis will be too and the next compilation will be incremental.

Known issues

- If you are using an annotation processor that reads resources (e.g. a configuration file), you need to declare those resources as an input of the compile task.
- If a resource file is changed, Gradle will trigger a full recompilation.
- Using a custom `executable` or `javaHome` deactivates some optimizations. The compile task does not use incremental build immediately after a compile error or if a Java constant changes. Use toolchains instead if possible.
- Having a source structure that does not match the package names, while legal for compilation, might end up causing trouble in the toolchain. Even more if annotation processing and caching are involved.

Incremental annotation processing

Starting with Gradle 4.7, the incremental compiler also supports incremental annotation processing. All annotation processors need to opt in to this feature, otherwise they will trigger a full recompilation.

As a user you can see which annotation processors are triggering full recompilations in the `--info`

log. Incremental annotation processing will be deactivated if a custom `executable` or `javaHome` is configured on the compile task.

Making an annotation processor incremental

Please first have a look at incremental Java compilation, as incremental annotation processing builds on top of it.

Gradle supports incremental compilation for two common categories of annotation processors: "isolating" and "aggregating". Please consult the information below to decide which category fits your processor.

You can then register your processor for incremental compilation using a file in the processor's META-INF directory. The format is one line per processor, with the fully qualified name of the processor class and its case-insensitive category separated by a comma.

Example: Registering incremental annotation processors

processor/src/main/resources/META-INF/gradle/incremental.annotation.processors

```
org.gradle.EntityProcessor,isolating
org.gradle.ServiceRegistryProcessor,dynamic
```

If your processor can only decide at runtime whether it is incremental or not, you can declare it as "dynamic" in the META-INF descriptor and return its true type at runtime using the Processor#getSupportedOptions() method.

Example: Registering incremental annotation processors dynamically

processor/src/main/java/org/gradle/ServiceRegistryProcessor.java

```
@Override
public Set<String> getSupportedOptions() {
    return Collections.singleton("org.gradle.annotation.processing.aggregating");
}
```

Both categories have the following limitations:

- They can only read `CLASS` or `RUNTIME` retention annotations.
- They can only read parameter names if the user passes the `-parameters` compiler argument.
- They must generate their files using the Filer API. Writing files any other way will result in silent failures later on, as these files won't be cleaned up correctly. If your processor does this, it cannot be incremental.
- They must not depend on compiler-specific APIs like `com.sun.source.util.Trees`. Gradle wraps the processing APIs, so attempts to cast to compiler-specific types will fail. If your processor does this, it cannot be incremental, unless you have some fallback mechanism.
- If they use Filer#createResource, the `location` argument must be one of these values from StandardLocation: `CLASS_OUTPUT`, `SOURCE_OUTPUT`, or `NATIVE_HEADER_OUTPUT`. Any other argument

will disable incremental processing.

"Isolating" annotation processors

The fastest category, these look at each annotated element in isolation, creating generated files or validation messages for it. For instance an `EntityProcessor` could create a `<TypeName>Repository` for each type annotated with `@Entity`.

Example: An isolated annotation processor

processor/src/main/java/org/gradle/EntityProcessor.java

```
Set<? extends Element> entities = roundEnv.getElementsAnnotatedWith(entityAnnotation);
for (Element entity : entities) {
    createRepository((TypeElement) entity);
}
```

"Isolating" processors have the following additional limitations:

- They must make all decisions (code generation, validation messages) for an annotated type based on information reachable from its AST. This means you can analyze the types' super-class, method return types, annotations etc., even transitively. But you cannot make decisions based on unrelated elements in the RoundEnvironment. Doing so will result in silent failures because too few files will be recompiled later. If your processor needs to make decisions based on a combination of otherwise unrelated elements, mark it as "aggregating" instead.
- They must provide exactly one originating element for each file generated with the `Filer` API. If zero or many originating elements are provided, Gradle will recompile all source files.

When a source file is recompiled, Gradle will recompile all files generated from it. When a source file is deleted, the files generated from it are deleted.

"Aggregating" annotation processors

These can aggregate several source files into one or more output files or validation messages. For instance, a `ServiceRegistryProcessor` could create a single `ServiceRegistry` with one method for each type annotated with `@Service`.

Example: An aggregating annotation processor

processor/src/main/java/org/gradle/ServiceRegistryProcessor.java

```
JavaFileObject serviceRegistry = filer.createSourceFile("ServiceRegistry");
Writer writer = serviceRegistry.openWriter();
writer.write("public class ServiceRegistry {");
for (Element service : roundEnv.getElementsAnnotatedWith(serviceAnnotation)) {
    addServiceCreationMethod(writer, (TypeElement) service);
}
writer.write("}");
writer.close();
```

Gradle will always reprocess (but not recompile) all annotated files that the processor was registered for. Gradle will always recompile any files the processor generates.

State of support in popular annotation processors

NOTE	Many popular annotation processors support incremental annotation processing (see the table below). Check with the annotation processor project directly for the most up-to-date information and documentation.

Annotation Processor	Supported since	Details
Auto Value	1.6.3	N/A
Auto Service	1.6.3	N/A
Auto Value extensions	Partly supported.	Details in issue
Butterknife	10.2.0	N/A
Lombok	1.16.22	N/A
DataBinding	AGP 3.5.0-alpha5	Hidden behind a feature toggle
Dagger	2.18	2.18 Feature toggle support, 2.24 Enabled by default
kapt	1.3.30	Hidden behind a feature toggle
Toothpick	2.0	N/A
Glide	4.9.0	N/A
Android-State	1.3.0	N/A
Parceler	1.1.11	N/A
Dart and Henson	3.1.0	N/A
MapStruct	1.4.0.Beta1	N/A
Assisted Inject	0.5.0	N/A
Realm	5.11.0	N/A
Requery	Open issue	N/A

Annotation Processor	Supported since	Details
EventBus	3.2.0	N/A
EclipseLink	Open issue	N/A
PermissionsDispatcher	4.2.0	N/A
Immutables	Open issue	N/A
Room	2.2.0	2.2.0 Feature toggle support, 2.3.0-alpha02 Enabled by default
Lifecycle	2.2.0-alpha02	N/A
AndroidAnnotations	4.7.0	N/A
DBFlow	Open issue	N/A
AndServer	Open issue	N/A
Litho	0.25.0	N/A
Moxy	2.0	N/A
Epoxy	4.0.0-beta1	N/A
JPA Static Metamodel Generator	5.4.11	N/A
DeepLinkDispatch	5.0.0-beta01	Hidden behind a feature toggle
Shortbread	1.1.0	N/A

Compilation avoidance

If a dependent project has changed in an ABI-compatible way (only its private API has changed), then Java compilation tasks will be up-to-date. This means that if project `A` depends on project `B` and a class in `B` is changed in an ABI-compatible way (typically, changing only the body of a method), then Gradle won't recompile `A`.

Some of the types of changes that do not affect the public API and are ignored:

- Changing a method body
- Changing a comment

- Adding, removing or changing private methods, fields, or inner classes
- Adding, removing or changing a resource
- Changing the name of jars or directories in the classpath
- Renaming a parameter

Since implementation details matter for annotation processors, they must be declared separately on the annotation processor path. Gradle ignores annotation processors on the compile classpath.

Example 654. Declaring annotation processors

build.gradle

```
dependencies {
    // The dagger compiler and its transitive dependencies will only be found
on annotation processing classpath
    annotationProcessor 'com.google.dagger:dagger-compiler:2.8'

    // And we still need the Dagger library on the compile classpath itself
    implementation 'com.google.dagger:dagger:2.8'
}
```

build.gradle.kts

```
dependencies {
    // The dagger compiler and its transitive dependencies will only be found
on annotation processing classpath
    annotationProcessor("com.google.dagger:dagger-compiler:2.8")

    // And we still need the Dagger library on the compile classpath itself
    implementation("com.google.dagger:dagger:2.8")
}
```

Variant aware selection

The whole set of JVM plugins leverage variant aware resolution for the dependencies used. They also install a set of attributes compatibility and disambiguation rules to configure the Gradle attributes for the specifics of the JVM ecosystem.

The Java Library Plugin

The Java Library plugin expands the capabilities of the Java Plugin (`java`) by providing specific knowledge about Java libraries. In particular, a Java library exposes an API to consumers (i.e., other projects using the Java or the Java Library plugin). All the source sets, tasks and configurations

exposed by the Java plugin are implicitly available when using this plugin.

Usage

To use the Java Library plugin, include the following in your build script:

Example 655. Using the Java Library plugin

build.gradle

```
plugins {
    id 'java-library'
}
```

build.gradle.kts

```
plugins {
    `java-library`
}
```

API and implementation separation

The key difference between the standard Java plugin and the Java Library plugin is that the latter introduces the concept of an *API* exposed to consumers. A library is a Java component meant to be consumed by other components. It's a very common use case in multi-project builds, but also as soon as you have external dependencies.

The plugin exposes two configurations that can be used to declare dependencies: `api` and `implementation`. The `api` configuration should be used to declare dependencies which are exported by the library API, whereas the `implementation` configuration should be used to declare dependencies which are internal to the component.

Example 656. Declaring API and implementation dependencies

build.gradle

```
dependencies {
    api 'org.apache.httpcomponents:httpclient:4.5.7'
    implementation 'org.apache.commons:commons-lang3:3.5'
}
```

build.gradle.kts

```
dependencies {
    api("org.apache.httpcomponents:httpclient:4.5.7")
    implementation("org.apache.commons:commons-lang3:3.5")
}
```

Dependencies appearing in the `api` configurations will be transitively exposed to consumers of the library, and as such will appear on the compile classpath of consumers. Dependencies found in the `implementation` configuration will, on the other hand, not be exposed to consumers, and therefore not leak into the consumers' compile classpath. This comes with several benefits:

- dependencies do not leak into the compile classpath of consumers anymore, so you will never accidentally depend on a transitive dependency
- faster compilation thanks to reduced classpath size
- less recompilations when implementation dependencies change: consumers would not need to be recompiled
- cleaner publishing: when used in conjunction with the new `maven-publish` plugin, Java libraries produce POM files that distinguish exactly between what is required to compile against the library and what is required to use the library at runtime (in other words, don't mix what is needed to compile the library itself and what is needed to compile against the library).

NOTE The `compile` and `runtime` configurations have been removed with Gradle 7.0. Please refer to the upgrade guide how to migrate to `implementation` and `api` configurations`.

If your build consumes a published module with POM metadata, the Java and Java Library plugins both honor api and implementation separation through the scopes used in the POM. Meaning that the compile classpath only includes Maven `compile` scoped dependencies, while the runtime classpath adds the Maven `runtime` scoped dependencies as well.

This often does not have an effect on modules published with Maven, where the POM that defines the project is directly published as metadata. There, the compile scope includes both dependencies that were required to compile the project (i.e. implementation dependencies) and dependencies

required to compile against the published library (i.e. API dependencies). For most published libraries, this means that all dependencies belong to the compile scope. If you encounter such an issue with an existing library, you can consider a component metadata rule to fix the incorrect metadata in your build. However, as mentioned above, if the library is published with Gradle, the produced POM file only puts `api` dependencies into the compile scope and the remaining `implementation` dependencies into the runtime scope.

If your build consumes modules with Ivy metadata, you might be able to activate api and implementation separation as described here if all modules follow a certain structure.

NOTE	Separating compile and runtime scope of modules is active by default in Gradle 5.0+. In Gradle 4.6+, you need to activate it by adding `enableFeaturePreview('IMPROVED_POM_SUPPORT')` in *settings.gradle*.

Recognizing API and implementation dependencies

This section will help you identify API and Implementation dependencies in your code using simple rules of thumb. The first of these is:

- Prefer the `implementation` configuration over `api` when possible

This keeps the dependencies off of the consumer's compilation classpath. In addition, the consumers will immediately fail to compile if any implementation types accidentally leak into the public API.

So when should you use the `api` configuration? An API dependency is one that contains at least one type that is exposed in the library binary interface, often referred to as its ABI (Application Binary Interface). This includes, but is not limited to:

- types used in super classes or interfaces
- types used in public method parameters, including generic parameter types (where *public* is something that is visible to compilers. I.e. , *public*, *protected* and *package private* members in the Java world)
- types used in public fields
- public annotation types

By contrast, any type that is used in the following list is irrelevant to the ABI, and therefore should be declared as an `implementation` dependency:

- types exclusively used in method bodies
- types exclusively used in private members
- types exclusively found in internal classes (future versions of Gradle will let you declare which packages belong to the public API)

The following class makes use of a couple of third-party libraries, one of which is exposed in the class's public API and the other is only used internally. The import statements don't help us determine which is which, so we have to look at the fields, constructors and methods instead:

Example: Making the difference between API and implementation

src/main/java/org/gradle/HttpClientWrapper.java

```java
// The following types can appear anywhere in the code
// but say nothing about API or implementation usage
import org.apache.commons.lang3.exception.ExceptionUtils;
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.HttpStatus;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.UnsupportedEncodingException;

public class HttpClientWrapper {

    private final HttpClient client; // private member: implementation details

    // HttpClient is used as a parameter of a public method
    // so "leaks" into the public API of this component
    public HttpClientWrapper(HttpClient client) {
        this.client = client;
    }

    // public methods belongs to your API
    public byte[] doRawGet(String url) {
        HttpGet request = new HttpGet(url);
        try {
            HttpEntity entity = doGet(request);
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            entity.writeTo(baos);
            return baos.toByteArray();
        } catch (Exception e) {
            ExceptionUtils.rethrow(e); // this dependency is internal only
        } finally {
            request.releaseConnection();
        }
        return null;
    }

    // HttpGet and HttpEntity are used in a private method, so they don't belong to
the API
    private HttpEntity doGet(HttpGet get) throws Exception {
        HttpResponse response = client.execute(get);
        if (response.getStatusLine().getStatusCode() != HttpStatus.SC_OK) {
            System.err.println("Method failed: " + response.getStatusLine());
        }
        return response.getEntity();
    }
}
```

The *public* constructor of `HttpClientWrapper` uses `HttpClient` as a parameter, so it is exposed to consumers and therefore belongs to the API. Note that `HttpGet` and `HttpEntity` are used in the signature of a *private* method, and so they don't count towards making HttpClient an API dependency.

On the other hand, the `ExceptionUtils` type, coming from the `commons-lang` library, is only used in a method body (not in its signature), so it's an implementation dependency.

Therefore, we can deduce that `httpclient` is an API dependency, whereas `commons-lang` is an implementation dependency. This conclusion translates into the following declaration in the build script:

Example 657. Declaring API and implementation dependencies

build.gradle

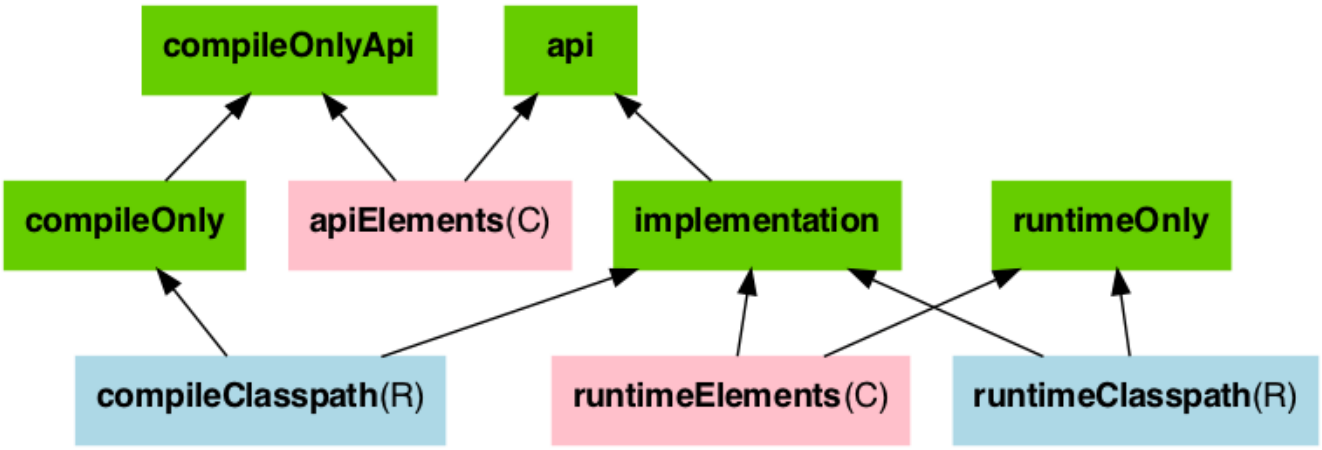
```
dependencies {
    api 'org.apache.httpcomponents:httpclient:4.5.7'
    implementation 'org.apache.commons:commons-lang3:3.5'
}
```

build.gradle.kts

```
dependencies {
    api("org.apache.httpcomponents:httpclient:4.5.7")
    implementation("org.apache.commons:commons-lang3:3.5")
}
```

The Java Library plugin configurations

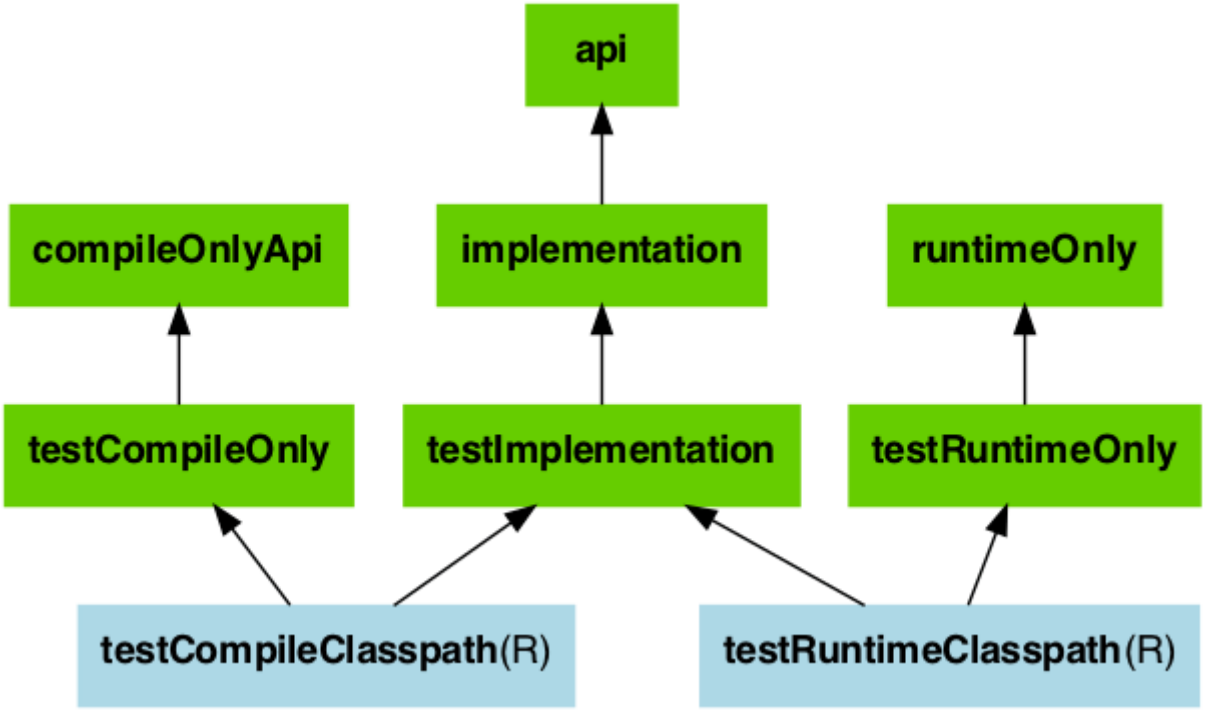
The following graph describes how configurations are setup when the Java Library plugin is in use.



- The configurations in *green* are the ones a user should use to declare dependencies

- The configurations in *pink* are the ones used when a component compiles, or runs against the library
- The configurations in *blue* are internal to the component, for its own use

And the next graph describes the test configurations setup:



The role of each configuration is described in the following tables:

Table 35. Java Library plugin - configurations used to declare dependencies

Configuration name	Role	Consumable?	Resolvable?	Description
`api`	Declaring API dependencies	no	no	This is where you declare dependencies which are transitively exported to consumers, for compile time and runtime.
`implementation`	Declaring implementation dependencies	no	no	This is where you declare dependencies which are purely internal and not meant to be exposed to consumers (they are still exposed to consumers at runtime).
`compileOnly`	Declaring compile only dependencies	no	no	This is where you declare dependencies which are required at compile time, but not at runtime. This typically includes dependencies which are shaded when found at runtime.
`compileOnlyApi`	Declaring compile only API dependencies	no	no	This is where you declare dependencies which are required at compile time by your module and consumers, but not at runtime. This typically includes dependencies which are shaded when found at runtime.

Configuration name	Role	Consumable?	Resolvable?	Description
`runtimeOnly`	Declaring runtime dependencies	no	no	This is where you declare dependencies which are only required at runtime, and not at compile time.
`testImplementation`	Test dependencies	no	no	This is where you declare dependencies which are used to compile tests.
`testCompileOnly`	Declaring test compile only dependencies	no	no	This is where you declare dependencies which are only required at test compile time, but should not leak into the runtime. This typically includes dependencies which are shaded when found at runtime.
`testRuntimeOnly`	Declaring test runtime dependencies	no	no	This is where you declare dependencies which are only required at test runtime, and not at test compile time.

Table 36. Java Library plugin — configurations used by consumers

Configuration name	Role	Consumable?	Resolvable?	Description
`apiElements`	For compiling against this library	yes	no	This configuration is meant to be used by consumers, to retrieve all the elements necessary to compile against this library. Unlike the `default` configuration, this doesn't leak implementation or runtime dependencies.
`runtimeElements`	For executing this library	yes	no	This configuration is meant to be used by consumers, to retrieve all the elements necessary to run against this library.

Table 37. Java Library plugin - configurations used by the library itself

Configuration name	Role	Consumable?	Resolvable?	Description
compileClasspath	For compiling this library	no	yes	This configuration contains the compile classpath of this library, and is therefore used when invoking the java compiler to compile it.
runtimeClasspath	For executing this library	no	yes	This configuration contains the runtime classpath of this library
testCompileClasspath	For compiling the tests of this library	no	yes	This configuration contains the test compile classpath of this library.
testRuntimeClasspath	For executing tests of this library	no	yes	This configuration contains the test runtime classpath of this library

Building Modules for the Java Module System

Since Java 9, Java itself offers a module system that allows for strict encapsulation during compile and runtime. You can turn a Java library into a *Java Module* by creating a `module-info.java` file in the `main/java` source folder.

```
src
 └─── main
     └─── java
         └─── module-info.java
```

In the module info file, you declare a *module name*, which packages of your module you want to *export* and which other modules you *require*.

module-info.java file

```
module org.gradle.sample {
    requires com.google.gson;          // real module
    requires org.apache.commons.lang3; // automatic module
    // commons-cli-1.4.jar is not a module and cannot be required
}
```

To tell the Java compiler that a Jar is a module, as opposed to a traditional Java library, Gradle needs to place it on the so called *module path*. It is an alternative to the *classpath*, which is the traditional way to tell the compiler about compiled dependencies. Gradle will automatically put a Jar of your dependencies on the module path, instead of the classpath, if these three things are true:

- `java.modularity.inferModulePath` is **not** turned off
- We are actually building a module (as opposed to a traditional library) which we expressed by adding the `module-info.java` file. (Another option is to add the `Automatic-Module-Name` Jar manifest attribute as described further down.)
- The Jar our module depends on is itself a module, which Gradles decides based on the presence of a `module-info.class` — the compiled version of the module descriptor — in the Jar. (Or, alternatively, the presence of an `Automatic-Module-Name` attribute the Jar manifest)

In the following, some more details about defining Java modules and how that interacts with Gradle's dependency management are described. You can also look at a ready made example to try out the Java Module support directly.

Declaring module dependencies

There is a direct relationship to the dependencies you declare in the build file and the module dependencies you declare in the `module-info.java` file. Ideally the declarations should be in sync as seen in the following table.

Table 38. Mapping between Java module directives and Gradle configurations to declare dependencies

Java Module Directive	Gradle Configuration	Purpose
`requires`	`implementation`	Declaring implementation dependencies
`requires transitive`	`api`	Declaring API dependencies
`requires static`	`compileOnly`	Declaring compile only dependencies
`requires static transitive`	`compileOnlyApi`	Declaring compile only API dependencies

Gradle currently does not automatically check if the dependency declarations are in sync. This may be added in future versions.

For more details on declaring module dependencies, please refer to documentation on the Java Module System.

Declaring package visibility and services

The Java module system supports additional more fine granular encapsulation concepts than Gradle itself currently does. For example, you explicitly need to declare which packages are part of your API and which are only visible inside your module. Some of these capabilities might be added to Gradle itself in future versions. For now, please refer to documentation on the Java Module System to learn how to use these features in Java Modules.

Declaring module versions

Java Modules also have a version that is encoded as part of the module identity in the `module-info.class` file. This version can be inspected when a module is running.

Example 658. Declare the module version in the build script or directly as compile task option

build.gradle

```
version = '1.2'

tasks.named('compileJava') {
    // use the project's version or define one directly
    options.javaModuleVersion = provider { project.version }
}
```

build.gradle.kts

```
version = "1.2"

tasks.compileJava {
    // use the project's version or define one directly
    options.javaModuleVersion.set(provider { project.version as String })
}
```

Using libraries that are not modules

You probably want to use external libraries, like OSS libraries from Maven Central, in your modular Java project. Some libraries, in their newer versions, are already full modules with a module descriptor. For example, `com.google.code.gson:gson:2.8.9` that has the module name `com.google.gson`.

Others, like `org.apache.commons:commons-lang3:3.10`, may not offer a full module descriptor but will at least contain an `Automatic-Module-Name` entry in their manifest file to define the module's name (`org.apache.commons.lang3` in the example). Such modules, that only have a name as module description, are called *automatic module* that export **all** their packages and can read **all** modules on the module path.

A third case are traditional libraries that provide no module information at all — for example `commons-cli:commons-cli:1.4`. Gradle puts such libraries on the classpath instead of the module path. The classpath is then treated as one module (the so called *unnamed* module) by Java.

Example 659. Dependencies to modules and libraries declared in build file

build.gradle

```
dependencies {
    implementation 'com.google.code.gson:gson:2.8.9'       // real module
    implementation 'org.apache.commons:commons-lang3:3.10' // automatic
module
    implementation 'commons-cli:commons-cli:1.4'           // plain library
}
```

build.gradle.kts

```
dependencies {
    implementation("com.google.code.gson:gson:2.8.9")       // real module
    implementation("org.apache.commons:commons-lang3:3.10") // automatic
module
    implementation("commons-cli:commons-cli:1.4")           // plain library
}
```

Module dependencies declared in module-info.java file

```
module org.gradle.sample.lib {
    requires com.google.gson;          // real module
    requires org.apache.commons.lang3; // automatic module
    // commons-cli-1.4.jar is not a module and cannot be required
}
```

While a real module cannot directly depend on the unnamed module (only by adding command line flags), automatic modules can also see the unnamed module. Thus, if you cannot avoid to rely on a library without module information, you can wrap that library in an automatic module as part of your project. How you do that is described in the next section.

Another way to deal with non-modules is to enrich existing Jars with module descriptors yourself using artifact transforms. This sample contains a small *buildSrc* plugin registering such a transform which you may use and adjust to your needs. This can be interesting if you want to build a fully modular application and want the java runtime to treat everything as a real module.

Disabling Java Module support

In rare cases, you might want to disable the built-in Java Module support and define the module path by other means. To achieve this, you can disable the functionality to automatically put any Jar on the module path. Then Gradle puts Jars with module information on the classpath, even if you have a `module-info.java` in your source set. This corresponds to the behaviour of Gradle versions

<7.0.

To make this work, you need to set `modularity.inferModulePath = false` on the Java extension (for all tasks) or on individual tasks.

Example 660. Disable Gradle's module path inference

build.gradle

```
java {
    modularity.inferModulePath = false
}

tasks.named('compileJava') {
    modularity.inferModulePath = false
}
```

build.gradle.kts

```
java {
    modularity.inferModulePath.set(false)
}

tasks.compileJava {
    modularity.inferModulePath.set(false)
}
```

Building an automatic module

If you can, you should always write complete `module-info.java` descriptors for your modules. Still, there are a few cases where you might consider to (initally) only provide a *module name* for an automatic module:

- You are working on a library that is **not** a module but you want to make it usable as such in the next release. Adding an `Automatic-Module-Name` is a good first step (most popular OSS libraries on Maven central have done it by now).
- As discussed in the previous section, an automatic module can be used as an adapter between your real modules and a traditional library on the classpath.

To turn a normal Java project into an *automatic module*, just add the manifest entry with the module name:

Example 661. Declare an automatic module name as Jar manifest attribute

build.gradle

```
tasks.named('jar') {
    manifest {
        attributes('Automatic-Module-Name': 'org.gradle.sample')
    }
}
```

build.gradle.kts

```
tasks.jar {
    manifest {
        attributes("Automatic-Module-Name" to "org.gradle.sample")
    }
}
```

NOTE You can define an automatic module as part of a multi-project that otherwise defines real modules (e.g. as an adapter to another library). While this works fine in the Gradle build, such automatic module projects are not correctly recognized by IDEA/Eclipse at the moment. You can work around it by manually adding the Jar built for the automatic module to the dependencies of the project that does not find it in the IDE's UI.

Using classes instead of jar for compilation

A feature of the `java-library` plugin is that projects which consume the library only require the classes folder for compilation, instead of the full JAR. This enables lighter inter-project dependencies as resources processing (`processResources` task) and archive construction (`jar` task) are no longer executed when only Java code compilation is performed during development.

NOTE The usage or not of the classes output instead of the JAR is a *consumer* decision. For example, Groovy consumers will request classes *and* processed resources as these may be needed for executing AST transformation as part of the compilation process.

Increased memory usage for consumers

An indirect consequence is that up-to-date checking will require more memory, because Gradle will snapshot individual class files instead of a single jar. This may lead to increased memory consumption for large projects, with the benefit of having the `compileJava` task up-to-date in more cases (e.g. changing resources no longer changes the input for `compileJava` tasks of upstream projects)

Significant build performance drop on Windows for huge multi-projects

Another side effect of the snapshotting of individual class files, only affecting Windows systems, is that the performance can significantly drop when processing a very large amount of class files on the compile classpath. This only concerns very large multi-projects where a lot of classes are present on the classpath by using many `api` or (deprecated) `compile` dependencies. To mitigate this, you can set the `org.gradle.java.compile-classpath-packaging` system property to `true` to change the behavior of the Java Library plugin to use jars instead of class folders for everything on the compile classpath. Note, since this has other performance impacts and potentially side effects, by triggering all jar tasks at compile time, it is only recommended to activate this if you suffer from the described performance issue on Windows.

Distributing a library

Aside from publishing a library to a component repository, you may sometimes need to package a library and its dependencies in a distribution deliverable. The Java Library Distribution Plugin is there to help you do just that.

The Java Library Distribution Plugin

The Java library distribution plugin adds support for building a distribution ZIP for a Java library. The distribution contains the JAR file for the library and its dependencies.

Usage

To use the Java library distribution plugin, include the following in your build script:

Example 662. Using the Java library distribution plugin

build.gradle

```
plugins {
    id 'java-library-distribution'
}
```

build.gradle.kts

```
plugins {
    `java-library-distribution`
}
```

To define the name for the distribution you have to set the `baseName` property as shown below:

Example 663. Configure the distribution name

build.gradle

```
distributions {
    main {
        distributionBaseName = 'my-name'
    }
}
```

build.gradle.kts

```
distributions {
    main {
        distributionBaseName.set("my-name")
    }
}
```

The plugin builds a distribution for your library. The distribution will package up the runtime dependencies of the library. All files stored in `src/main/dist` will be added to the root of the archive distribution. You can run "`gradle distZip`" to create a ZIP file containing the distribution.

Tasks

The Java library distribution plugin adds the following tasks to the project.

`distZip` — **Zip**

Depends on: `jar`

Creates a full distribution ZIP archive including runtime libraries.

Including other resources in the distribution

All of the files from the `src/dist` directory are copied. To include any static files in the distribution, simply arrange them in the `src/dist` directory, or add them to the content of the distribution.

Example 664. Include files in the distribution

build.gradle

```
distributions {
    main {
        distributionBaseName = 'my-name'
        contents {
            from 'src/dist'
        }
    }
}
```

build.gradle.kts

```
distributions {
    main {
        distributionBaseName.set("my-name")
        contents {
            from("src/dist")
        }
    }
}
```

The Java Platform Plugin

The Java Platform plugin brings the ability to declare platforms for the Java ecosystem. A platform can be used for different purposes:

- a description of modules which are published together (and for example, share the same version)
- a set of recommended versions for heterogeneous libraries. A typical example includes the Spring Boot BOM
- sharing a set of dependency versions between subprojects

A platform is a special kind of software component which doesn't contain any sources: it is only used to reference other libraries, so that they play well together during dependency resolution.

Platforms can be published as Gradle Module Metadata and Maven BOMs.

NOTE	The `java-platform` plugin cannot be used in combination with the `java` or `java-library` plugins in a given project. Conceptually a project is either a platform, with no binaries, *or* produces binaries.

Usage

To use the Java Platform plugin, include the following in your build script:

Example 665. Using the Java Platform plugin

build.gradle

```
plugins {
    id 'java-platform'
}
```

build.gradle.kts

```
plugins {
    `java-platform`
}
```

API and runtime separation

A major difference between a Maven BOM and a Java platform is that in Gradle dependencies and constraints are declared and scoped to a configuration and the ones extending it. While many users will only care about declaring constraints for *compile time* dependencies, thus inherited by runtime and tests ones, it allows declaring dependencies or constraints that only apply to runtime or test.

For this purpose, the plugin exposes two configurations that can be used to declare dependencies: `api` and `runtime`. The `api` configuration should be used to declare constraints and dependencies which should be used when compiling against the platform, whereas the `runtime` configuration should be used to declare constraints or dependencies which are visible at runtime.

Example 666. Declaring API and runtime constraints

build.gradle

```
dependencies {
    constraints {
        api 'commons-httpclient:commons-httpclient:3.1'
        runtime 'org.postgresql:postgresql:42.2.5'
    }
}
```

build.gradle.kts

```
dependencies {
    constraints {
        api("commons-httpclient:commons-httpclient:3.1")
        runtime("org.postgresql:postgresql:42.2.5")
    }
}
```

Note that this example makes use of *constraints* and not dependencies. In general, this is what you would like to do: constraints will only apply if such a component is added to the dependency graph, either directly or transitively. This means that all constraints listed in a platform would not add a dependency unless another component brings it in: they can be seen as *recommendations*.

NOTE For example, if a platform declares a constraint on `org:foo:1.1`, and that nothing else brings in a dependency on `foo`, `foo` will *not* appear in the graph. However, if `foo` appears, then usual conflict resolution would kick in. If a dependency brings in `org:foo:1.0`, then we would select `org:foo:1.1` to satisfy the platform constraint.

By default, in order to avoid the common mistake of adding a dependency in a platform instead of a constraint, Gradle will fail if you try to do so. If, for some reason, you also want to add *dependencies* in addition to constraints, you need to enable it explicitly:

Example 667. Allowing declaration of dependencies

build.gradle

```
javaPlatform {
    allowDependencies()
}
```

build.gradle.kts

```
javaPlatform {
    allowDependencies()
}
```

Local project constraints

If you have a multi-project build and want to publish a platform that links to subprojects, you can do it by declaring constraints on the subprojects which belong to the platform, as in the example below:

Example 668. Declaring constraints on subprojects

build.gradle

```
dependencies {
    constraints {
        api project(":core")
        api project(":lib")
    }
}
```

build.gradle.kts

```
dependencies {
    constraints {
        api(project(":core"))
        api(project(":lib"))
    }
}
```

The project notation will become a classical `group:name:version` notation in the published metadata.

Sourcing constraints from another platform

Sometimes the platform you define is an extension of another existing platform.

In order to have your platform include the constraints from that third party platform, it needs to be imported as a `platform` *dependency*:

Example 669. Importing a platform

build.gradle

```
javaPlatform {
    allowDependencies()
}

dependencies {
    api platform('com.fasterxml.jackson:jackson-bom:2.9.8')
}
```

build.gradle.kts

```
javaPlatform {
    allowDependencies()
}

dependencies {
    api(platform("com.fasterxml.jackson:jackson-bom:2.9.8"))
}
```

Publishing platforms

Publishing Java platforms is done by applying the `maven-publish` plugin and configuring a Maven publication that uses the `javaPlatform` component:

Example 670. Publishing as a BOM

build.gradle

```
publishing {
    publications {
        myPlatform(MavenPublication) {
            from components.javaPlatform
        }
    }
}
```

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("myPlatform") {
            from(components["javaPlatform"])
        }
    }
}
```

This will generate a BOM file for the platform, with a `<dependencyManagement>` block where its `<dependencies>` correspond to the constraints defined in the platform module.

Consuming platforms

Because a Java Platform is a special kind of component, a dependency on a Java platform has to be declared using the `platform` or `enforcedPlatform` keyword, as explained in the managing transitive dependencies section. For example, if you want to share dependency versions between subprojects, you can define a platform module which would declare all versions:

Example 671. Recommend versions in a platform module

build.gradle

```
dependencies {
    constraints {
        // Platform declares some versions of libraries used in subprojects
        api 'commons-httpclient:commons-httpclient:3.1'
        api 'org.apache.commons:commons-lang3:3.8.1'
    }
}
```

build.gradle.kts

```
dependencies {
    constraints {
        // Platform declares some versions of libraries used in subprojects
        api("commons-httpclient:commons-httpclient:3.1")
        api("org.apache.commons:commons-lang3:3.8.1")
    }
}
```

And then have subprojects depend on the platform to get recommendations:

Example 672. Get recommendations from a platform

build.gradle

```
dependencies {
    // get recommended versions from the platform project
    api platform(project(':platform'))
    // no version required
    api 'commons-httpclient:commons-httpclient'
}
```

build.gradle.kts

```
dependencies {
    // get recommended versions from the platform project
    api(platform(project(":platform")))
    // no version required
    api("commons-httpclient:commons-httpclient")
}
```

Maven Publish Plugin

The Maven Publish Plugin provides the ability to publish build artifacts to an Apache Maven repository. A module published to a Maven repository can be consumed by Maven, Gradle (see Declaring Dependencies) and other tools that understand the Maven repository format. You can learn about the fundamentals of publishing in Publishing Overview.

Usage

To use the Maven Publish Plugin, include the following in your build script:

Example 673. Applying the Maven Publish Plugin

build.gradle

```
plugins {
    id 'maven-publish'
}
```

build.gradle.kts

```
plugins {
    `maven-publish`
}
```

The Maven Publish Plugin uses an extension on the project named `publishing` of type PublishingExtension. This extension provides a container of named publications and a container of named repositories. The Maven Publish Plugin works with MavenPublication publications and MavenArtifactRepository repositories.

Tasks

`generatePomFileFor`**`PubName`**`Publication` — **GenerateMavenPom**

Creates a POM file for the publication named *PubName*, populating the known metadata such as project name, project version, and the dependencies. The default location for the POM file is *build/publications/$pubName/pom-default.xml*.

`publish`**`PubName`**`PublicationTo`**`RepoName`**`Repository` — **PublishToMavenRepository**

Publishes the *PubName* publication to the repository named *RepoName*. If you have a repository definition without an explicit name, *RepoName* will be "Maven".

`publish`**`PubName`**`PublicationToMavenLocal` — **PublishToMavenLocal**

Copies the *PubName* publication to the local Maven cache — typically *$USER_HOME/.m2/repository* — along with the publication's POM file and other metadata.

`publish`

Depends on: All `publishPubNamePublicationToRepoNameRepository` tasks

An aggregate task that publishes all defined publications to all defined repositories. It does *not* include copying publications to the local Maven cache.

`publishToMavenLocal`

Depends on: All `publishPubNamePublicationToMavenLocal` tasks

Copies all defined publications to the local Maven cache, including their metadata (POM files,

etc.).

Publications

This plugin provides publications of type MavenPublication. To learn how to define and use publications, see the section on basic publishing.

There are four main things you can configure in a Maven publication:

- A component — via MavenPublication.from(org.gradle.api.component.SoftwareComponent).
- Custom artifacts — via the MavenPublication.artifact(java.lang.Object) method. See MavenArtifact for the available configuration options for custom Maven artifacts.
- Standard metadata like `artifactId`, `groupId` and `version`.
- Other contents of the POM file — via MavenPublication.pom(org.gradle.api.Action).

You can see all of these in action in the complete publishing example. The API documentation for `MavenPublication` has additional code samples.

Identity values in the generated POM

The attributes of the generated POM file will contain identity values derived from the following project properties:

- `groupId` - Project.getGroup()
- `artifactId` - Project.getName()
- `version` - Project.getVersion()

Overriding the default identity values is easy: simply specify the `groupId`, `artifactId` or `version` attributes when configuring the MavenPublication.

Example 674. Customizing the publication identity

build.gradle

```
publishing {
    publications {
        maven(MavenPublication) {
            groupId = 'org.gradle.sample'
            artifactId = 'library'
            version = '1.1'

            from components.java
        }
    }
}
```

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("maven") {
            groupId = "org.gradle.sample"
            artifactId = "library"
            version = "1.1"

            from(components["java"])
        }
    }
}
```

TIP Certain repositories will not be able to handle all supported characters. For example, the `:` character cannot be used as an identifier when publishing to a filesystem-backed repository on Windows.

Maven restricts `groupId` and `artifactId` to a limited character set (`[A-Za-z0-9_\\-.]+`) and Gradle enforces this restriction. For `version` (as well as the artifact `extension` and `classifier` properties), Gradle will handle any valid Unicode character.

The only Unicode values that are explicitly prohibited are `\`, `/` and any ISO control character. Supplied values are validated early in publication.

Customizing the generated POM

The generated POM file can be customized before publishing. For example, when publishing a library to Maven Central you will need to set certain metadata. The Maven Publish Plugin provides

a DSL for that purpose. Please see MavenPom in the DSL Reference for the complete documentation of available properties and methods. The following sample shows how to use the most common ones:

Example 675. Customizing the POM file

build.gradle

```
publishing {
    publications {
        mavenJava(MavenPublication) {
            pom {
                name = 'My Library'
                description = 'A concise description of my library'
                url = 'http://www.example.com/library'
                properties = [
                    myProp: "value",
                    "prop.with.dots": "anotherValue"
                ]
                licenses {
                    license {
                        name = 'The Apache License, Version 2.0'
                        url = 'http://www.apache.org/licenses/LICENSE-
2.0.txt'
                    }
                }
                developers {
                    developer {
                        id = 'johnd'
                        name = 'John Doe'
                        email = 'john.doe@example.com'
                    }
                }
                scm {
                    connection = 'scm:git:git://example.com/my-library.git'
                    developerConnection = 'scm:git:ssh://example.com/my-
library.git'
                    url = 'http://example.com/my-library/'
                }
            }
        }
    }
}
```

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("mavenJava") {
            pom {
                name.set("My Library")
                description.set("A concise description of my library")
                url.set("http://www.example.com/library")
                properties.set(mapOf(
                    "myProp" to "value",
                    "prop.with.dots" to "anotherValue"
                ))
                licenses {
                    license {
                        name.set("The Apache License, Version 2.0")
                        url.set("http://www.apache.org/licenses/LICENSE-
2.0.txt")
                    }
                }
                developers {
                    developer {
                        id.set("johnd")
                        name.set("John Doe")
                        email.set("john.doe@example.com")
                    }
                }
                scm {
                    connection.set("scm:git:git://example.com/my-
library.git")
                    developerConnection.set("scm:git:ssh://example.com/my-
library.git")
                    url.set("http://example.com/my-library/")
                }
            }
        }
    }
}
```

Customizing dependencies versions

Two strategies are supported for publishing dependencies:

Declared versions (default)

This strategy publishes the versions that are defined by the build script author with the dependency declarations in the `dependencies` block. Any other kind of processing, for example through a rule changing the resolved version, will not be taken into account for the publication.

Resolved versions

This strategy publishes the versions that were resolved during the build, possibly by applying resolution rules and automatic conflict resolution. This has the advantage that the published versions correspond to the ones the published artifact was tested against.

Example use cases for resolved versions:

- A project uses dynamic versions for dependencies but prefers exposing the resolved version for a given release to its consumers.
- In combination with dependency locking, you want to publish the locked versions.
- A project leverages the rich versions constraints of Gradle, which have a lossy conversion to Maven. Instead of relying on the conversion, it publishes the resolved versions.

This is done by using the `versionMapping` DSL method which allows to configure the VersionMappingStrategy:

Example 676. Using resolved versions

build.gradle

```
publishing {
    publications {
        mavenJava(MavenPublication) {
            versionMapping {
                usage('java-api') {
                    fromResolutionOf('runtimeClasspath')
                }
                usage('java-runtime') {
                    fromResolutionResult()
                }
            }
        }
    }
}
```

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("mavenJava") {
            versionMapping {
                usage("java-api") {
                    fromResolutionOf("runtimeClasspath")
                }
                usage("java-runtime") {
                    fromResolutionResult()
                }
            }
        }
    }
}
```

In the example above, Gradle will use the versions resolved on the `runtimeClasspath` for dependencies declared in `api`, which are mapped to the `compile` scope of Maven. Gradle will also use the versions resolved on the `runtimeClasspath` for dependencies declared in `implementation`, which are mapped to the `runtime` scope of Maven. `fromResolutionResult()` indicates that Gradle should use the default classpath of a variant and `runtimeClasspath` is the default classpath of `java-runtime`.

Repositories

This plugin provides repositories of type MavenArtifactRepository. To learn how to define and use

repositories for publishing, see the section on basic publishing.

Here's a simple example of defining a publishing repository:

Example 677. Declaring repositories to publish to

build.gradle

```
publishing {
    repositories {
        maven {
            // change to point to your repo, e.g. http://my.org/repo
            url = layout.buildDirectory.dir('repo')
        }
    }
}
```

build.gradle.kts

```
publishing {
    repositories {
        maven {
            // change to point to your repo, e.g. http://my.org/repo
            url = uri(layout.buildDirectory.dir("repo"))
        }
    }
}
```

The two main things you will want to configure are the repository's:

- URL (required)
- Name (optional)

You can define multiple repositories as long as they have unique names within the build script. You may also declare one (and only one) repository without a name. That repository will take on an implicit name of "Maven".

You can also configure any authentication details that are required to connect to the repository. See MavenArtifactRepository for more details.

Snapshot and release repositories

It is a common practice to publish snapshots and releases to different Maven repositories. A simple way to accomplish this is to configure the repository URL based on the project version. The following sample uses one URL for versions that end with "SNAPSHOT" and a different URL for the

rest:

Example 678. Configuring repository URL based on project version

build.gradle

```
publishing {
    repositories {
        maven {
            def releasesRepoUrl = layout.buildDirectory.dir('repos/releases')
            def snapshotsRepoUrl = layout.buildDirectory.dir('
repos/snapshots')
            url = version.endsWith('SNAPSHOT') ? snapshotsRepoUrl :
releasesRepoUrl
        }
    }
}
```

build.gradle.kts

```
publishing {
    repositories {
        maven {
            val releasesRepoUrl = layout.buildDirectory.dir("repos/releases")
            val snapshotsRepoUrl =
layout.buildDirectory.dir("repos/snapshots")
            url = uri(if (version.toString().endsWith("SNAPSHOT"))
snapshotsRepoUrl else releasesRepoUrl)
        }
    }
}
```

Similarly, you can use a project or system property to decide which repository to publish to. The following example uses the release repository if the project property `release` is set, such as when a user runs `gradle -Prelease publish`:

Example 679. Configuring repository URL based on project property

build.gradle

```
publishing {
    repositories {
        maven {
            def releasesRepoUrl = layout.buildDirectory.dir('repos/releases')
            def snapshotsRepoUrl = layout.buildDirectory.dir('
repos/snapshots')
            url = project.hasProperty('release') ? releasesRepoUrl :
snapshotsRepoUrl
        }
    }
}
```

build.gradle.kts

```
publishing {
    repositories {
        maven {
            val releasesRepoUrl = layout.buildDirectory.dir("repos/releases")
            val snapshotsRepoUrl =
layout.buildDirectory.dir("repos/snapshots")
            url = uri(if (project.hasProperty("release")) releasesRepoUrl
else snapshotsRepoUrl)
        }
    }
}
```

Publishing to Maven Local

For integration with a local Maven installation, it is sometimes useful to publish the module into the Maven local repository (typically at *$USER_HOME/.m2/repository*), along with its POM file and other metadata. In Maven parlance, this is referred to as 'installing' the module.

The Maven Publish Plugin makes this easy to do by automatically creating a PublishToMavenLocal task for each MavenPublication in the `publishing.publications` container. The task name follows the pattern of `publishPubNamePublicationToMavenLocal`. Each of these tasks is wired into the `publishToMavenLocal` aggregate task. You do not need to have `mavenLocal()` in your `publishing.repositories` section.

Publishing Maven relocation information

When a project changes the `groupId` or `artifactId` (the *coordinates*) of an artifact it publishes, it is

important to let users know where the new artifact can be found. Maven can help with that through the *relocation* feature. The way this works is that a project publishes an additional artifact under the old coordinates consisting only of a minimal *relocation POM*; that POM file specifies where the new artifact can be found. Maven repository browsers and build tools can then inform the user that the coordinates of an artifact have changed.

For this, a project adds an additional `MavenPublication` specifying a MavenPomRelocation:

Example 680. Specifying a relocation POM

build.gradle

```
publishing {
    publications {
        // ... artifact publications

        // Specify relocation POM
        relocation(MavenPublication) {
            pom {
                // Old artifact coordinates
                groupId = "com.example"
                artifactId = "lib"
                version = "2.0.0"

                distributionManagement {
                    relocation {
                        // New artifact coordinates
                        groupId = "com.new-example"
                        artifactId = "lib"
                        version = "2.0.0"
                        message = "groupId has been changed"
                    }
                }
            }
        }
    }
}
```

build.gradle.kts

```
publishing {
    publications {
        // ... artifact publications

        // Specify relocation POM
        create<MavenPublication>("relocation") {
            pom {
                // Old artifact coordinates
                groupId = "com.example"
                artifactId = "lib"
                version = "2.0.0"

                distributionManagement {
                    relocation {
                        // New artifact coordinates
                        groupId.set("com.new-example")
                        artifactId.set("lib")
                        version.set("2.0.0")
                        message.set("groupId has been changed")
                    }
                }
            }
        }
    }
}
```

Only the property which has changed needs to be specified under `relocation`, that is `artifactId` and / or `groupId`. All other properties are optional.

TIP Specifying the `version` can be useful when the new artifact has a different version, for example because version numbering has started at 1.0.0 again. A custom `message` allows explaining why the artifact coordinates have changed.

The relocation POM should be created for what would be the next version of the old artifact. For example when the artifact coordinates of `com.example:lib:1.0.0` are changed and the artifact with the new coordinates continues version numbering and is published as `com.new-example:lib:2.0.0`, then the relocation POM should specify a relocation from `com.example:lib:2.0.0` to `com.new-example:lib:2.0.0`.

A relocation POM only has to be published once, the build file configuration for it should be removed again once it has been published.

Note that a relocation POM is not suitable for all situations; when an artifact has been split into two or more separate artifacts then a relocation POM might not be helpful.

Retroactively publishing relocation information

It is possible to publish relocation information retroactively after the coordinates of an artifact have changed in the past, and no relocation information was published back then.

The same recommendations as described above apply. To ease migration for users, it is important to pay attention to the `version` specified in the relocation POM. The relocation POM should allow the user to move to the new artifact in one step, and then allow them to update to the latest version in a separate step. For example when for the coordinates of `com.new-example:lib:5.0.0` were changed in version 2.0.0, then ideally the relocation POM should be published for the old coordinates `com.example:lib:2.0.0` relocating to `com.new-example:lib:2.0.0`. The user can then switch from `com.example:lib` to `com.new-example` and then separately update from version 2.0.0 to 5.0.0, handling breaking changes (if any) step by step.

When relocation information is published retroactively, it is not necessary to wait for next regular release of the project, it can be published in the meantime. As mentioned above, the relocation information should then be removed again from the build file once the relocation POM has been published.

Avoiding duplicate dependencies

When only the coordinates of the artifact have changed, but package names of the classes inside the artifact have remained the same, dependency conflicts can occur. A project might (transitively) depend on the old artifact but at the same time also have a dependency on the new artifact which both contain the same classes, potentially with incompatible changes.

To detect such conflicting duplicate dependencies, capabilities can be published as part of the Gradle Module Metadata. For an example using a Java Library project, see declaring additional capabilities for a local component.

Performing a dry run

To verify that relocation information works as expected before publishing it to a remote repository, it can first be published to the local Maven repository. Then a local test Gradle or Maven project can be created which has the relocation artifact as dependency.

Complete example

The following example demonstrates how to sign and publish a Java library including sources, Javadoc, and a customized POM:

Example 681. Publishing a Java library

build.gradle

```
plugins {
    id 'java-library'
    id 'maven-publish'
    id 'signing'
}

group = 'com.example'
version = '1.0'

java {
    withJavadocJar()
    withSourcesJar()
}

publishing {
    publications {
        mavenJava(MavenPublication) {
            artifactId = 'my-library'
            from components.java
            versionMapping {
                usage('java-api') {
                    fromResolutionOf('runtimeClasspath')
                }
                usage('java-runtime') {
                    fromResolutionResult()
                }
            }
            pom {
                name = 'My Library'
                description = 'A concise description of my library'
                url = 'http://www.example.com/library'
                properties = [
                    myProp: "value",
                    "prop.with.dots": "anotherValue"
                ]
                licenses {
                    license {
                        name = 'The Apache License, Version 2.0'
                        url = 'http://www.apache.org/licenses/LICENSE-
2.0.txt'
                    }
                }
                developers {
                    developer {
                        id = 'johnd'
                        name = 'John Doe'
                        email = 'john.doe@example.com'
                    }
```

```
                }
                scm {
                    connection = 'scm:git:git://example.com/my-library.git'
                    developerConnection = 'scm:git:ssh://example.com/my-
library.git'
                    url = 'http://example.com/my-library/'
                }
            }
        }
    }
    repositories {
        maven {
            // change URLs to point to your repos, e.g. http://my.org/repo
            def releasesRepoUrl = layout.buildDirectory.dir('repos/releases')
            def snapshotsRepoUrl = layout.buildDirectory.dir('
repos/snapshots')
            url = version.endsWith('SNAPSHOT') ? snapshotsRepoUrl :
releasesRepoUrl
        }
    }
}

signing {
    sign publishing.publications.mavenJava
}


javadoc {
    if(JavaVersion.current().isJava9Compatible()) {
        options.addBooleanOption('html5', true)
    }
}
```

build.gradle.kts

```
plugins {
    `java-library`
    `maven-publish`
    signing
}

group = "com.example"
version = "1.0"

java {
    withJavadocJar()
    withSourcesJar()
}

publishing {
    publications {
        create<MavenPublication>("mavenJava") {
            artifactId = "my-library"
            from(components["java"])
            versionMapping {
                usage("java-api") {
                    fromResolutionOf("runtimeClasspath")
                }
                usage("java-runtime") {
                    fromResolutionResult()
                }
            }
            pom {
                name.set("My Library")
                description.set("A concise description of my library")
                url.set("http://www.example.com/library")
                properties.set(mapOf(
                    "myProp" to "value",
                    "prop.with.dots" to "anotherValue"
                ))
                licenses {
                    license {
                        name.set("The Apache License, Version 2.0")
                        url.set("http://www.apache.org/licenses/LICENSE-
2.0.txt")
                    }
                }
                developers {
                    developer {
                        id.set("johnd")
                        name.set("John Doe")
                        email.set("john.doe@example.com")
                    }
```

```
                }
                scm {
                    connection.set("scm:git:git://example.com/my-
library.git")
                    developerConnection.set("scm:git:ssh://example.com/my-
library.git")
                    url.set("http://example.com/my-library/")
                }
            }
        }
    }
    repositories {
        maven {
            // change URLs to point to your repos, e.g. http://my.org/repo
            val releasesRepoUrl =
uri(layout.buildDirectory.dir("repos/releases"))
            val snapshotsRepoUrl =
uri(layout.buildDirectory.dir("repos/snapshots"))
            url = if (version.toString().endsWith("SNAPSHOT"))
snapshotsRepoUrl else releasesRepoUrl
        }
    }
}

signing {
    sign(publishing.publications["mavenJava"])
}

tasks.javadoc {
    if (JavaVersion.current().isJava9Compatible) {
        (options as StandardJavadocDocletOptions).addBooleanOption("html5",
true)
    }
}
```

The result is that the following artifacts will be published:

- The POM: `my-library-1.0.pom`
- The primary JAR artifact for the Java component: `my-library-1.0.jar`
- The sources JAR artifact that has been explicitly configured: `my-library-1.0-sources.jar`
- The Javadoc JAR artifact that has been explicitly configured: `my-library-1.0-javadoc.jar`

The Signing Plugin is used to generate a signature file for each artifact. In addition, checksum files will be generated for all artifacts and signature files.

Removal of deferred configuration behavior

Prior to Gradle 5.0, the `publishing {}` block was (by default) implicitly treated as if all the logic inside it was executed after the project is evaluated. This behavior caused quite a bit of confusion and was deprecated in Gradle 4.8, because it was the only block that behaved that way.

You may have some logic inside your publishing block or in a plugin that is depending on the deferred configuration behavior. For instance, the following logic assumes that the subprojects will be evaluated when the artifactId is set:

build.gradle

```
subprojects {
    publishing {
        publications {
            mavenJava(MavenPublication) {
                from components.java
                artifactId = jar.archiveBaseName
            }
        }
    }
}
```

build.gradle.kts

```
subprojects {
    publishing {
        publications {
            create<MavenPublication>("mavenJava") {
                from(components["java"])
                artifactId = tasks.jar.get().archiveBaseName.get()
            }
        }
    }
}
```

This kind of logic must now be wrapped in an `afterEvaluate {}` block.

build.gradle

```
subprojects {
    publishing {
        publications {
            mavenJava(MavenPublication) {
                from components.java
                afterEvaluate {
                    artifactId = jar.archiveBaseName
                }
            }
        }
    }
}
```

build.gradle.kts

```
subprojects {
    publishing {
        publications {
            create<MavenPublication>("mavenJava") {
                from(components["java"])
                afterEvaluate {
                    artifactId = tasks.jar.get().archiveBaseName.get()
                }
            }
        }
    }
}
```

The PMD Plugin

The PMD plugin performs quality checks on your project's Java source files using PMD and generates reports from these checks.

Usage

To use the PMD plugin, include the following in your build script:

Example 682. Using the PMD plugin

build.gradle

```
plugins {
    id 'pmd'
}
```

build.gradle.kts

```
plugins {
    pmd
}
```

The plugin adds a number of tasks to the project that perform the quality checks. You can execute the checks by running `gradle check`.

Note that PMD will run with the same Java version used to run Gradle.

Tasks

The PMD plugin adds the following tasks to the project:

`pmdMain` — **Pmd**

Runs PMD against the production Java source files.

`pmdTest` — **Pmd**

Runs PMD against the test Java source files.

`pmdSourceSet` — **Pmd**

Runs PMD against the given source set's Java source files.

The PMD plugin adds the following dependencies to tasks defined by the Java plugin.

Table 39. PMD plugin - additional task dependencies

Task name	Depends on
`check`	All PMD tasks, including `pmdMain` and `pmdTest`.

Dependency management

The PMD plugin adds the following dependency configurations:

Table 40. PMD plugin - dependency configurations

Name	Meaning
pmd	The PMD libraries to use
pmdAux	The additional libraries that are available for type resolution during analysis. This might be useful if PMD complains about missing classes.

Configuration

build.gradle

```
pmd {
    consoleOutput = true
    toolVersion = "6.21.0"
    rulesMinimumPriority = 5
    ruleSets = ["category/java/errorprone.xml",
"category/java/bestpractices.xml"]
}
```

build.gradle.kts

```
pmd {
    isConsoleOutput = true
    toolVersion = "6.21.0"
    rulesMinimumPriority.set(5)
    ruleSets = listOf("category/java/errorprone.xml",
"category/java/bestpractices.xml")
}
```

See the PmdExtension class in the API documentation.

Parallel analysis

You can configure the number of threads to be used by PMD for running its analysis.

build.gradle

```
pmd {
    threads = 4
}
```

build.gradle.kts

```
pmd {
    threads.set(4)
}
```

NOTE

This configuration is internal to PMD and is not linked to the number of workers used by Gradle. It means that you have to pay attention to the value entered here and make sure it still makes sense in a multi project build. This is because parallel Gradle task execution could result in different PMD tasks from different projects running in parallel. If multiple PMD tasks execute simultaneously in n projects, then up to a maximum of (n * `thread`) PMD threads could run at the same time.

The Scala Plugin

The Scala plugin extends the Java plugin to add support for Scala projects. It can deal with Scala code, mixed Scala and Java code, and even pure Java code (although we don't necessarily recommend to use it for the latter). The plugin supports *joint compilation*, which allows you to freely mix and match Scala and Java code, with dependencies in both directions. For example, a Scala class can extend a Java class that in turn extends a Scala class. This makes it possible to use the best language for the job, and to rewrite any class in the other language if needed.

Note that if you want to benefit from the API / implementation separation, you can also apply the `java-library` plugin to your Scala project.

Usage

To use the Scala plugin, include the following in your build script:

Example 683. Using the Scala plugin

build.gradle

```
plugins {
    id 'scala'
}
```

build.gradle.kts

```
plugins {
    scala
}
```

Tasks

The Scala plugin adds the following tasks to the project. Information about altering the dependencies to Java compile tasks are found here.

`compileScala` — **ScalaCompile**

Depends on: `compileJava`

Compiles production Scala source files.

`compileTestScala` — **ScalaCompile**

Depends on: `compileTestJava`

Compiles test Scala source files.

`compileSourceSetScala` — **ScalaCompile**

Depends on: `compileSourceSetJava`

Compiles the given source set's Scala source files.

`scaladoc` — **ScalaDoc**

Generates API documentation for the production Scala source files.

The `ScalaCompile` and `ScalaDoc` tasks can also leverage the Java toolchain support.

The Scala plugin adds the following dependencies to tasks added by the Java plugin.

Table 41. Scala plugin - additional task dependencies

Task name	Depends on
`classes`	`compileScala`
`testClasses`	`compileTestScala`
`sourceSetClasses`	`compileSourceSetScala`

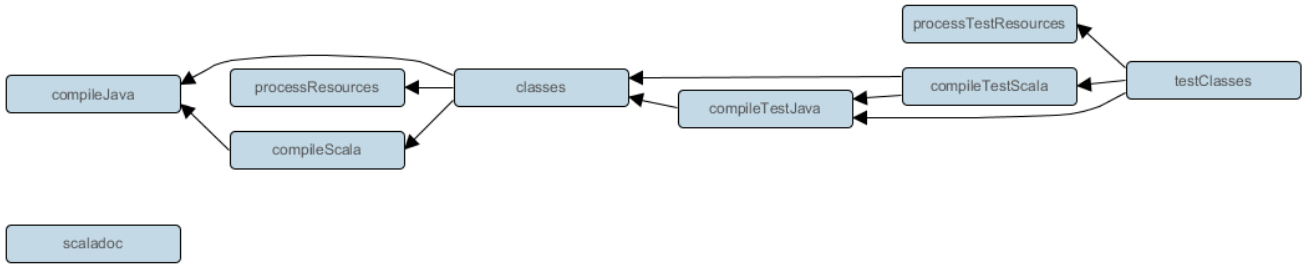


Figure 37. Scala plugin - tasks

Project layout

The Scala plugin assumes the project layout shown below. All the Scala source directories can contain Scala *and* Java code. The Java source directories may only contain Java source code. None of these directories need to exist or have anything in them; the Scala plugin will simply compile whatever it finds.

`src/main/java`
: Production Java source.

`src/main/resources`
: Production resources, such as XML and properties files.

`src/main/scala`
: Production Scala source. May also contain Java source files for joint compilation.

`src/test/java`
: Test Java source.

`src/test/resources`
: Test resources.

`src/test/scala`
: Test Scala source. May also contain Java source files for joint compilation.

`src/sourceSet/java`
: Java source for the source set named *sourceSet*.

`src/sourceSet/resources`
: Resources for the source set named *sourceSet*.

`src/sourceSet/scala`

Scala source files for the given source set. May also contain Java source files for joint compilation.

Changing the project layout

Just like the Java plugin, the Scala plugin allows you to configure custom locations for Scala production and test source files.

Example 684. Custom Scala source layout

build.gradle

```
sourceSets {
    main {
        scala {
            srcDirs = ['src/scala']
        }
    }
    test {
        scala {
            srcDirs = ['test/scala']
        }
    }
}
```

build.gradle.kts

```
sourceSets {
    main {
        withConvention(ScalaSourceSet::class) {
            scala {
                setSrcDirs(listOf("src/scala"))
            }
        }
    }
    test {
        withConvention(ScalaSourceSet::class) {
            scala {
                setSrcDirs(listOf("test/scala"))
            }
        }
    }
}
```

Dependency management

Scala projects need to declare a `scala-library` dependency. This dependency will then be used on compile and runtime class paths. It will also be used to get hold of the Scala compiler and Scaladoc tool, respectively.[18]

If Scala is used for production code, the `scala-library` dependency should be added to the `implementation` configuration:

Example 685. Declaring a Scala dependency for production code

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.scala-lang:scala-library:2.11.12'
    testImplementation 'org.scalatest:scalatest_2.11:3.0.0'
    testImplementation 'junit:junit:4.13'
}
```

build.gradle.kts

```
repositories {
    mavenCentral()
}

dependencies {
    implementation("org.scala-lang:scala-library:2.11.12")
    testImplementation("org.scalatest:scalatest_2.11:3.0.0")
    testImplementation("junit:junit:4.13")
}
```

If you want to use Scala 3 instead of the `scala-library` dependency you should add the `scala3-library_3` dependency:

Example 686. Declaring a Scala 3 dependency for production code

build.gradle

```
plugins {
    id 'scala'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.scala-lang:scala3-library_3:3.0.1'
    testImplementation 'org.scalatest:scalatest_3:3.2.9'
    testImplementation 'junit:junit:4.13'
}

dependencies {
    implementation 'commons-collections:commons-collections:3.2.2'
}
```

build.gradle.kts

```
plugins {
    scala
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.scala-lang:scala3-library_3:3.0.1")
    testImplementation("org.scalatest:scalatest_3:3.2.9")
    testImplementation("junit:junit:4.13")
}

dependencies {
    implementation("commons-collections:commons-collections:3.2.2")
}
```

If Scala is only used for test code, the `scala-library` dependency should be added to the `testImplementation` configuration:

Example 687. Declaring a Scala dependency for test code

build.gradle

```
dependencies {
    testImplementation 'org.scala-lang:scala-library:2.11.1'
}
```

build.gradle.kts

```
dependencies {
    testImplementation("org.scala-lang:scala-library:2.11.1")
}
```

Automatic configuration of scalaClasspath

The `ScalaCompile` and `ScalaDoc` tasks consume Scala code in two ways: on their `classpath`, and on their `scalaClasspath`. The former is used to locate classes referenced by the source code, and will typically contain `scala-library` along with other libraries. The latter is used to load and execute the Scala compiler and Scaladoc tool, respectively, and should only contain the `scala-compiler` library and its dependencies.

Unless a task's `scalaClasspath` is configured explicitly, the Scala (base) plugin will try to infer it from the task's `classpath`. This is done as follows:

- If a `scala-library` jar is found on `classpath`, and the project has at least one repository declared, a corresponding `scala-compiler` repository dependency will be added to `scalaClasspath`.
- Otherwise, execution of the task will fail with a message saying that `scalaClasspath` could not be inferred.

Configuring the Zinc compiler

The Scala plugin uses a configuration named `zinc` to resolve the Zinc compiler and its dependencies. Gradle will provide a default version of Zinc, but if you need to use a particular Zinc version, you can change it. Gradle supports version 1.6.0 of Zinc and above.

Example 688. Declaring a version of the Zinc compiler to use

build.gradle

```
scala {
    zincVersion = "1.6.1"
}
```

build.gradle.kts

```
scala {
    zincVersion.set("1.6.1")
}
```

The Zinc compiler itself needs a compatible version of `scala-library` that may be different from the version required by your application. Gradle takes care of specifying a compatible version of `scala-library` for you.

You can diagnose problems with the version of the Zinc compiler selected by running dependencyInsight for the `zinc` configuration.

Table 42. Zinc compatibility table

Gradle version	Supported Zinc versions	Zinc coordinates	Required Scala version	Supported Scala compilation version
7.5 and newer	SBT Zinc. Versions 1.6.0 and above.	`org.scala-sbt:zinc_2.13`	Scala `2.13.x` is required for *running* Zinc.	Scala `2.10.x` through `3.x` can be compiled.
6.0 to 7.5	SBT Zinc. Versions 1.2.0 and above.	`org.scala-sbt:zinc_2.12`	Scala `2.12.x` is required for *running* Zinc.	Scala `2.10.x` through `2.13.x` can be compiled.
1.x through 5.x	**Deprecated** Typesafe Zinc compiler. Versions 0.3.0 and above, except for 0.3.2 through 0.3.5.2.	`com.typesafe.zinc:zinc`	Scala `2.10.x` is required for *running* Zinc.	Scala `2.9.x` through `2.12.x` can be compiled.

Adding plugins to the Scala compiler

The Scala plugin adds a configuration named `scalaCompilerPlugins` which is used to declare and resolve optional compiler plugins.

Example 689. Adding a dependency on a Scala compiler plugin

build.gradle

```
dependencies {
    implementation "org.scala-lang:scala-library:2.13.1"
    scalaCompilerPlugins "org.typelevel:kind-projector_2.13.1:0.11.0"
}
```

build.gradle.kts

```
dependencies {
    implementation("org.scala-lang:scala-library:2.13.1")
    scalaCompilerPlugins("org.typelevel:kind-projector_2.13.1:0.11.0")
}
```

Convention properties

The Scala plugin does not add any convention properties to the project.

Source set properties

The Scala plugin adds the following extensions to each source set in the project. You can use these in your build script as though they were properties of the source set object.

`scala` — **SourceDirectorySet (read-only)**

The Scala source files of this source set. Contains all `.scala` and `.java` files found in the Scala source directories, and excludes all other types of files. *Default value:* non-null.

`scala.srcDirs` — `Set<File>`

The source directories containing the Scala source files of this source set. May also contain Java source files for joint compilation. Can set using anything described in Understanding implicit conversion to file collections. *Default value:* `[projectDir/src/name/scala]`.

`allScala` — **FileTree (read-only)**

All Scala source files of this source set. Contains only the `.scala` files found in the Scala source directories. *Default value:* non-null.

These extensions are backed by an object of type ScalaSourceSet.

The Scala plugin also modifies some source set properties:

Table 43. Scala plugin - source set properties

Property name	Change
`allJava`	Adds all `.java` files found in the Scala source directories.
`allSource`	Adds all source files found in the Scala source directories.

Compiling in external process

Scala compilation takes place in an external process.

Memory settings for the external process default to the defaults of the JVM. To adjust memory settings, configure the `scalaCompileOptions.forkOptions` property as needed:

Example 690. Adjusting memory settings

build.gradle

```
tasks.withType(ScalaCompile) {
    scalaCompileOptions.forkOptions.with {
        memoryMaximumSize = '1g'
        jvmArgs = ['-XX:MaxMetaspaceSize=512m']
    }
}
```

build.gradle.kts

```
tasks.withType<ScalaCompile>().configureEach {
    scalaCompileOptions.forkOptions.apply {
        memoryMaximumSize = "1g"
        jvmArgs = listOf("-XX:MaxMetaspaceSize=512m")
    }
}
```

Incremental compilation

By compiling only classes whose source code has changed since the previous compilation, and classes affected by these changes, incremental compilation can significantly reduce Scala compilation time. It is particularly effective when frequently compiling small code increments, as is often done at development time.

The Scala plugin defaults to incremental compilation by integrating with Zinc, a standalone version of sbt's incremental Scala compiler. If you want to disable the incremental compilation, set `force = true` in your build file:

Example 691. Forcing all code to be compiled

build.gradle

```
tasks.withType(ScalaCompile) {
    scalaCompileOptions.with {
        force = true
    }
}
```

build.gradle.kts

```
tasks.withType<ScalaCompile>().configureEach {
    scalaCompileOptions.apply {
        isForce = true
    }
}
```

Note: This will only cause all classes to be recompiled if at least one input source file has changed. If there are no changes to the source files, the `compileScala` task will still be considered `UP-TO-DATE` as usual.

The Zinc-based Scala Compiler supports joint compilation of Java and Scala code. By default, all Java and Scala code under `src/main/scala` will participate in joint compilation. Even Java code will be compiled incrementally.

Incremental compilation requires dependency analysis of the source code. The results of this analysis are stored in the file designated by `scalaCompileOptions.incrementalOptions.analysisFile` (which has a sensible default). In a multi-project build, analysis files are passed on to downstream `ScalaCompile` tasks to enable incremental compilation across project boundaries. For `ScalaCompile` tasks added by the Scala plugin, no configuration is necessary to make this work. For other `ScalaCompile` tasks that you might add, the property `scalaCompileOptions.incrementalOptions.publishedCode` needs to be configured to point to the classes folder or Jar archive by which the code is passed on to compile class paths of downstream `ScalaCompile` tasks. Note that if `publishedCode` is not set correctly, downstream tasks may not recompile code affected by upstream changes, leading to incorrect compilation results.

Note that Zinc's Nailgun based daemon mode is not supported. Instead, we plan to enhance Gradle's own compiler daemon to stay alive across Gradle invocations, reusing the same Scala compiler. This is expected to yield another significant speedup for Scala compilation.

Eclipse Integration

When the Eclipse plugin encounters a Scala project, it adds additional configuration to make the project work with Scala IDE out of the box. Specifically, the plugin adds a Scala nature and

dependency container.

IntelliJ IDEA Integration

When the IDEA plugin encounters a Scala project, it adds additional configuration to make the project work with IDEA out of the box. Specifically, the plugin adds a Scala SDK (IntelliJ IDEA 14+) and a Scala compiler library that matches the Scala version on the project's class path. The Scala plugin is backwards compatible with earlier versions of IntelliJ IDEA and it is possible to add a Scala facet instead of the default Scala SDK by configuring `targetVersion` on IdeaModel.

Example 692. Explicitly specify a target IntelliJ IDEA version

build.gradle

```
idea {
    targetVersion = '13'
}
```

build.gradle.kts

```
idea {
    targetVersion = "13"
}
```

The Signing Plugin

The Signing Plugin adds the ability to digitally sign built files and artifacts. These digital signatures can then be used to prove who built the artifact the signature is attached to as well as other information such as when the signature was generated.

The Signing Plugin currently only provides support for generating OpenPGP signatures (which is the signature format required for publication to the Maven Central Repository).

Usage

To use the Signing Plugin, include the following in your build script:

Example 693. Using the Signing Plugin

build.gradle

```
plugins {
    id 'signing'
}
```

build.gradle.kts

```
plugins {
    signing
}
```

Signatory credentials

In order to create OpenPGP signatures, you will need a key pair (instructions on creating a key pair using the GnuPG tools can be found in the GnuPG HOWTOs). You need to provide the Signing Plugin with your key information, which means three things:

- The public key ID (The last 8 symbols of the keyId. You can use `gpg -K` to get it).
- The absolute path to the secret key ring file containing your private key. (Since gpg 2.1, you need to export the keys with command `gpg --keyring secring.gpg --export-secret-keys > ~/.gnupg/secring.gpg`).
- The passphrase used to protect your private key.

These items must be supplied as the values of the `signing.keyId`, `signing.secretKeyRingFile`, and `signing.password` properties, respectively.

NOTE	Given the personal and private nature of these values, a good practice is to store them in the `gradle.properties` file in the user's Gradle home directory (described in System properties) instead of in the project directory itself.

```
signing.keyId=24875D73
signing.password=secret
signing.secretKeyRingFile=/Users/me/.gnupg/secring.gpg
```

If specifying this information (especially `signing.password`) in the user `gradle.properties` file is not feasible for your environment, you can supply the information via the command line:

```
> gradle sign -Psigning.secretKeyRingFile=/Users/me/.gnupg/secring.gpg
-Psigning.password=secret -Psigning.keyId=24875D73
```

Using in-memory ascii-armored keys

In some setups it is easier to use environment variables to pass the secret key and password used for signing. For instance, when using a CI server to sign artifacts, securely providing the keyring file is often troublesome. On the other hand, most CI servers provide means to securely store environment variables and provide them to builds. Using the following setup, you can pass the secret key (in ascii-armored format) and the password using the `ORG_GRADLE_PROJECT_signingKey` and `ORG_GRADLE_PROJECT_signingPassword` environment variables, respectively:

build.gradle

```
signing {
    def signingKey = findProperty("signingKey")
    def signingPassword = findProperty("signingPassword")
    useInMemoryPgpKeys(signingKey, signingPassword)
    sign stuffZip
}
```

build.gradle.kts

```
signing {
    val signingKey: String? by project
    val signingPassword: String? by project
    useInMemoryPgpKeys(signingKey, signingPassword)
    sign(tasks["stuffZip"])
}
```

Using in-memory ascii-armored OpenPGP subkeys

To prevent sharing of the master key and to keep it secure it is also possible to use in-memory ascii-armored subkeys. The main difference between using in-memory ascii-armored keys and subkeys is that it is necessary to specify key identifier as well. Using the following setup, you can pass the key identifier, secret key (in ascii-armored format) and the password using the `ORG_GRADLE_PROJECT_signingKeyId`, `ORG_GRADLE_PROJECT_signingKey` and `ORG_GRADLE_PROJECT_signingPassword` environment variables respectively:

build.gradle

```
signing {
    def signingKeyId = findProperty("signingKeyId")
    def signingKey = findProperty("signingKey")
    def signingPassword = findProperty("signingPassword")
    useInMemoryPgpKeys(signingKeyId, signingKey, signingPassword)
    sign stuffZip
}
```

build.gradle.kts

```
signing {
    val signingKeyId: String? by project
    val signingKey: String? by project
    val signingPassword: String? by project
    useInMemoryPgpKeys(signingKeyId, signingKey, signingPassword)
    sign(tasks["stuffZip"])
}
```

Using OpenPGP subkeys

OpenPGP supports subkeys, which are like the normal keys, except they're bound to a master key pair. One feature of OpenPGP subkeys is that they can be revoked independently of the master keys which makes key management easier. A practical case study of how subkeys can be leveraged in software development can be read on the Debian wiki.

The Signing Plugin supports OpenPGP subkeys out of the box. Just specify a subkey ID as the value in the `signing.keyId` property.

Using gpg-agent

By default the Signing Plugin uses a Java-based implementation of PGP for signing. This implementation cannot use the gpg-agent program for managing private keys, though. If you want to use the gpg-agent, you can change the signatory implementation used by the Signing Plugin:

Example 694. Sign with GnuPG

build.gradle

```
signing {
    useGpgCmd()
    sign configurations.archives
}
```

build.gradle.kts

```
signing {
    useGpgCmd()
    sign(configurations.archives.get())
}
```

This tells the Signing Plugin to use the `GnupgSignatory` instead of the default PgpSignatory. The `GnupgSignatory` relies on the gpg2 program to sign the artifacts. Of course, this requires that GnuPG is installed.

Without any further configuration the `gpg` (on Windows: `gpg.exe`) executable found on the `PATH` will be used. The password is supplied by the `gpg-agent` and the default key is used for signing.

Gnupg signatory configuration

The `GnupgSignatory` supports a number of configuration options for controlling how gpg is invoked. These are typically set in gradle.properties:

Example: Configure the GnupgSignatory

gradle.properties

```
signing.gnupg.executable=gpg
signing.gnupg.useLegacyGpg=true
signing.gnupg.homeDir=gnupg-home
signing.gnupg.optionsFile=gnupg-home/gpg.conf
signing.gnupg.keyName=24875D73
signing.gnupg.passphrase=gradle
```

`signing.gnupg.executable`

The gpg executable that is invoked for signing. The default value of this property depends on `useLegacyGpg`. If that is `true` then the default value of executable is "gpg" otherwise it is "gpg2".

`signing.gnupg.useLegacyGpg`

Must be `true` if GnuPG version 1 is used and `false` otherwise. The default value of the property is `false`.

`signing.gnupg.homeDir`

Sets the home directory for GnuPG. If not given the default home directory of GnuPG is used.

`signing.gnupg.optionsFile`

Sets a custom options file for GnuPG. If not given GnuPG's default configuration file is used.

`signing.gnupg.keyName`

The id of the key that should be used for signing. If not given then the default key configured in GnuPG will be used.

`signing.gnupg.passphrase`

The passphrase for unlocking the secret key. If not given then the gpg-agent program is used for getting the passphrase.

All configuration properties are optional.

Specifying what to sign

As well as configuring how things are to be signed (i.e. the signatory configuration), you must also specify what is to be signed. The Signing Plugin provides a DSL that allows you to specify the tasks and/or configurations that should be signed.

Signing Publications

When publishing artifacts, you often want to sign them so the consumer of your artifacts can verify their signature. For example, the Java plugin defines a component that you can use to define a publication to a Maven (or Ivy) repository using the Maven Publish Plugin (or the Ivy Publish Plugin, respectively). Using the Signing DSL, you can specify that all of the artifacts of this publication should be signed.

Example 695. Signing a publication

build.gradle

```
signing {
    sign publishing.publications.mavenJava
}
```

build.gradle.kts

```
signing {
    sign(publishing.publications["mavenJava"])
}
```

This will create a task (of type Sign) in your project named `signMavenJavaPublication` that will build all artifacts that are part of the publication (if needed) and then generate signatures for them. The signature files will be placed alongside the artifacts being signed.

Example: Signing a publication output

Output of `gradle signMavenJavaPublication`

```
> gradle signMavenJavaPublication
> Task :compileJava
> Task :processResources
> Task :classes
> Task :jar
> Task :javadoc
> Task :javadocJar
> Task :sourcesJar
> Task :generateMetadataFileForMavenJavaPublication
> Task :generatePomFileForMavenJavaPublication
> Task :signMavenJavaPublication

BUILD SUCCESSFUL in 0s
9 actionable tasks: 9 executed
```

In addition, the above DSL allows to `sign` multiple comma-separated publications. Alternatively, you may specify `publishing.publications` to sign all publications, or use `publishing.publications.matching { ⋯ }` to sign all publications that match the specified predicate.

Signing Configurations

It is common to want to sign the artifacts of a configuration. For example, the Java plugin configures a jar to build and this jar artifact is added to the `archives` configuration. Using the

Signing DSL, you can specify that all of the artifacts of this configuration should be signed.

Example 696. Signing a configuration

build.gradle

```
signing {
    sign configurations.archives
}
```

build.gradle.kts

```
signing {
    sign(configurations.archives.get())
}
```

This will create a task (of type Sign) in your project named `signArchives`, that will build any `archives` artifacts (if needed) and then generate signatures for them. The signature files will be placed alongside the artifacts being signed.

Example: Signing a configuration output

Output of `gradle signArchives`

```
> gradle signArchives
> Task :compileJava
> Task :processResources
> Task :classes
> Task :jar
> Task :signArchives

BUILD SUCCESSFUL in 0s
4 actionable tasks: 4 executed
```

Signing Task Output

In some cases the artifact that you need to sign may not be part of a configuration. In this case you can directly sign the task that produces the artifact to sign.

Example 697. Signing a task output

build.gradle

```
tasks.register('stuffZip', Zip) {
    archiveBaseName = 'stuff'
    from 'src/stuff'
}

signing {
    sign stuffZip
}
```

build.gradle.kts

```
tasks.register<Zip>("stuffZip") {
    archiveBaseName.set("stuff")
    from("src/stuff")
}

signing {
    sign(tasks["stuffZip"])
}
```

This will create a task (of type Sign) in your project named `signStuffZip`, that will build the input task's archive (if needed) and then sign it. The signature file will be placed alongside the artifact being signed.

Example: Signing a task output

Output of `gradle signStuffZip`

```
> gradle signStuffZip
> Task :stuffZip
> Task :signStuffZip

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

For a task to be *signable*, it must produce an archive of some type, i.e. it must extend AbstractArchiveTask. Tasks that do this are the Tar, Zip, Jar, War and Ear tasks.

Conditional Signing

A common usage pattern is to require the signing of build artifacts only under certain conditions.

For example, you may not need to sign artifacts for non-release versions. To achieve this, you can specify the condition as an argument of the `required()` method.

Example 698. Specifying when signing is required

build.gradle

```
version = '1.0-SNAPSHOT'
ext.isReleaseVersion = !version.endsWith("SNAPSHOT")

signing {
    required { isReleaseVersion && gradle.taskGraph.hasTask("publish") }
    sign publishing.publications.main
}
```

build.gradle.kts

```
version = "1.0-SNAPSHOT"
extra["isReleaseVersion"] = !version.toString().endsWith("SNAPSHOT")

signing {
    setRequired({
        (project.extra["isReleaseVersion"] as Boolean) &&
gradle.taskGraph.hasTask("publish")
    })
    sign(publishing.publications["main"])
}
```

In this example, we only want to require signing if we are building a release version and we are going to publish it. Because we are inspecting the task graph to determine if we are going to be publishing, we must set the `signing.required` property to a closure to defer the evaluation. See SigningExtension.setRequired(java.lang.Object) for more information.

If the `required` condition does not hold true, artifacts will only be signed if signatory credentials are configured. Alternatively, you may want to skip signing entirely whether or not signatory credentials are available. If so, you can configure the Sign tasks to be skipped, for example by attaching a predicate using the `onlyIf()` method shown in the following example:

Example 699. Specifying when signing is skipped

build.gradle

```
tasks.withType(Sign) {
    onlyIf("isReleaseVersion is set") { isReleaseVersion }
}
```

build.gradle.kts

```
tasks.withType<Sign>().configureEach {
    onlyIf("isReleaseVersion is set") { project.extra["isReleaseVersion"] as
Boolean }
}
```

Publishing the signatures

When signing publications, the resultant signature artifacts are automatically added to the corresponding publication. Thus, when publishing to a repository, e.g. by executing the `publish` task, your signatures will be distributed along with the other artifacts without any additional configuration.

When signing configurations and tasks, the resultant signature artifacts are automatically added to the `signatures` and `archives` dependency configurations.

The War Plugin

The War plugin extends the Java plugin to add support for assembling web application WAR files. It disables the default JAR archive generation of the Java plugin and adds a default WAR archive task.

Usage

To use the War plugin, include the following in your build script:

Example 700. Using the War plugin

build.gradle

```
plugins {
    id 'war'
}
```

build.gradle.kts

```
plugins {
    war
}
```

Project layout

In addition to the standard Java project layout, the War Plugin adds:

`src/main/webapp`
: Web application sources

Tasks

The War plugin adds and modifies the following tasks:

`war` — **War**
: *Depends on*: `compile`

  Assembles the application WAR file.

`assemble` - ***lifecycle task***
: *Depends on*: `war`

The War plugin adds the following dependencies to tasks added by the Java plugin;

Figure 38. War plugin - tasks

Dependency management

The War plugin adds two dependency configurations:

`providedCompile`

This configuration should be used for dependencies required at compilation but which are provided by the environment in which the WAR is deployed. Dependencies declared here are thus visible to the `main` and `test` compilation classpaths.

`providedRuntime`

This configuration should be used for dependencies required at runtime but which are provided by the environment in which the WAR is deployed. Dependencies declared here are only visible to the `main` and `test` runtime classpaths.

NOTE	It is important to note that these `provided` configurations work transitively. Let's say you add `commons-httpclient:commons-httpclient:3.0` to any of the provided configurations. This dependency has a dependency on `commons-codec`. Because this is a "provided" configuration, this means that neither of these dependencies will be added to your WAR, even if the `commons-codec` library is an explicit dependency of your `implementation` configuration. If you don't want this transitive behavior, simply declare your `provided` dependencies like `commons-httpclient:commons-httpclient:3.0@jar`.

Publishing

`components.web`

A SoftwareComponent for publishing the production WAR created by the `war` task.

Convention properties (deprecated)

`webAppDirName` — `String`

Default value: `src/main/webapp`

The name of the web application source directory, relative to the project directory.

`webAppDir` — **(read-only)** `File`

Default value: `$webAppDirName`, e.g. *src/main/webapp*

The path to the web application source directory.

These properties are provided by a WarPluginConvention object.

Configuring war tasks via convention properties is **deprecated**. If you need to set default values the `war` task then configure the task directly. If you want to configure all tasks of type `War` in the project then use tasks.withType(War.class).configureEach(...).

War

The default behavior of the `War` task is to copy the content of `src/main/webapp` to the root of the archive. Your `webapp` directory may of course contain a `WEB-INF` sub-directory, which may contain a `web.xml` file. Your compiled classes are compiled to `WEB-INF/classes`. All the dependencies of the `runtime` [19] configuration are copied to `WEB-INF/lib`.

The War class in the API documentation has additional useful information.

Customizing

Here is an example with the most important customization options:

Example 701. Customization of war plugin

build.gradle

```
configurations {
   moreLibs
}

repositories {
   flatDir { dirs "lib" }
   mavenCentral()
}

dependencies {
    implementation module(":compile:1.0") {
        dependency ":compile-transitive-1.0@jar"
        dependency ":providedCompile-transitive:1.0@jar"
    }
    providedCompile "javax.servlet:servlet-api:2.5"
    providedCompile module(":providedCompile:1.0") {
        dependency ":providedCompile-transitive:1.0@jar"
    }
    runtimeOnly ":runtime:1.0"
    providedRuntime ":providedRuntime:1.0@jar"
    testImplementation "junit:junit:4.13"
    moreLibs ":otherLib:1.0"
}

war {
    webAppDirectory = file('src/main/webapp')
    from 'src/rootContent' // adds a file-set to the root of the archive
    webInf { from 'src/additionalWebInf' } // adds a file-set to the WEB-INF
dir.
    classpath fileTree('additionalLibs') // adds a file-set to the WEB-
INF/lib dir.
    classpath configurations.moreLibs // adds a configuration to the WEB-
INF/lib dir.
    webXml = file('src/someWeb.xml') // copies a file to WEB-INF/web.xml
}
```

build.gradle.kts

```
val moreLibs = configurations.create("moreLibs")

repositories {
    flatDir { dir("lib") }
    mavenCentral()
}

dependencies {
    implementation(module(":compile:1.0") {
        dependency(":compile-transitive-1.0@jar")
        dependency( ":providedCompile-transitive:1.0@jar")
    })
    providedCompile("javax.servlet:servlet-api:2.5")
    providedCompile(module(":providedCompile:1.0") {
        dependency(":providedCompile-transitive:1.0@jar")
    })
    runtimeOnly(":runtime:1.0")
    providedRuntime(":providedRuntime:1.0@jar")
    testImplementation("junit:junit:4.13")
    moreLibs(":otherLib:1.0")
}

tasks.war {
    webAppDirectory.set(file("src/main/webapp"))
    from("src/rootContent") // adds a file-set to the root of the archive
    webInf { from("src/additionalWebInf") } // adds a file-set to the WEB-INF
dir.
    classpath(fileTree("additionalLibs")) // adds a file-set to the WEB-
INF/lib dir.
    classpath(moreLibs) // adds a configuration to the WEB-INF/lib dir.
    webXml = file("src/someWeb.xml") // copies a file to WEB-INF/web.xml
}
```

Of course one can configure the different file-sets with a closure to define excludes and includes.

[17] Gradle uses the same conventions as introduced by Russel Winder's Gant tool.

[18] See Automatic configuration of Scala classpath.

[19] The `runtime` configuration extends the `compile` configuration.

License Information

License Information

Gradle Documentation

Copyright © 2007-2019 Gradle, Inc.

Gradle build tool source code is open-source and licensed under the Apache License 2.0.

Gradle user manual and DSL reference manual are licensed under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Gradle Build Scan Plugin

Use of the build scan plugin is subject to Gradle's Terms of Service.