Gradle User Manual

Version 5.0-rc-2

Table of Contents

About Gradle 1
- Overview 1
- What is Gradle? 3

Getting Started 8
- Getting Started 8
- Installing Gradle 10
- Gradle Kotlin DSL Primer 13
- Upgrading your build from Gradle 4.x to 5.0 48

Using Gradle Builds 69
- Command-Line Interface 69
- Build Environment 84
- Directory Layout 93
- The Gradle Daemon 95
- Initialization Scripts 101
- Executing Multi-Project Builds 109
- The Gradle Wrapper 112
- Troubleshooting 118

Authoring Gradle Builds 124
- The Feature Lifecycle 124
- Authoring Maintainable Builds 125
- Organizing Gradle Projects 136
- Build Cache 145
- Build Init Plugin 161
- Build Lifecycle 165
- Build Script Basics 180
- Composite builds 199
- Authoring Multi-Project Builds 208
- Authoring Tasks 256
- Logging 321
- Standard Gradle plugins 329
- Testing Build Logic with TestKit 332
- Using Gradle Plugins 351
- Working With Files 369
- Writing Build Scripts 430
- Writing Custom Task Classes 445
- The Base Plugin 475

Dependency Management 478
- Introduction to Dependency Management 478

Dependency Management Terminology . . . 481
Dependency Types . . . 483
Repository Types . . . 489
Declaring Dependencies . . . 515
Declaring Repositories . . . 529
Inspecting Dependencies . . . 532
Managing Dependency Configurations . . . 539
Managing Transitive Dependencies . . . 543
Dependency Locking . . . 563
Troubleshooting Dependency Resolution . . . 569
Customizing Dependency Resolution Behavior . . . 573
The Dependency Cache . . . 600
Working with Dependencies . . . 602
Attribute based matching in dependency resolution . . . 609
Publishing Artifacts . . . 619
Publishing . . . 619
Maven Publish Plugin . . . 635
Ivy Publish Plugin . . . 649
Legacy publishing . . . 659
Maven Plugin . . . 665
The Signing Plugin . . . 679
The Distribution Plugin . . . 690
Native Projects . . . 697
Building native software . . . 697
Software model concepts . . . 736
Rule based model configuration . . . 736
Implementing model rules in a plugin . . . 760
Extending the software model . . . 760
Groovy Projects . . . 772
Groovy Quickstart . . . 772
The Groovy Plugin . . . 774
The CodeNarc Plugin . . . 785
Java Projects . . . 788
Java Quickstart . . . 788
Building Java & JVM projects . . . 802
Testing in Java & JVM projects . . . 826
The Java Plugin . . . 849
The Java Library Plugin . . . 871
The Java Library Distribution Plugin . . . 879
Dependency Management for Java Projects . . . 881
Using Ant from Gradle . . . 887

The ANTLR Plugin . . . 905
The Application Plugin . . . 908
The Checkstyle Plugin . . . 914
The FindBugs Plugin . . . 917
The JaCoCo Plugin . . . 920
The JDepend Plugin . . . 928
The OSGi Plugin . . . 930
The PMD Plugin . . . 933
Java Web Projects . . . 935
The War Plugin . . . 935
The Ear Plugin . . . 938
Building Play applications . . . 943
Scala Projects . . . 958
The Scala Plugin . . . 958
Integrating Gradle . . . 971
The Eclipse Plugins . . . 971
The IDEA Plugin . . . 979
Embedding Gradle using the Tooling API . . . 986
Extending Gradle . . . 990
Writing Custom Plugins . . . 990
Gradle Plugin Development Plugin . . . 1012
Lazy Configuration . . . 1014
Reference . . . 1035
Gradle Plugin Reference . . . 1035
Gradle & Third-party Tools . . . 1037
Licenses . . . 1039
License Information . . . 1040
Gradle Documentation . . . 1040
Gradle Build Scan Plugin . . . 1040

About Gradle

Overview

Features

Here is a list of some of Gradle's features.

Declarative builds and build-by-convention

At the heart of Gradle lies a rich extensible Domain Specific Language (DSL) based on Groovy. Gradle pushes declarative builds to the next level by providing declarative language elements that you can assemble as you like. Those elements also provide build-by-convention support for Java, Groovy, OSGi, Web and Scala projects. Even more, this declarative language is extensible. Add your own new language elements or enhance the existing ones, thus providing concise, maintainable and comprehensible builds.

Language for dependency based programming

The declarative language lies on top of a general purpose task graph, which you can fully leverage in your builds. It provides utmost flexibility to adapt Gradle to your unique needs.

Structure your build

The suppleness and richness of Gradle finally allows you to apply common design principles to your build. For example, it is very easy to compose your build from reusable pieces of build logic. Inline stuff where unnecessary indirections would be inappropriate. Don't be forced to tear apart what belongs together (e.g. in your project hierarchy). Avoid smells like shotgun changes or divergent change that turn your build into a maintenance nightmare. At last you can create a well structured, easily maintained, comprehensible build.

Deep API

From being a pleasure to be used embedded to its many hooks over the whole lifecycle of build execution, Gradle allows you to monitor and customize its configuration and execution behavior to its very core.

Gradle scales

Gradle scales very well. It significantly increases your productivity, from simple single project builds up to huge enterprise multi-project builds. This is true for structuring the build. With the state-of-art incremental build function, this is also true for tackling the performance pain many large enterprise builds suffer from.

Multi-project builds

Gradle's support for multi-project build is outstanding. Project dependencies are first class citizens. We allow you to model the project relationships in a multi-project build as they really are for your problem domain. Gradle follows your layout not vice versa.

Gradle provides partial builds. If you build a single subproject Gradle takes care of building all the subprojects that subproject depends on. You can also choose to rebuild the subprojects that depend on a particular subproject. Together with incremental builds this is a big time saver for

larger builds.

Many ways to manage your dependencies

Different teams prefer different ways to manage their external dependencies. Gradle provides convenient support for any strategy. From transitive dependency management with remote Maven and Ivy repositories to jars or directories on the local file system.

Gradle is the first build integration tool

Ant tasks are first class citizens. Even more interesting, Ant projects are first class citizens as well. Gradle provides a deep import for any Ant project, turning Ant targets into native Gradle tasks at runtime. You can depend on them from Gradle, you can enhance them from Gradle, you can even declare dependencies on Gradle tasks in your build.xml. The same integration is provided for properties, paths, etc …

Gradle fully supports your existing Maven or Ivy repository infrastructure for publishing and retrieving dependencies. Gradle also provides a converter for turning a Maven `pom.xml` into a Gradle script. Runtime imports of Maven projects will come soon.

Ease of migration

Gradle can adapt to any structure you have. Therefore you can always develop your Gradle build in the same branch where your production build lives and both can evolve in parallel. We usually recommend to write tests that make sure that the produced artifacts are similar. That way migration is as less disruptive and as reliable as possible. This is following the best-practices for refactoring by applying baby steps.

Groovy

Gradle's build scripts are written in Groovy or Kotlin, not XML. But unlike other approaches this is not for simply exposing the raw scripting power of a dynamic language. That would just lead to a very difficult to maintain build. The whole design of Gradle is oriented towards being used as a language, not as a rigid framework. And Groovy is our glue that allows you to tell your individual story with the abstractions Gradle (or you) provide. Gradle provides some standard stories but they are not privileged in any form. This is for us a major distinguishing feature compared to other declarative build systems. Our Groovy support is not just sugar coating. The whole Gradle API is fully Groovy-ized. Adding Groovy results in an enjoyable and productive experience.

The Gradle wrapper

The Gradle Wrapper allows you to execute Gradle builds on machines where Gradle is not installed. This is useful for example for some continuous integration servers. It is also useful for an open source project to keep the barrier low for building it. The wrapper is also very interesting for the enterprise. It is a zero administration approach for the client machines. It also enforces the usage of a particular Gradle version thus minimizing support issues.

Free and open source

Gradle is an open source project, and is licensed under the Apache License 2.0.

Why Groovy?

We think the advantages of an internal DSL (based on a dynamic language) over XML are tremendous when used in *build scripts*. There are a couple of dynamic languages out there. Why Groovy? The answer lies in the context Gradle is operating in. Although Gradle is a general purpose build tool at its core, its main focus are Java projects. In such projects the team members will be very familiar with Java. We think a build should be as transparent as possible to *all* team members.

In that case, you might argue why we don't just use Java as the language for build scripts. We think this is a valid question. It would have the highest transparency for your team and the lowest learning curve, but because of the limitations of Java, such a build language would not be as nice, expressive and powerful as it could be. [1: At http://www.defmacro.org/ramblings/lisp.html you find an interesting article comparing Ant, XML, Java and Lisp. It's funny that the 'if Java had that syntax' syntax in this article is actually the Groovy syntax.] Languages like Python, Groovy or Ruby do a much better job here. We have chosen Groovy as it offers by far the greatest transparency for Java people. Its base syntax is the same as Java's as well as its type system, its package structure and other things. Groovy provides much more on top of that, but with the common foundation of Java.

For Java developers with Python or Ruby knowledge or the desire to learn them, the above arguments don't apply. The Gradle design is well-suited for creating another build script engine in JRuby or Jython. It just doesn't have the highest priority for us at the moment. We happily support any community effort to create additional build script engines.

What is Gradle?

Overview

Gradle is an open-source build automation tool that is designed to be flexible enough to build almost any type of software. The following is a high-level overview of some of its most important features:

High performance

Gradle avoids unnecessary work by only running the tasks that need to run because their inputs or outputs have changed. You can also use a build cache to enable the reuse of task outputs from previous runs or even from a different machine (with a shared build cache).

There are many other optimizations that Gradle implements and the development team continually work to improve Gradle's performance.

JVM foundation

Gradle runs on the JVM and you must have a Java Development Kit (JDK) installed to use it. This is a bonus for users familiar with the Java platform as you can use the standard Java APIs in your build logic, such as custom task types and plugins. It also makes it easy to run Gradle on different platforms.

Note that Gradle isn't limited to building just JVM projects, and it even comes packaged with support for building native projects.

Conventions

Gradle takes a leaf out of Maven's book and makes common types of projects — such as Java projects — easy to build by implementing conventions. Apply the appropriate plugins and you can easily end up with slim build scripts for many projects. But these conventions don't limit you: Gradle allows you to override them, add your own tasks, and make many other customizations to your convention-based builds.

Extensibility

You can readily extend Gradle to provide your own task types or even build model. See the Android build support for an example of this: it adds many new build concepts such as flavors and build types.

IDE support

Several major IDEs allow you to import Gradle builds and interact with them: Android Studio, IntelliJ IDEA, Eclipse, and NetBeans. Gradle also has support for generating the solution files required to load a project into Visual Studio.

Insight

Build scans provide extensive information about a build run that you can use to identify build issues. They are particularly good at helping you to identify problems with a build's performance. You can also share build scans with others, which is particularly useful if you need ask for advice in fixing an issue with the build.

Five things you need to know about Gradle

Gradle is a flexible and powerful build tool that can easily feel intimidating when you first start. However, understanding the following core principles will make Gradle much more approachable and you will become adept with the tool before you know it.

1. Gradle is a general-purpose build tool

Gradle allows you to build any software, because it makes few assumptions about what you're trying to build or how it should be done. The most notable restriction is that dependency management currently only supports Maven- and Ivy-compatible repositories and the filesystem.

This doesn't mean you have to do a lot of work to create a build. Gradle makes it easy to build common types of project — say Java libraries — by adding a layer of conventions and prebuilt functionality through *plugins*. You can even create and publish custom plugins to encapsulate your own conventions and build functionality.

2. The core model is based on tasks

Gradle models its builds as Directed Acyclic Graphs (DAGs) of tasks (units of work). What this means is that a build essentially configures a set of tasks and wires them together — based on their dependencies — to create that DAG. Once the task graph has been created, Gradle determines which tasks need to be run in which order and then proceeds to execute them.

This diagram shows two example task graphs, one abstract and the other concrete, with the dependencies between the tasks represented as arrows:

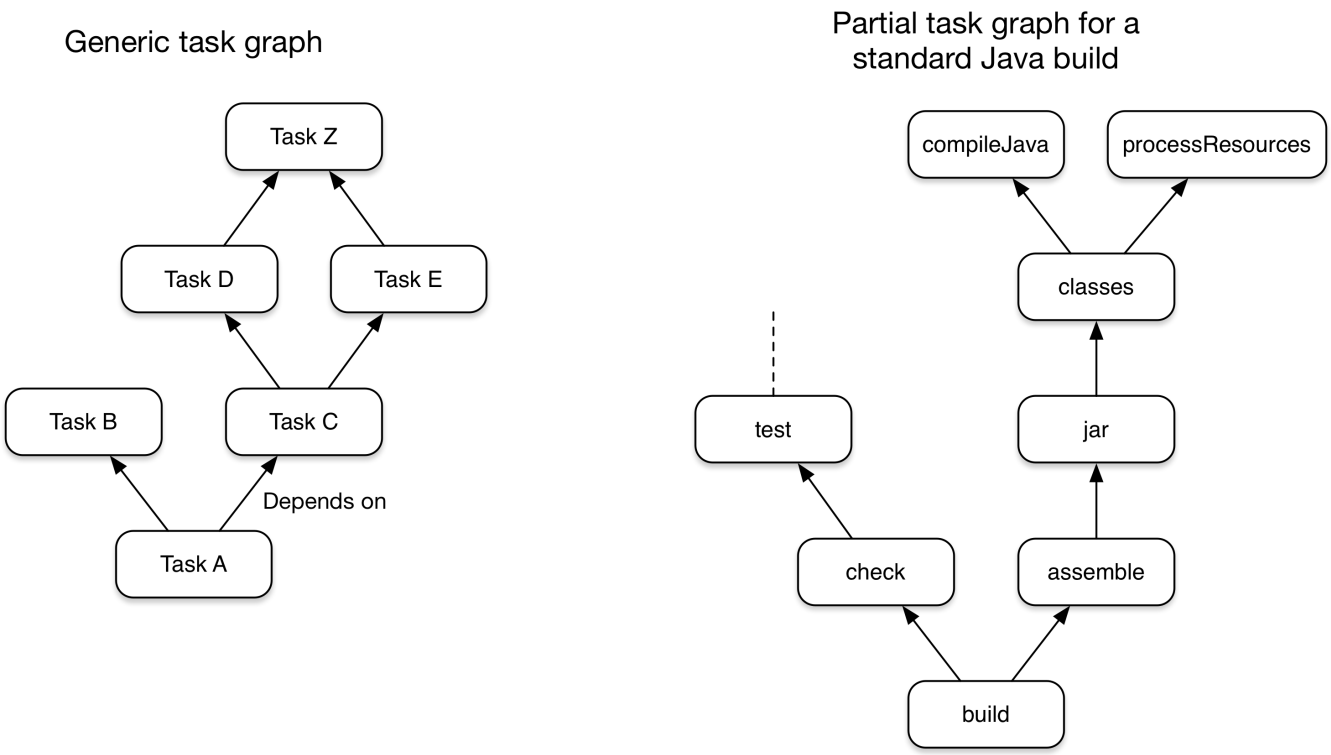


Figure 1. Two examples of Gradle task graphs

Almost any build process can be modeled as a graph of tasks in this way, which is one of the reasons why Gradle is so flexible. And that task graph can be defined by both plugins and your own build scripts, with tasks linked together via the task dependency mechanism.

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

In fact, all of the above are optional depending on what the task needs to do. Some tasks — such as the standard lifecycle tasks — don't even have any actions. They simply aggregate multiple tasks together as a convenience.

NOTE

You choose which task to run. Save time by specifying the task that does what you need, but no more than that. If you just want to run the unit tests, choose the task that does that — typically `test`. If you want to package an application, most builds have an `assemble` task for that.

One last thing: Gradle's incremental build support is robust and reliable, so keep your builds running fast by avoiding the `clean` task unless you actually do want to perform a clean.

3. Gradle has several fixed build phases

It's important to understand that Gradle evaluates and executes build scripts in three phases:

1. Initialization

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

2. Configuration

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

3. Execution

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

These phases form Gradle's Build Lifecycle.

NOTE	*Comparison to Apache Maven terminology* Gradle's build phases are not like Maven's phases. Maven uses its phases to divide the build execution into multiple stages. They serve a similar role to Gradle's task graph, although less flexibly. Maven's concept of a build lifecycle is loosely similar to Gradle's lifecycle tasks.

Well-designed build scripts consist mostly of declarative configuration rather than imperative logic. That configuration is understandably evaluated during the configuration phase. Even so, many such builds also have task actions — for example via `doLast {}` and `doFirst {}` blocks — which are evaluated during the execution phase. This is important because code evaluated during the configuration phase won't see changes that happen during the execution phase.

Another important aspect of the configuration phase is that everything involved in it is evaluated *every time the build runs*. That is why it's best practice to avoid expensive work during the configuration phase. Build scans can help you identify such hotspots, among other things.

4. Gradle is extensible in more ways than one

It would be great if you could build your project using only the build logic bundled with Gradle, but that's rarely possible. Most builds have some special requirements that mean you need to add custom build logic.

Gradle provides several mechanisms that allow you to extend it, such as:

- Custom task types.

  When you want the build to do some work that an existing task can't do, you can simply write your own task type. It's typically best to put the source file for a custom task type in the *buildSrc* directory or in a packaged plugin. Then you can use the custom task type just like any of the Gradle-provided ones.

- Custom task actions.

  You can attach custom build logic that executes before or after a task via the Task.doFirst() and Task.doLast() methods.

- Extra properties on projects and tasks.

  These allows you to add your own properties to a project or task that you can then use from your own custom actions or any other build logic. Extra properties can even be applied to tasks that aren't explicitly created by you, such as those created by Gradle's core plugins.

- Custom conventions.

  Conventions are a powerful way to simplify builds so that users can understand and use them more easily. This can be seen with builds that use standard project structures and naming conventions, such as Java builds. You can write your own plugins that provide conventions — they just need to configure default values for the relevant aspects of a build.

- A custom model.

  Gradle allows you to introduce new concepts into a build beyond tasks, files and dependency configurations. You can see this with most language plugins, which add the concept of *source sets* to a build. Appropriate modeling of a build process can greatly improve a build's ease of use and its efficiency.

5. Build scripts operate against an API

It's easy to view Gradle's build scripts as executable code, because that's what they are. But that's an implementation detail: well-designed build scripts describe *what* steps are needed to build the software, not *how* those steps should do the work. That's a job for custom task types and plugins.

NOTE	There is a common misconception that Gradle's power and flexibility come from the fact that its build scripts are code. This couldn't be further from the truth. It's the underlying model and API that provide the power. As we recommend in our best practices, you should avoid putting much, if any, imperative logic in your build scripts.

Yet there is one area in which it is useful to view a build script as executable code: in understanding how the syntax of the build script maps to Gradle's API. The API documentation — formed of the Groovy DSL Reference and the Javadocs — lists methods and properties, and refers to closures and actions. What do these mean within the context of a build script? Check out the Groovy Build Script Primer to learn the answer to that question so that you can make effective use of the API documentation.

NOTE	As Gradle runs on the JVM, build scripts can also use the standard Java API. Groovy build scripts can additionally use the Groovy APIs, while Kotlin build scripts can use the Kotlin ones.

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?.

Even if you're experienced with using Gradle, we suggest you read the section 5 things you need to know about Gradle as it clears up some common misconceptions.

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:

- Creating a basic Gradle build
- Building Android apps
- Building Java libraries
- Building Kotlin JVM libraries
- Building C++ libraries
- Creating build scans

There are also many other tutorials and guides available, which you can filter by category — for example Fundamentals.

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 5 things you need to know about 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 more tutorials on various Gradle features among the Gradle Guides.

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!, Homebrew, or Scoop, 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.

Installing with a package manager

SDKMAN! is a tool for managing parallel versions of multiple Software Development Kits on most Unix-based systems.

```
   sdk install gradle
```

Homebrew is "the missing package manager for macOS".

```
   brew install gradle
```

Scoop is a command-line installer for Windows inspired by Homebrew.

```
   scoop install gradle
```

Chocolatey is "the package manager for Windows".

```
   choco install gradle
```

MacPorts is a system for managing tools on macOS:

```
   sudo port install gradle
```

↓ 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-5.0-rc-2-bin.zip
   ls /opt/gradle/gradle-5.0-rc-2
LICENSE  NOTICE  bin  getting-started.html  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-5.0-rc-2` 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

For running Gradle, firstly add the environment variable `GRADLE_HOME`. This should point to the unpacked files from the Gradle website. Next add `GRADLE_HOME/bin` to your `PATH` environment variable. Usually, this is sufficient to run Gradle.

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-5.0-rc-2/bin
```

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-5.0-rc-2\bin`. Click OK to save.

↓ 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 5.0-rc-2
------------------------------------------------------------

Build time:   2018-02-21 15:28:42 UTC
Revision:     819e0059da49f469d3e9b2896dc4e72537c4847d

Groovy:       2.4.15
Ant:          Apache Ant(TM) version 1.9.9 compiled on February 2 2017
JVM:          1.8.0_151 (Oracle Corporation 25.151-b12)
OS:           Mac OS X 10.13.3 x86_64
```

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.

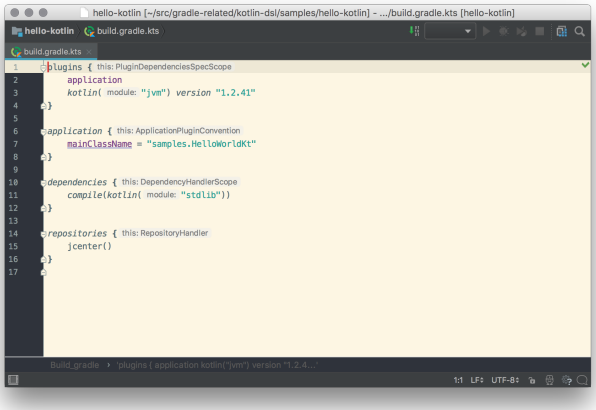
Next steps

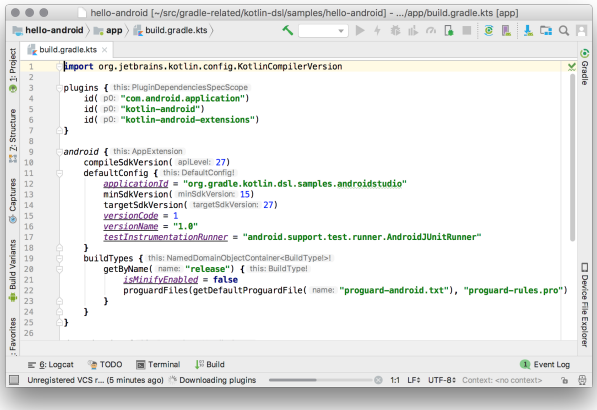
Now that you have Gradle installed, use these resources for getting started:

- Create your first Gradle project by following the Creating New Gradle Builds tutorial.
- 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.

Gradle Kotlin DSL Primer

Gradle's Kotlin DSL provides a pleasant editing experience in supported IDEs: content-assist, refactoring, documentation, and more.





TIP

This user manual and guides contain build script excerpts that demonstrate both the Groovy DSL and the Kotlin DSL. This is the best place where to find how to do this and what with the Gradle Kotlin DSL ; and it covers all Gradle features from using plugins to customizing the dependency resolution behavior.

If you are looking into migrating an existing build to the Gradle Kotlin DSL, please also check out the migration guide.

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 should be useful to you.
- Use of the plugins {} block to declare Gradle plugins significantly improves the editing experience, and is highly recommended.

Limitations

- There are some situations where the Kotlin DSL is slower. First use, on clean checkouts or ephemeral CI agents for example, are known to be slower. The same applies to the scenario in which something in the *buildSrc* directory changes, which invalidates build-script caching. The main bottleneck is script compilation.
- In IntelliJ IDEA, you must import your project from the Gradle model to get content-assist and refactoring tools for Kotlin DSL scripts.
- The Kotlin DSL will not support `model {}` elements. This is part of the discontinued Gradle Software Model. Applying model rules from scripts is supported though, see the model-rules sample.
- Enabling the incubating configuration on demand feature is not recommended as it can lead to very hard-to-diagnose problems.
- Functions of the Gradle API taking `Any` that accept lazy constructs such as `CopySpec.from(Any)` don't support Kotlin lambdas, use a `Callable { .. }` or a `provider { .. }` instead, see gradle/kotlin-dsl#1077.

If you run to trouble or a suspected bug, please take advantage of the `gradle/kotlin-dsl` issue tracker.

IDE support

Kotlin DSL is fully supported in Intellij IDEA and Android Studio. Other IDEs do not yet provide helpful tools for editing Gradle Kotlin DSL files, however, importing and working with Kotlin DSL-based builds work as usual.

Table 1. IDE support matrix

	Build import	Syntax highlighting [1]	Semantic editor [2]
IntelliJ IDEA	✓	✓	✓
Android Studio	✓	✓	✓
Eclipse IDE	✓	✓	
CLion	✓	✓	
Apache Netbeans	✓	✓	
Visual Studio Code (LSP)	✓	✓	
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

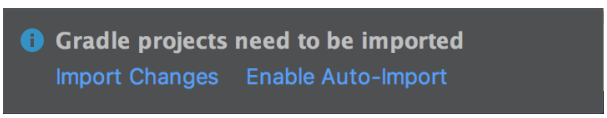
In IntelliJ IDEA, you must import your project from the Gradle model to get content-assist and refactoring tools for Kotlin DSL scripts.

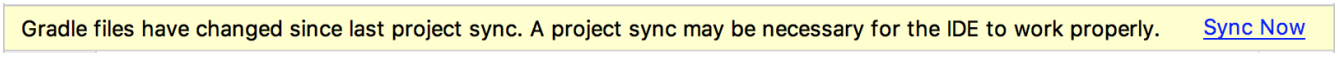
In addition, IntelliJ IDEA and Android Studio might spawn up to 3 Gradle daemons when editing scripts; one per each type of script: project scripts, settings scripts and initialization scripts. Builds with slow configuration time might affect the IDE responsiveness, please check out the performance guide.

Automatic build import vs. automatic reload of script dependencies

Both IntelliJ IDEA and Android Studio will detect when you make changes to your build logic and offer suggestions as to what to do.

Both will suggest to re-import the whole build into the IDE:





Both will also suggest reloading script dependencies when editing a build script:



It is recommended to **disable automatic build import** but **enable automatic reload of script dependencies.** That way you get early feedback while editing Gradle scripts while controlling when the whole build setup gets synchronized with your IDE.

Troubleshooting

The IDE support is provided by two components:

- the IntelliJ IDEA or Android Studio Kotlin Plugin version,
- the version of Gradle used to import the build.

In you run into trouble, the first thing to try is to run `gradle tasks` to see if your issue is IDE related only and to possibly get more details. If running Gradle works but your script editor is complaining then you can try restarting your IDE and invalidating its caches.

If, after trying the above, you suspect an issue with the `.gradle.kts` script editor you can:

- run `gradle tasks` to get more details,
- check logs in
  - `$HOME/Library/Logs/gradle-kotlin-dsl` on macos,
  - `$HOME/.gradle-kotlin-dsl/logs` on Linux,
  - `$HOME/Application Data/gradle-kotlin-dsl/log` on Windows,
- and finally, open an issue on the `gradle/kotlin-dsl` issue tracker with as much detail as you can.

For IDE issues out of the `.gradle.kts` script editor, please open issues in the IDE issue tracker, respectively

- JetBrains's IDEA issue tracker,
- Google's Android Studio issue tracker.

Last but not least, if you face problems with Gradle itself or with the Gradle Kotlin DSL, please open issues on the corresponding issue tracker, respectively

- `gradle/gradle` issue tracker,
- `gradle/kotlin-dsl` issue tracker.

Gradle Kotlin DSL scripts

Just like the Gradle Groovy DSL, the Gradle Kotlin DSL is implemented on top of the Java Gradle API. Everything you can read in a Gradle Kotlin DSL script is Kotlin code compiled and executed by Gradle. In Gradle Kotlin DSL scripts you write Kotlin code that uses objects, functions and properties from the Gradle API and applied Gradle plugins APIs.

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 use the Kotlin DSL, simply name your files `build.gradle.kts` instead of `build.gradle`.

The settings file, `settings.gradle`, can also be renamed `settings.gradle.kts`.

In a multi-project build, you can have some modules using the Groovy DSL (with `build.gradle`) and others using the Kotlin DSL (with `build.gradle.kts`).

On top of that, apply the following conventions for better IDE support:

- Name scripts that are applied to `Settings` according to the pattern `*.settings.gradle.kts`,
- Name init scripts according to the pattern `*.init.gradle.kts`.

TIP

Precompiled script plugins

The Gradle Kotlin DSL can also be used to author Gradle plugins as part of your local build logic or for published community or in-house plugins.

You can either write plugins in plain Kotlin (`.kt` files in `src/main/kotlin`) and make use of the Gradle Kotlin DSL API, or, use precompiled script plugins (`.gradle.kts` files in `src/main/kotlin`).

See the kotlin-dsl plugin section below for more information.

Implicit imports

All `gradle.kts` script has implicit imports comprised of:

- the default Gradle API imports,
- the Gradle Kotlin DSL API: `org.gradle.kotlin.dsl.*`.

CAUTION

Avoiding Gradle Kotlin DSL internal APIs

Use of Gradle Kotlin DSL internal APIs in plugins and build scripts has the potential to break builds when either Gradle or plugins change.

The Gradle Kotlin DSL extends the Gradle public API definition with `org/gradle/kotlin/dsl/*` excluding any subpackages.

Configuring plugins

Type-safe model accessors

Gradle Plugins contribute reusable and configurable models to a build. For example, the `java` plugin contributes artifact configurations such as `implementation` and `testImplementation`, it also adds a Gradle extension named `sourceSets` that can be used to configure and create source sets, it also registers Gradle tasks such as `compileJava` etc...

Gradle scripts configure what applied plugins contribute to the build model. For example, adding dependencies or configuring dependency resolution on the `implementation` artifact configuration, changing the source directories for the `main` and `test` Java source sets, or, making `compileJava` fail on compiler warnings.

The Gradle Kotlin DSL provides a static Kotlin view over the dynamic model contributed by plugins in the form of type-safe model accessors.

NOTE	*Configuration avoidance* Configuration avoidance was introduced in Gradle 4.9. It allows Gradle to avoid configuring unnecessary build logic, leading to faster build configuration time. See the corresponding Evolving the Gradle API to reduce configuration time blog post. The Gradle Kotlin DSL embraces configuration avoidance by making the type-safe model accessors leverage the new APIs.

Where and when are type-safe model accessors available?

The Gradle Kotlin DSL currently supports type-safe model accessors for:

- Artifact Configurations
- Project Extensions and Conventions
- Elements of the `tasks` container
- Elements of Project Extensions that are containers
- Extensions on each of the above

IMPORTANT	Only project build scripts have type-safe model accessors. Init scripts, settings scripts and applied script plugins don't. Nor do precompiled script plugins. Theses 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, after the `plugins {}` block. It means that all model elements contributed after that point can not be reached with a type-safe model accessor. It also means that accessors for model elements contributed by plugins applied by parent projects are available.

Let's look at a project build script that makes use of accessors:

Example 1. Using type-safe model accessors

build.gradle.kts

```
plugins {
    `java-library`
}

dependencies {                                  ①
    api("junit:junit:4.12")
    implementation("junit:junit:4.12")
    testImplementation("junit:junit:4.12")
}

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 `api`, `implementation` and `testImplementation` artifact configurations contributed by the `java-library` plugin

② the `sourceSets` project extension contributed by the `java-library` plugin

③ the `main` source set created by default by the `java-library` plugin

④ the `java` project convention contributed by the `java-library` plugin

⑤ the `test` task contributed by the `java-library` plugin

TIP

Because your IDE knows about the accessors, it will include them when you ask your IDE for suggestions. This will happen both at the top level of your build scripts — most plugin extensions are added to the `Project` object — and within an extension's configuration block.

Note that accessors for elements of containers such as `configurations`, `tasks` or `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.

```
tasks.test {
    // lazy configuration
}

// Lazy reference
val testProvider: TaskProvider<SourceSet> = 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, accessors for elements are of type `NamedDomainObjectProvider<T>` providing the same functionality.

What to do when type-safe model accessors are not available?

Following up on the sample build script shown above using the `plugins {}` block, here is the exact same build logic but using the imperative `apply(plugin = "some.id")`. Note that because imperative plugin application happens in the body of a build script, the model contributed by the plugin can not be made available as type-safe accessors.

Example 2. Configuring plugins without type-safe accessors

build.gradle.kts

```
apply(plugin = "java-library")

dependencies {
    "api"("junit:junit:4.12")
    "implementation"("junit:junit:4.12")
    "testImplementation"("junit:junit:4.12")
}

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
    }
}
```

TIP

All models contributed by plugins are reachable via the API.

In some circumstances, type-safe accessors are not available: imperative configuration (`apply(plugin = "id")` or configuration of model elements created in the same script), script plugins (`apply(from = "script-plugin.gradle.kts")`), cross project configuration (see below), Gradle plugins implemented in Kotlin and precompiled script plugins (see below).

Whenever you can't find a type-safe accessor you should fallback to using the API.

Using the API instead of accessors to configure plugins is not that different, a bit more verbose though. It also requires knowing names and/or types of the configured model elements. We'll now

see how those can be discovered by looking at the script above in details.

Let's start with artifact configurations.

Example 3. Artifact configurations

build.gradle.kts

```
apply(plugin = "java-library")

dependencies {
    "api"("junit:junit:4.12")
    "implementation"("junit:junit:4.12")
    "testImplementation"("junit:junit:4.12")
}

configurations {
    "implementation" {
        resolutionStrategy.failOnVersionConflict()
    }
}
```

Artifact configurations have a name. Here, instead of using an accessor we refer to them by name using string literals.

One can discover what artifact configurations are available by either looking at the `java-library` plugin documentation, source code or by running `gradle dependencies`.

Now let's look at `Project` extensions and conventions.

Example 4. 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
}
```

`Project` extensions and conventions have a name and a type. Here, instead of using the `sourceSets` and `java` accessors we use the `configure<T> {}` function to refer to them by type and apply a configuration block. `sourceSets` is a Gradle extension on `Project` of type `SourceSetContainer`. `java` is a Gradle convention on `Project` of type `JavaPluginConvention`.

One can discover what extensions and conventions are available by either looking at the applied plugins documentation, source code or by running `gradle kotlinDslAccessorsReport` which prints the Kotlin code for accessing the model elements contributed by applied plugins providing names and types.

Note that you can also use the `the<T>()` function if you only need a reference to the extension or convention or for one liners:

```
the<SourceSetContainer>()["main"].srcDir("src/core/java")
```

The snippet above also configures the existing element of the source set container named `main`:

Example 5. Elements of `Project` extensions that are containers

build.gradle.kts

```
apply(plugin = "java-library")

configure<SourceSetContainer> {
    named("main") {
        java.srcDir("src/core/java")
    }
}
```

Source sets have a name. Here, instead of using the `main` accessor we use the Gradle API and refer to it by name using a string literal.

One can discover what source sets, or what existing elements are present in any container, by either looking at the applied plugins documentation, source code or by running `gradle kotlinDslAccessorsReport` which prints the Kotlin code for accessing the model elements contributed by applied plugins providing names and types.

This is very similar to what our sample build script does to configure a task:

Example 6. Tasks

build.gradle.kts

```
apply(plugin = "java-library")

tasks {
    named<Test>("test") {
        testLogging.showExceptions = true
    }
}
```

Gradle tasks have a name and a type. Here, instead of using an accessor we use the Gradle API and refer to them by name and type. 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 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`, and then to know the type of a given task by running `gradle help --task taskName`:

```
   ./gradlew help --task test
...
Type
     Test (org.gradle.api.tasks.testing.Test)
```

Note that the IDE can assist us with the required imports, so we only need the simple names, i.e. without the full packages. In this case, no need for an import, the `Test` task type is part of the Gradle API and of implicit imports.

About conventions

Some of the Gradle core plugins expose configurability with the help of a so-called `Convention`. The `Convention` concept precedes that of extension and serves a similar purpose. Please avoid using the `Convention` concept when writing new plugins. The long term plan is to migrate all Gradle core plugins to use extensions and remove the `Convention` concept altogether.

As seen above, the Gradle Kotlin DSL provides accessors only for `Conventions` on `Project`. There are situations that require you to interact with a Gradle plugin that uses `Conventions` on other objects. The most common example is configuring source sets conventions. The Gradle Kotlin DSL provides the `withConvention(T::class) {}` extension function to do so:

Example 7. Configuring source set conventions

build.gradle.kts

```
plugins {
    groovy
}

sourceSets {
    main {
        withConvention(GroovySourceSet::class) {
            groovy.srcDir("src/core/groovy")
        }
    }
}
```

Multi-project builds (e.g. Android)

Multi-project builds are very common. Android builds are, for example, multi-project builds from the start. When creating a new build from Android Studio it creates a multi-project build.

To get the most benefits of the Gradle Kotlin DSL you should strive to keep your build scripts declarative. The main thing to remember here is that in order to get type-safe accessors, plugins must be applied before the body of build scripts. As seen above, this can be achieved by using the `plugins {}` block.

Applying plugins

In multi-project builds you should centrally declare plugin dependencies in the root project build file, specifying the version. This always allow for applying the plugins in sub projects build scripts using the `plugins {}` block.

It is recommended to do so using the `plugins {}` block in your root project build script as documented in the Using Gradle plugins chapter:

Example 8. 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 "1.5.4" 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 {
    mainClassName = "example.App"
}

ratpack.baseDir = file("src/ratpack/baseDir")
```

If your build requires additional plugin repositories, they should be declared in the `pluginManagement {}` block in your `settings.gradle.kts` file:

Example 9. Declare additional plugin repositories

settings.gradle.kts

```
pluginManagement {
    repositories {
        jcenter()
        gradlePluginPortal()
    }
}
```

Plugins fetched from a source other than the Gradle Plugin Portal can only be requested using the `plugins {}` block if they were published with their plugin marker artifacts.

For example, the Android Plugin for Gradle present in the `google()` repository is published without plugin marker artifacts, at least up to version 3.2.0 of the plugin. Please star the corresponding Google issue.

In that case, you can fall back to declaring your plugin dependencies using the `buildscript {}` block in the root project build script:

Example 10. 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:3.2.0")
    }
}
```

lib/build.gradle.kts

```
plugins {
    id("com.android.library")
}

android {
    // ...
}
```

app/build.gradle.kts

```
plugins {
    id("com.android.application")
}

android {
    // ...
}
```

It's not that different from what Android Studio produces when creating a new build. The main difference is that sub projects build scripts apply plugins using the `plugins {}` block.

If you want to use a plugin that can't be applied using the `plugins {}` block directly in your root project or in a single project build, see the dedicated section below.

Cross-configuring projects

You might want to cross-configure projects, for example configuring sub projects from the root project build script. In that case, you won't have the help of the type-safe accessors for model elements contributed by the plugins and have to rely on string literals and the API.

For the sake of the example, let's look at the same Java/Ratpack build as above but fully configured from the root build script:

Example 11. 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 "1.5.4" 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")
    val ratpack = the<RatpackExtension>()
    dependencies {
        "implementation"(project(":domain"))
        "implementation"(project(":infra"))
        "implementation"(ratpack.dependency("dropwizard-metrics"))
        "runtime"("org.slf4j:slf4j-simple:1.7.25")
    }
    configure<ApplicationPluginConvention> {
        mainClassName = "example.App"
    }
    ratpack.baseDir = file("src/ratpack/baseDir")
}
```

It’s more verbose and requires more gymnastics to write but it works and can be useful. See the Configuring plugins section above for a detailed explanation.

Plugins not usable in `plugins {}`

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.

If your build is a multi-project build and you don't need to apply such a plugin to your root project, see the dedicated section above. Otherwise, keep reading.

The Android Plugin for Gradle, for example, 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.

We will show you in this section how to apply the Android Plugin to your single or root project.

NOTE	When publishing plugins, please use Gradle's built-in Java Gradle Plugin Plugin. It automates the publication of the metadata necessary to make your plugins usable with the `plugins {}` block.

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 file
- Map the plugin ID to the corresponding artifact coordinates

This happens in the settings file within the `pluginManagement {}` block. The following sample maps the `com.android.application` ID to the `com.android.tools.build:gradle:<version>` artifact available in the `google()` repository:

Example 12. 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 "3.2.0"
}

android {
    // ...
}
```

See the Plugin Management section of the Gradle user manual for more information.

In the case of a multi-project build, the above is very similar to using `buildscript { dependencies { classpath(..) } }` in your root project build script to depend on third party plugins and then using `plugins {}` in your sub project build scripts to apply them. The main difference is that the root project can also benefit from the `plugins {}` block and it is more aligned with Gradle best practices.

The same approach can be used to resolve plugins from composite builds, which do not expose plugin markers yet. Simply map the plugin ID to the corresponding artifact coordinates as shown in the Android samples above.

Working with containers

The Gradle build model makes heavy use of containers. For example, Gradle core contributes `configurations` which is a container of `Configuration` objects and `tasks` which is a container of `Task` objects; core plugins such as `java` and `distribution` contribute containers such as `sourceSets` and `distributions`. Community plugins also contribute containers, such as `android.buildTypes`

contributed by the `android` plugin. Gradle plugins create containers using `Project.container<Type>()`.

> **TIP** Note that if you are configuring existing elements on supported containers you can leverage the type-safe accessors as described in the Configuring plugins section above.

The Gradle Kotlin DSL provides build authors with uniform ways of interacting with containers. We already saw some examples configuring source sets or tasks in the Configuring plugins section above. Let's now look in detail at how to interact with containers using the Gradle Kotlin DSL by using the `tasks` container as an example.

Using the containers API

All the containers implement `NamedDomainObjectContainer<DomainObjectType>`, some of them can contain objects of different types and implement `PolymorphicDomainObjectContainer<BaseType>`. The first and simplest way to interact with containers is through their API.

Example 13. Using the containers 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 to the existing task named `check`

② registers a new 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 task named `assemble` and configures it

⑥ registers a new task named `myTask2` and configures it

⑦ gets a reference to the existing task named `test` of type `Test` and configures it

⑧ registers a new task named `myCopy2` of type `Copy` and configures it

NOTE The above rely on configuration avoidance APIs. If you need to eagerly configure or register container elements simply replace `named` with `getByName` and `register` with `create`.

Via Kotlin delegated properties

Another way to interact with containers is via Kotlin delegated properties that are particularly useful if you need a reference to a container element for further usage. By not using string literals for the element names, Kotlin delegated properties allow for easy renaming via IDE refactorings. The sample below does the exact same things as the one in the section above, reusing references to tasks instead of using task paths in string literals:

Example 14. Via 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)
}
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")
}
```

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`.

In a container scope

When configuring several elements of a container one can group interactions in a block in order not to repeat the container's name on each interaction. The following example mixes the usage of type-safe accessors, the container API and Kotlin delegated properties:

Example 15. 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 properties

The Gradle Kotlin DSL also provides some sugar on top of the Gradle API for working with properties.

Gradle, Settings & Project properties

`Gradle`, `Settings` and `Project` properties are documented in the Build Environment chapter.

They can also be used via Kotlin delegated properties, see the sample below for `Project` properties:

build.gradle.kts

```
val myProperty: String by project
val myNullableProperty: String? by project
```

It works the exact same way in settings scripts for `Settings` properties and in init scripts for `Gradle` properties, respectively replacing `by project` with `by settings` or `by gradle`.

Extra properties

Extra properties are documented in the Writing Build Scripts chapter.

On every `ExtensionAware` type they can be used via the `extra` property as a Kotlin delegated property, see the sample below.

build.gradle.kts

```
// Define new extra properties
val myNewProperty by extra("initial value")
val myOtherNewProperty by extra { "lazy initial value" }

// Reference existing extra properties
val myProperty: String by extra
val myNullableProperty: String? by extra
```

Let's say the snippet above is from your root project build script, in order to reference the `myNewProperty` from a sub project you'd do:

my-sub-project/build.gradle.kts

```
val myNewProperty: String by rootProject.extra
```

It works the same way in settings or init scripts.

All `ExtensionAware` types can host extra properties, `Task` is a common example:

build.gradle.kts

```
tasks {
    test {
        // Define new extra properties
        val myNewTaskProperty by extra("initial value")
        val myOtherNewTaskProperty by extra { "lazy initial value" }

        // Reference existing extra properties
        val myTaskProperty: String by extra
        val myNullableTaskProperty: String? by extra
    }
}
```

The `kotlin-dsl` plugin

When applied to a project, the `kotlin-dsl` Gradle plugin configures the project's Kotlin source sets to enable Gradle Kotlin DSL features.

It is useful for all projects implementing build logic: buildSrc, included builds for build logic or Gradle plugin builds.

Example 16. Apply the `kotlin-dsl` plugin to `buildSrc`

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.
    jcenter()
}
```

The `kotlin-dsl` plugin

- applies the org.jetbrains.kotlin.jvm plugin,
- configures an embedded repository that contains all Kotlin libraries embedded with the Gradle Kotlin DSL,
- adds the `kotlin-stdlib-jdk8`, `kotlin-reflect` and `gradleKotlinDsl()` dependencies to the `compileOnly` and `testImplementation` configurations,
- configures the Kotlin compiler with the same set of Kotlin compiler settings that is used for Gradle Kotlin DSL scripts,
- and, enables precompiled script plugins, see below.

CAUTION The `kotlin-dsl` plugin enables experimental Kotlin compiler features. See the Kotlin compiler arguments section below for more information.

By default, the `kotlin-dsl` plugin warns about using experimental features of the Kotlin compiler. You can silence the warning by setting the `experimentalWarning` property of the `kotlinDslPluginOptions` extension to `false` as follows:

Example 17. Disable experimental Kotlin compiler features warning

buildSrc/build.gradle.kts

```
plugins {
    `kotlin-dsl`
}

kotlinDslPluginOptions {
    experimentalWarning.set(false)
}
```

Precompiled script plugins

The `kotlin-dsl` plugin can be used to author Gradle plugins as part of your local build logic or for published community or in-house plugins. You can write plugins in plain Kotlin (`.kt` files in `src/main/kotlin`) making use of the `gradleKotlinDsl()` API, and, use precompiled script plugins (`.gradle.kts` files in `src/main/kotlin`).

Precompiled script plugins are Gradle Kotlin DSL scripts compiled as part of a regular Kotlin source-set and distributed in the usual way, as java class files packaged in some library, meant to be consumed as a binary Gradle plugin. They follow the same file name conventions, are compiled with the same Kotlin compiler settings, and plugin metadata is generated for each one of them thanks to the integration with the `java-gradle-plugin`. They can then be applied by plugin ID like any other plugin.

The Gradle plugin ID by which the precompiled script can be referenced is derived from its name and optional package declaration.

Thus, the script `src/main/kotlin/java-library-convention.gradle.kts` is exposed as the `java-library-convention` plugin (assuming it has no package declaration) whereas the script `src/main/kotlin/my/java-library-convention.gradle.kts` is exposed as the `my.java-library-convention` plugin, again assuming it has the matching package declaration.

Let's walk through the steps needed to implement a plugin in `buildSrc` using a precompiled script plugin.

First, you need a `buildSrc/build.gradle.kts` file applying the `kotlin-dsl` plugin:

Example 18. Apply the `kotlin-dsl` plugin to `buildSrc`

buildSrc/build.gradle.kts

```
plugins {
    `kotlin-dsl`
}
```

`buildSrc` being a separate build, don't forget to add a `buildSrc/settings.gradle.kts` file. It can be empty.

Then create the new `java-library-convention.gradle.kts` file in `buildSrc/src/main/kotlin`:

Example 19. A `java-library` convention plugin

buildSrc/src/main/kotlin/java-library-convention.gradle.kts

```
plugins {
    `java-library`
    checkstyle
}

configure<JavaPluginConvention> {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}

configure<CheckstyleExtension> {
    maxWarnings = 0
    // ...
}

tasks.withType<JavaCompile> {
    options.isWarnings = true
    // ...
}

dependencies {
    "testImplementation"("junit:junit:4.12")
    // ...
}
```

IMPORTANT	Precompiled script plugins do not get type-safe accessors. This limitation will be lifted in a future Gradle release.

The precompiled script plugin can then be applied to the root project as follows:

Example 20. Apply the `java-library-convention` plugin

build.gradle.kts

```
plugins {
    id("java-library-convention")
}
```

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 example, Gradle 4.3 ships with the Gradle Kotlin DSL v0.12.1 that includes Kotlin 1.1.51 versions of these modules. 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 Gradle 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.

`-XX:NewInference`
: Enables the experimental Kotlin compiler inference engine (required for SAM conversion for Kotlin functions).

`-XX:SamConversionForKotlinFunctions`
: Enables SAM (Single Abstract Method) conversion for Kotlin functions in order to allow Kotlin build logic to expose and consume `org.gradle.api.Action<T>` based APIs. Such APIs can then be used uniformly from both the Kotlin and Groovy DSLs.

  As an example, given the following hypothetical Kotlin function with a Java SAM parameter type:

  ```
  fun kotlinFunctionWithJavaSam(action: org.gradle.api.Action<Any>) = TODO()
  ```

  SAM conversion for Kotlin functions enables the following usage of the function:

  ```
  kotlinFunctionWithJavaSam {
      // ...
  }
  ```

  Without SAM conversion for Kotlin functions one would have to explicitly convert the passed lambda:

  ```
  kotlinFunctionWithJavaSam(Action {
      // ...
  })
  ```

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 21. Call 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 see 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 22. Call 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 23. 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 24. Use `closureOf<T> {}`

build.gradle.kts

```
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 25. Use `delegateClosureOf<T> {}`

build.gradle.kts

```
farms {
    farm("OldCoreWar", delegateClosureOf<FarmExtension> {
        // Config for the war here
    })
}
```

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 26. 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"
    }))
}
```

Also see the groovy-interop sample.

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 27. 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

The maven-plugin sample demonstrates the use of the `withGroovyBuilder()` utility extensions for configuring the `uploadArchives` task to deploy to a Maven repository with a custom POM using Gradle's core Maven Plugin. Note that the recommended Maven Publish Plugin provides a type-safe and Kotlin-friendly DSL that allows you to easily do the same and more without resorting to `withGroovyBuilder()`.

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 28. 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

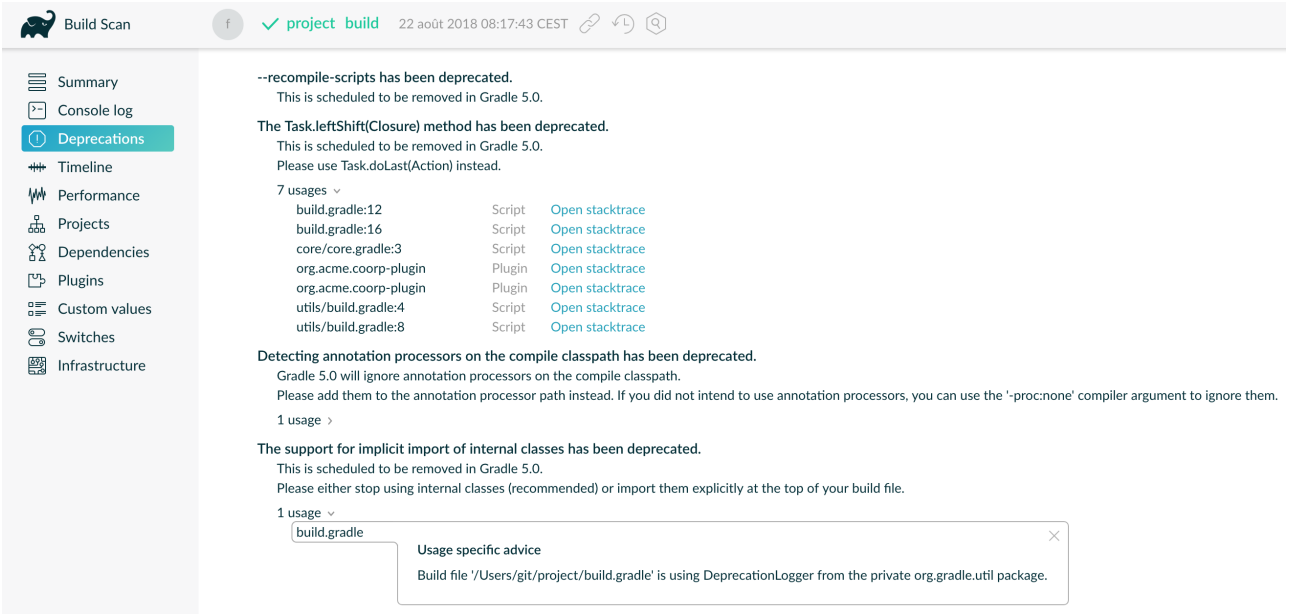
③ The Groovy script uses dynamic Groovy to configure plugin

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, you will also need to apply the changes since 4.4, 4.5, etc.

We recommend the following steps for all users:

1. If you are not already on version 4.10.2, read the sections below for help upgrading your project to 4.10.2.
2. 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. 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.

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 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 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

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(https://docs.gradle.org/4.8/release-notes.html#overwriting-gradle's-built-in-tasks).

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 2.5.3.
- 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:

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

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

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

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

[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.

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`):

```
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.

```
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:

```
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.baseName` doesn't change after `artifactId` is set:

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

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

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

[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:

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

to this:

```
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:

```
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.

Using Gradle Builds

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 run a task and all of its dependencies.

```
$ gradle myTask
```

You can learn about what projects and tasks are available in the project reporting section.

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 :mySubproject:taskName
$ gradle mySubproject:taskName
```

You can also run a task for all subprojects using the task name only. For example, this will run the "test" task for all subprojects when invoked from the root project directory.

```
$ gradle test
```

When invoking Gradle from within a subproject, the project name should be omitted:

```
$ cd mySubproject
$ 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. 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
```

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.

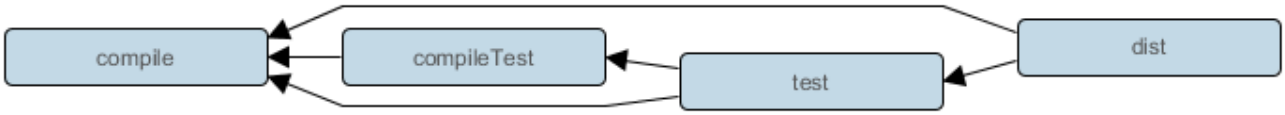


Figure 2. Simple Task Graph

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.

Continuing the build when a failure occurs

By default, Gradle will abort execution and fail the build as soon as any task fails. This allows the build to complete sooner, but hides other failures that would have occurred. In order to discover as many failures as possible in a single build execution, you can use the `--continue` option.

```
$ gradle test --continue
```

When executed with `--continue`, Gradle will execute *every* task to be executed where all of the dependencies for that task completed without failure, instead of stopping as soon as the first failure is encountered. Each of the encountered failures will be reported at the end of the build.

If a task fails, any subsequent tasks that were depending on it will not be executed. For example, tests will not run if there is a compilation failure in the code under test; because the test task will depend on the compilation task (either directly or indirectly).

Task 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.

You can also abbreviate each word in a camel case task name. For example, you can execute task `compileTest` by running `gradle compTest` or even `gradle cT`.

Abbreviated camel case task name

```
$ gradle cT

> Task :compile
compiling source

> Task :compileTest
compiling unit tests

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

You can also use these abbreviations with the -x command-line option.

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
```

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)

Description
     Builds the JAR

Group
     build
```

This information includes the full task path, the task type, possible 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 Inspecting 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 Inspecting 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
```

Software Model reports

You can get a hierarchical view of elements for software model projects using the `model` task:

```
$ gradle model
```

Learn more about the model report in the software model documentation.

Command-line completion

Gradle provides bash and zsh tab completion support for tasks, options, and Gradle properties through gradle-completion, installed separately.

Debugging options

`-?`, `-h`, `--help`

Shows a help message with all available CLI options.

`-v`, `--version`

Prints Gradle, Groovy, Ant, JVM, and operating system version information.

`-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.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*.

`--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`

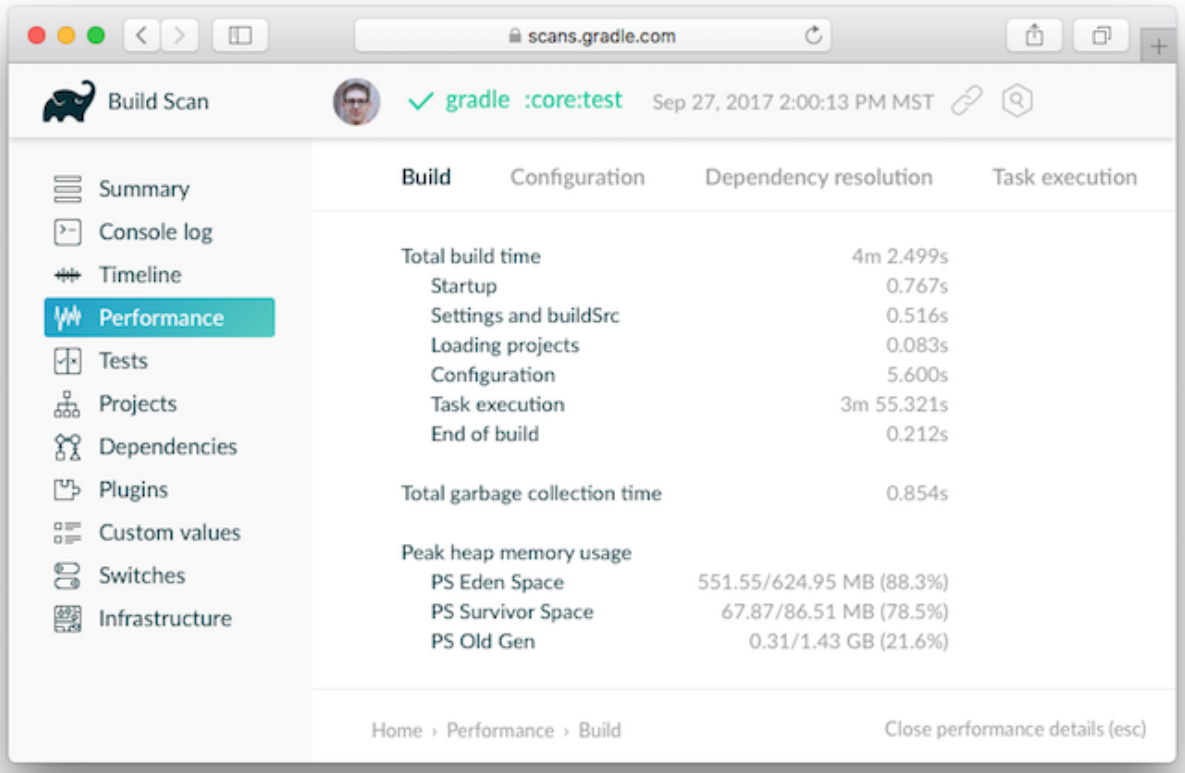
Specifies the scheduling priority for the Gradle daemon and all processes launched by it. Values are `normal` or `low`. *Default is normal*.

`--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.



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,none,summary)`
: Specify warning mode via Gradle properties. Different modes described immediately below.

`--warning-mode=(all,none,summary)`
: Specifies how to log warnings. Default is `summary`.

  Set to `all` to log all 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.

`--dry-run`
: Run Gradle with all task actions disabled. Use this to show which task would have executed.

`--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.

`---no-rebuild`
: Do not rebuild project dependencies. Useful for debugging and fine-tuning `buildSrc`, but can lead to wrong results. Use with caution!

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`
: Specifies the build file. For example: `gradle --build-file=foo.gradle`. The default is `build.gradle`, then `build.gradle.kts`, then `myProjectName.gradle`.

`-c`, `--settings-file`
: 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 System Properties.

`-Dorg.gradle.jvmargs`
: Set JVM arguments.

`-Dorg.gradle.java.home`
: Set JDK home dir.

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.

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.

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 and quirks

There are several issues to be aware with the current implementation of continuous build. These are likely to be addressed in future Gradle releases.

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.

Restrictions with Java 9

Due to class access restrictions related to Java 9, Gradle cannot set some operating system specific options, which means that:

- On macOS, Gradle will poll for file changes every 10 seconds instead of every 2 seconds.
- On Windows, Gradle must use individual file watches (like on Linux/Mac OS), which may cause continuous build to no longer work on very large projects.

Performance and stability

The JDK file watching facility relies on inefficient file system polling on macOS (see: JDK-7133447). This can significantly delay notification of changes on large projects with many source files.

Additionally, the watching mechanism may deadlock under *heavy* load on macOS (see: JDK-8079620). This will manifest as Gradle appearing not to notice file changes. If you suspect this is occurring, exit continuous build and start again.

On Linux, OpenJDK's implementation of the file watch service can sometimes miss file system events (see: JDK-8145981).

Changes to symbolic links

- Creating or removing symbolic link to files will initiate a build.
- Modifying the target of a symbolic link will not cause a rebuild.
- Creating or removing symbolic links to directories will not cause rebuilds.
- Creating new files in the target directory of a symbolic link will not cause a rebuild.
- Deleting the target directory will not cause a rebuild.

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.

Build Environment

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.
- Gradle properties such as `org.gradle.caching=true` that are typically stored in a `gradle.properties` file in a project root directory or `GRADLE_USER_HOME` environment variable.
- Environment variables such as `GRADLE_OPTS` sourced by the environment that executes Gradle.

Aside from configuring the build environment, you can configure a given project build using Project properties such as `-PreleaseType=final`.

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 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.

Setting up a consistent environment for your build is as simple as placing these settings into a `gradle.properties` file. The configuration is applied in following order (if an option is configured in multiple locations the *last one wins*):

- `gradle.properties` in project root directory.
- `gradle.properties` in `GRADLE_USER_HOME` directory.
- system properties, e.g. when `-Dgradle.user.home` is set 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 is much faster builds. Learn more about using the build cache.

`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.

`org.gradle.configureondemand=(true,false)`
: Enables incubating configuration on demand, where Gradle will attempt to configure only necessary projects.

`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.daemon=(true,false)`
: When set to `true` the Gradle Daemon is used to run the build. Default is `true`.

`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.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. A reasonable default is used 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.

`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. The `lifecycle` level is the default. See Choosing a log level.

`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 Gradle performance guide.

`org.gradle.warning.mode=(all,none,summary)`
: When set to `all`, `summary` or `none`, Gradle will use different warning type display. See Command-line logging options for details.

`org.gradle.workers.max=(max # of worker processes)`
: When configured, Gradle will use a maximum of the given number of workers. Default is number of CPU processors. See also performance command-line options.

`org.gradle.priority=(low,normal)`
: Specifies the scheduling priority for the Gradle daemon and all processes launched by it. Default is `normal`. See also performance command-line options.

The following example demonstrates usage of various properties.

Example 29. Setting properties with a gradle.properties file

gradle.properties

```
gradlePropertiesProp=gradlePropertiesValue
sysProp=shouldBeOverWrittenBySysProp
systemProp.system=systemValue
```

build.gradle

```
task printProps {
    doLast {
        println commandLineProjectProp
        println gradlePropertiesProp
        println systemProjectProp
        println System.properties['system']
    }
}
```

build.gradle.kts

```
// Project properties can be accessed via delegation
val commandLineProjectProp: String by project
val gradlePropertiesProp: String by project
val systemProjectProp: String by project

task("printProps") {
    doLast {
        println(commandLineProjectProp)
        println(gradlePropertiesProp)
        println(systemProjectProp)
        println(System.getProperty("system"))
    }
}
```

```
$ gradle -q -PcommandLineProjectProp=commandLineProjectPropValue
-Dorg.gradle.project.systemProjectProp=systemPropertyValue printProps
commandLineProjectPropValue
gradlePropertiesValue
systemPropertyValue
systemValue
```

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.`

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.

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.

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.

Project properties

You can add properties directly to your Project object via the `-P` command line option.

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"`.

Setting a project property via a system property

```
org.gradle.project.foo=bar
```

Setting a project property via an environment variable

```
ORG_GRADLE_PROJECT_foo=bar
```

NOTE	The properties file in the user's home directory has precedence over property files in the project directories.

This feature is very 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.

You can access a project property in your build script simply by using its name as you would use a variable.

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"`

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`

Changing JVM settings for the client VM

```
JAVA_OPTS="-Xmx64m -XX:MaxPermSize=64m -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 30. Set Java compile options for JavaCompile tasks

build.gradle

```
plugins {
    id 'java'
}

tasks.withType(JavaCompile) {
    options.compilerArgs += ['-Xdoclint:none', '-Xlint:none', '-nowarn']
}
```

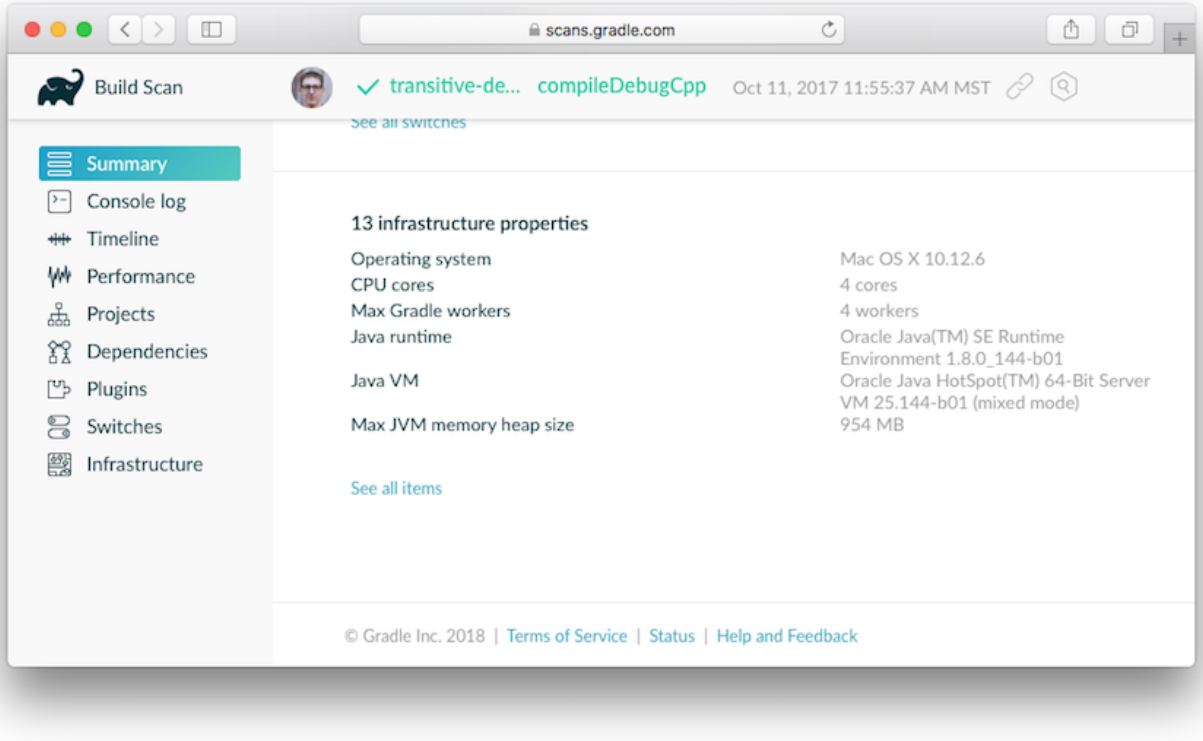
build.gradle.kts

```
plugins {
    java
}

tasks.withType<JavaCompile> {
    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 31. Prevent releasing outside of CI

build.gradle

```
task performRelease {
    doLast {
        if (project.hasProperty("isCI")) {
            println("Performing release actions")
        } else {
            throw new InvalidUserDataException("Cannot perform release
outside of CI")
        }
    }
}
```

build.gradle.kts

```
task("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 HTTP proxy

Configuring an HTTP or HTTPS 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.

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.

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
systemProp.https.nonProxyHosts=*.nonproxyrepos.com|localhost
```

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 7 Networking Properties

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.

Directory Layout

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
├─── wrapper
│    └─── dists ⑥
│         ├─── ⋮
│         ├─── gradle-4.8-bin
│         ├─── gradle-4.9-all
│         └─── gradle-4.9-bin
└─── gradle.properties ⑦
```

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 Distributions downloaded by the Gradle Wrapper

⑦ 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
├──── build.gradle or build.gradle.kts ⑤
├──── gradle.properties ⑥
├──── gradlew ⑦
├──── gradlew.bat ⑦
└──── settings.gradle or settings.gradle.kts ⑧
```

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 The project's Gradle build script

⑥ Project-specific Gradle configuration properties

⑦ Scripts for executing builds using the Gradle Wrapper

⑧ The project's settings file

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.

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 that require a non-trivial initialization time. As a result, it can sometimes seem a little slow to start. The solution to this problem is the Gradle *Daemon*: a long-lived background process that executes your builds much more quickly than would otherwise be the case. We accomplish this by avoiding the expensive bootstrapping process as well as leveraging caching, by keeping data about your project in memory. Running Gradle builds with the Daemon is no different than without. Simply configure whether you want to use it or not - everything else is handled transparently by Gradle.

Why the Gradle Daemon is important for performance

The Daemon is a long-lived process, so not only are we able to avoid the cost of JVM startup for every build, but we are able to cache information about project structure, files, tasks, and more in memory.

The reasoning is simple: improve build speed by reusing computations from previous builds. However, the benefits are dramatic: we typically measure build times reduced by 15-75% on subsequent builds. We recommend profiling your build by using `--profile` to get a sense of how much impact the Gradle Daemon can have for you.

The Gradle Daemon is enabled by default starting with Gradle 3.0, so you don't have to do anything to benefit from it.

If you run CI builds in ephemeral environments (such as containers) that do not reuse any processes, use of the Daemon will slightly decrease performance (due to caching additional information) for no benefit, and may be disabled.

Running Daemon Status

To get a list of running Gradle Daemons and their statuses use the `--status` command.

Sample output:

```
  PID VERSION              STATUS
28411 3.0                  IDLE
34247 3.0                  BUSY
```

Currently, a given Gradle version can only connect to daemons of the same version. This means the status output will only show Daemons for the version of Gradle being invoked and not for any other versions. Future versions of Gradle will lift this constraint and will show the running Daemons for all versions of Gradle.

Disabling the Daemon

The Gradle Daemon is enabled by default, and we recommend always enabling it. There are several ways to disable the Daemon, but the most common one is to add the line

```
org.gradle.daemon=false
```

to the file `«USER_HOME»/.gradle/gradle.properties`, where `«USER_HOME»` is your home directory. That's typically one of the following, depending on your platform:

- `C:\Users\<username>` (Windows Vista & 7+)
- `/Users/<username>` (macOS)
- `/home/<username>` (Linux)

If that file doesn't exist, just create it using a text editor. You can find details of other ways to disable (and enable) the Daemon in Daemon FAQ further down. That section also contains more detailed information on how the Daemon works.

Note that having the Daemon enabled, all your builds will take advantage of the speed boost, regardless of the version of Gradle a particular build uses.

TIP

Continuous integration

Since Gradle 3.0, we enable Daemon by default and recommend using it for both developers' machines and Continuous Integration servers. However, if you suspect that Daemon makes your CI builds unstable, you can disable it to use a fresh runtime for each build since the runtime is *completely* isolated from any previous builds.

Stopping an existing Daemon

As mentioned, the Daemon is a background process. You needn't worry about a build up of Gradle processes on your machine, though. Every Daemon monitors its memory usage compared to total system memory and will stop itself if idle when available system memory is low. If you want to explicitly stop running Daemon processes for any reason, just use the command `gradle --stop`.

This will terminate all Daemon processes that were started with the same version of Gradle used to execute the command. If you have the Java Development Kit (JDK) installed, you can easily verify that a Daemon has stopped by running the `jps` command. You'll see any running Daemons listed with the name `GradleDaemon`.

FAQ

How do I disable the Gradle Daemon?

There are two recommended ways to disable the Daemon persistently for an environment:

- Via environment variables: add the flag `-Dorg.gradle.daemon=false` to the `GRADLE_OPTS` environment variable

- Via properties file: add `org.gradle.daemon=false` to the `«GRADLE_USER_HOME»/gradle.properties` file

NOTE	Note, `«GRADLE_USER_HOME»` defaults to `«USER_HOME»/.gradle`, where `«USER_HOME»` is the home directory of the current user. This location can be configured via the `-g` and `--gradle-user-home` command line switches, as well as by the `GRADLE_USER_HOME` environment variable and `org.gradle.user.home` JVM system property.

Both approaches have the same effect. Which one to use is up to personal preference. Most Gradle users choose the second option and add the entry to the user `gradle.properties` file.

On Windows, this command will disable 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 will disable the Daemon for the current user:

```
mkdir -p ~/.gradle && echo "org.gradle.daemon=false" >> ~/.gradle/gradle.properties
```

Once the Daemon is disabled for a build environment in this way, a Gradle Daemon will not be started unless explicitly requested using the `--daemon` option.

The `--daemon` and `--no-daemon` command line options enable and disable usage of the Daemon for individual build invocations when using the Gradle command line interface. These command line options have the *highest* precedence when considering the build environment. Typically, it is more convenient to enable the Daemon for an environment (e.g. a user account) so that all builds use the Daemon without requiring to remember to supply the `--daemon` option.

Why is there more than one Daemon process on my machine?

There are several reasons why Gradle will create a new Daemon, instead of using one that is already running. The basic rule is that Gradle will start a new Daemon if there are no existing idle or compatible Daemons available. Gradle will kill any Daemon that has been idle for 3 hours or more, so you don't have to worry about cleaning them up manually.

idle

An idle Daemon is one that is not currently executing a build or doing other useful work.

compatible

A compatible Daemon is one that can (or can be made to) meet the requirements of the requested build environment. The Java runtime used to execute the build is an example aspect of the build environment. Another example is the set of JVM system properties required by the build runtime.

Some aspects of the requested build environment may not be met by an Daemon. If the Daemon is running with a Java 8 runtime, but the requested environment calls for Java 10, then the Daemon is not compatible and another must be started. Moreover, certain properties of a Java runtime cannot be changed once the JVM has started. For example, it is not possible to change the memory allocation (e.g. `-Xmx1024m`), default text encoding, default locale, etc of a running JVM.

The "requested build environment" is typically constructed implicitly from aspects of the build client's (e.g. Gradle command line client, IDE etc.) environment and explicitly via command line switches and settings. See Build Environment for details on how to specify and control the build environment.

The following JVM system properties are effectively immutable. If the requested build environment requires any of these properties, with a different value than a Daemon's JVM has for this property, the Daemon is not compatible.

- 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 effectively immutable. The corresponding attributes of the requested build environment and the Daemon's environment must match exactly in order for a Daemon to be compatible.

- The maximum heap size (i.e. the -Xmx JVM argument)
- The minimum heap size (i.e. the -Xms JVM argument)
- The boot classpath (i.e. the -Xbootclasspath argument)
- The "assertion" status (i.e. the -ea argument)

The required Gradle version is another aspect of the requested build environment. Daemon processes are coupled to a specific Gradle runtime. Working on multiple Gradle projects during a session that use different Gradle versions is a common reason for having more than one running Daemon process.

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 will use up to 1GB of heap. It will use the JVM's default minimum heap size. 1GB is more than enough for most builds. Larger builds with hundreds of subprojects, lots of configuration, and source code may require, or perform better, with more memory.

To increase the amount of memory the Daemon can use, specify the appropriate flags as part of the requested build environment. Please see Build Environment for details.

How can I stop a Daemon?

Daemon processes will automatically terminate themselves after 3 hours of inactivity or less. If you wish to stop a Daemon process before this, you can either kill the process via your operating system or run the `gradle --stop` command. The `--stop` switch causes Gradle to request that *all* running Daemon processes, *of the same Gradle version used to run the command*, terminate themselves.

What can go wrong with Daemon?

Considerable engineering effort has gone into making the Daemon robust, transparent and unobtrusive during day to day development. However, Daemon processes can occasionally be corrupted or exhausted. A Gradle build executes arbitrary code from multiple sources. While Gradle itself is designed for and heavily tested with the Daemon, user build scripts and third party plugins can destabilize the Daemon process through defects such as memory leaks or global state corruption.

It is also possible to destabilize the Daemon (and build environment in general) by running builds that do not release resources correctly. This is a particularly poignant problem when using Microsoft Windows as it is less forgiving of programs that fail to close files after reading or writing.

Gradle actively monitors heap usage and attempts to detect when a leak is starting to exhaust the available heap space in the daemon. When it detects a problem, the Gradle daemon will finish the currently running build and proactively restart the daemon on the next build. This monitoring is enabled by default, but can be disabled by setting the `org.gradle.daemon.performance.enable-monitoring` system property to false.

If it is suspected that the Daemon process has become unstable, it can simply be killed. Recall that the `--no-daemon` switch can be specified for a build to prevent use of the Daemon. This can be useful to diagnose whether or not the Daemon is actually the culprit of a problem.

Tools & IDEs

The Gradle Tooling API that is used by IDEs and other tools to integrate with Gradle *always* uses the Gradle Daemon to execute builds. If you are executing Gradle builds from within your IDE you are using the Gradle Daemon and do not need to enable it for your environment.

How does the Gradle Daemon make builds faster?

The Gradle Daemon is a *long lived* build process. In between builds it waits idly for the next build. This has the obvious benefit of only requiring Gradle to be loaded into memory once for multiple

builds, as opposed to once for each build. This in itself is a significant performance optimization, but that's not where it stops.

A significant part of the story for modern JVM performance is runtime code optimization. For example, HotSpot (the JVM implementation provided by Oracle and used as the basis of OpenJDK) applies optimization to code while it is running. The optimization is progressive and not instantaneous. That is, the code is progressively optimized during execution which means that subsequent builds can be faster purely due to this optimization process. Experiments with HotSpot have shown that it takes somewhere between 5 and 10 builds for optimization to stabilize. The difference in perceived build time between the first build and the 10th for a Daemon can be quite dramatic.

The Daemon also allows more effective in memory caching across builds. For example, the classes needed by the build (e.g. plugins, build scripts) can be held in memory between builds. Similarly, Gradle can maintain in-memory caches of build data such as the hashes of task inputs and outputs, used for incremental building.

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.

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 32. Using init script to perform extra configuration before projects are evaluated

build.gradle

```
repositories {
    mavenCentral()
}

task showRepos {
    doLast {
        println "All repos:"
        println repositories.collect { it.name }
    }
}
```

init.gradle

```
allprojects {
    repositories {
        mavenLocal()
    }
}
```

build.gradle.kts

```
repositories {
    mavenCentral()
}

task("showRepos") {
    doLast {
        println("All repos:")
        //TODO:kotlin-dsl remove filter once we're no longer on a kotlin eap
        println(repositories.map { it.name }.filter { it != "maven" })
    }
}
```

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 33. 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 34. 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)
```

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))
```

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 35. 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

```
/*
 * Copyright 2013 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

// tag::show-repos-task[]
repositories{
    mavenCentral()
}

 task showRepositories {
     doLast {
         repositories.each {
             println "repository: ${it.name} ('${it.url}')"
         }
     }
}
// end::show-repos-task[]
```

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

```
// tag::show-repos-task[]
repositories{
    mavenCentral()
}

task("showRepositories") {
    doLast {
        repositories.map { it as MavenArtifactRepository }.forEach {
            println("repository: ${it.name} ('${it.url}')")
        }
    }
}
// end::show-repos-task[]
```

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.

Structure of a multi-project build

Such builds come in all shapes and sizes, but they do have some common characteristics:

- A `settings.gradle` file in the root or `master` directory of the project

- A `build.gradle` file in the root or `master` directory
- Child directories that have their own `*.gradle` build files (some multi-project builds may omit child project build scripts)

The `settings.gradle` file tells Gradle how the project and subprojects are structured. Fortunately, you don't have to read this file simply to learn what the project structure is as you can run the command `gradle projects`. Here's the output from using that command on the Java *multiproject* build in the Gradle samples:

Example: Listing the projects in a build

Output of `gradle -q projects`

```
> gradle -q projects

------------------------------------------------------------
Root project
------------------------------------------------------------

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
```

This tells you that *multiproject* has three immediate child projects: *api*, *services* and *shared*. The *services* project then has its own children, *shared* and *webservice*. These map to the directory structure, so it's easy to find them. For example, you can find *webservice* in `<root>/services/webservice`.

By default, Gradle uses the name of the directory it finds the `settings.gradle` as the name of the root project. This usually doesn't cause problems since all developers check out the same directory name when working on a project. On Continuous Integration servers, like Jenkins, the directory name may be auto-generated and not match the name in your VCS. For that reason, it's recommended that you always set the root project name to something predictable, even in single project builds. You can configure the root project name by setting `rootProject.name`.

Each project will usually have its own build file, but that's not necessarily the case. In the above example, the *services* project is just a container or grouping of other subprojects. There is no build file in the corresponding directory. However, *multiproject* does have one for the root project.

The root `build.gradle` is often used to share common configuration between the child projects, for example by applying the same sets of plugins and dependencies to all the child projects. It can also be used to configure individual subprojects when it is preferable to have all the configuration in one place. This means you should always check the root build file when discovering how a

particular subproject is being configured.

Another thing to bear in mind is that the build files might not be called `build.gradle`. Many projects will name the build files after the subproject names, such as `api.gradle` and `services.gradle` from the previous example. Such an approach helps a lot in IDEs because it's tough to work out which `build.gradle` file out of twenty possibilities is the one you want to open. This little piece of magic is handled by the `settings.gradle` file, but as a build user you don't need to know the details of how it's done. Just have a look through the child project directories to find the files with the `.gradle` suffix.

Once you know what subprojects are available, the key question for a build user is how to execute the tasks within the project.

Executing a multi-project build

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. You have two options here:

- Change to the directory corresponding to the subproject you're interested in and just execute `gradle <task>` as normal.
- Use a qualified task name from any directory, although this is usually done from the root. For example: `gradle :services:webservice:build` will build the *webservice* subproject and any subprojects it depends on.

The first approach is similar to the single-project use case, but Gradle works slightly differently in the case of a multi-project build. The command `gradle test` will execute the `test` task in any subprojects, relative to the current working directory, that have that task. So 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*.

For more control over what gets executed, use qualified names (the second approach mentioned). These are paths just like directory paths, but use ':' instead of '/' or '\'. If the path begins with a ':', then the path is resolved relative to the root project. In other words, the leading ':' represents the root project itself. All other colons are path separators.

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 guide.

There's one last thing to note. When you're using the Gradle wrapper, the first approach 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 you're in the *webservice* subproject directory, you would have to run `../../gradlew build`.

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.

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.

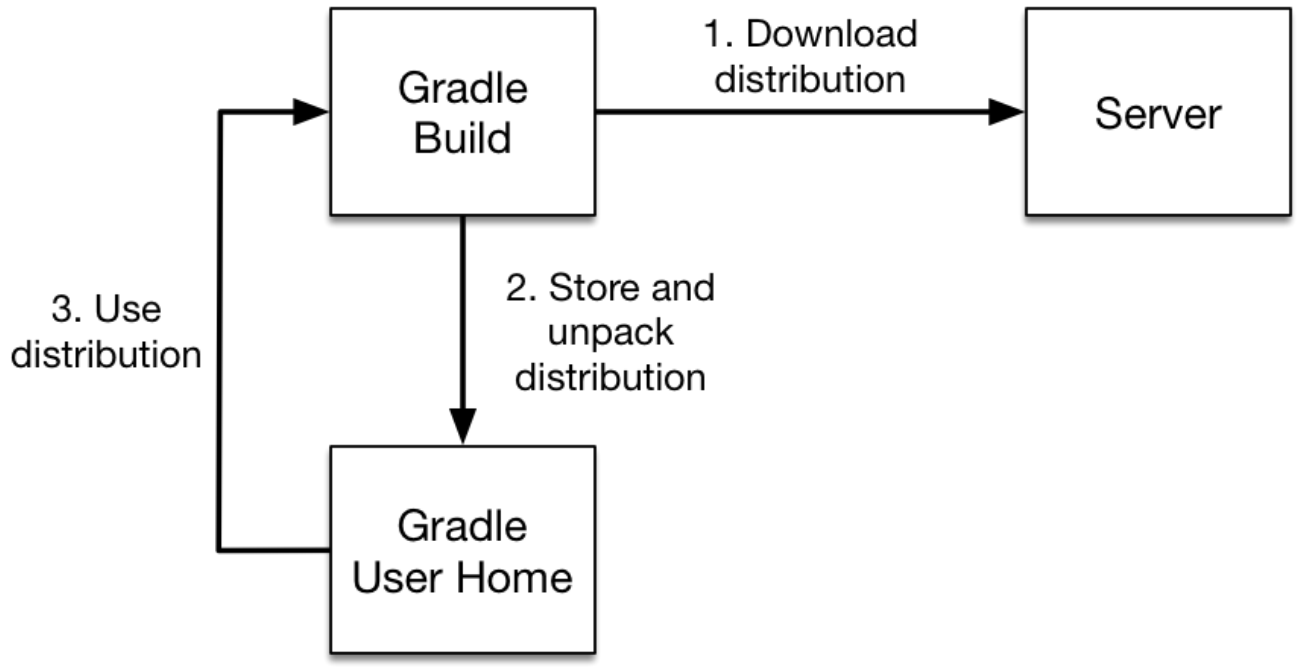


Figure 3. The Wrapper workflow

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.

`gradle/wrapper/gradle-wrapper.properties`

```
distributionUrl=https\://services.gradle.org/distributions/gradle-5.0-rc-2-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.

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 5.0-rc-2 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 5.0-rc-2 --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-5.0-rc-2-all.zip
```

Let's have a look at the following project layout to illustrate the expected Wrapper files:

```
.
├─── build.gradle
├─── settings.gradle
├─── gradle
│    └─── wrapper
│         ├─── gradle-wrapper.jar
│         └─── gradle-wrapper.properties
├─── gradlew
└─── gradlew.bat
```

A Gradle project typically provides a `build.gradle` and a `settings.gradle` file. 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.

`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 property 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'd commit the changes to the Wrapper files to version control.

Use the Gradle `wrapper` task to generate the wrapper, specifying a version. The default is the current version, which you can check by executing `./gradlew --version`.

Example: Upgrading the Wrapper version

```
$ ./gradlew wrapper --gradle-version 5.0-rc-2

BUILD SUCCESSFUL in 4s
1 actionable task: 1 executed
```

Checking the Wrapper version after upgrading

```
$ ./gradlew -v
Downloading https://services.gradle.org/distributions/gradle-4.2.1-bin.zip
.............................................................
Unzipping /Users/claudia/.gradle/wrapper/dists/gradle-4.2.1-
bin/dajvke9o8kmaxbu0kc5gcgeju/gradle-4.2.1-bin.zip to
/Users/claudia/.gradle/wrapper/dists/gradle-4.2.1-bin/dajvke9o8kmaxbu0kc5gcgeju
Set executable permissions for: /Users/claudia/.gradle/wrapper/dists/gradle-4.2.1-
bin/dajvke9o8kmaxbu0kc5gcgeju/gradle-4.2.1/bin/gradle

------------------------------------------------------------
Gradle 4.2.1
------------------------------------------------------------

Build time:   2017-10-02 15:36:21 UTC
Revision:     a88ebd6be7840c2e59ae4782eb0f27fbe3405ddf

Groovy:       2.4.12
Ant:          Apache Ant(TM) version 1.9.6 compiled on June 29 2015
JVM:          1.8.0_60 (Oracle Corporation 25.60-b23)
OS:           Mac OS X 10.13.1 x86_64
```

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 36. Customizing the Wrapper task

build.gradle

```
wrapper {
    distributionType = Wrapper.DistributionType.ALL
}
```

build.gradle.kts

```
tasks.wrapper {
    distributionType = Wrapper.DistributionType.ALL
}
```

With the configuration in place running `./gradlew wrapper --gradle-version 5.0-rc-2` 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-5.0-rc-2-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.

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.

Troubleshooting

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 4.6
------------------------------------------------------------

Build time:   2018-02-21 15:28:42 UTC
Revision:     819e0059da49f469d3e9b2896dc4e72537c4847d

Groovy:       2.4.12
Ant:          Apache Ant(TM) version 1.9.9 compiled on February 2 2017
JVM:          1.8.0_151 (Oracle Corporation 25.151-b12)
OS:           Mac OS X 10.13.3 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.

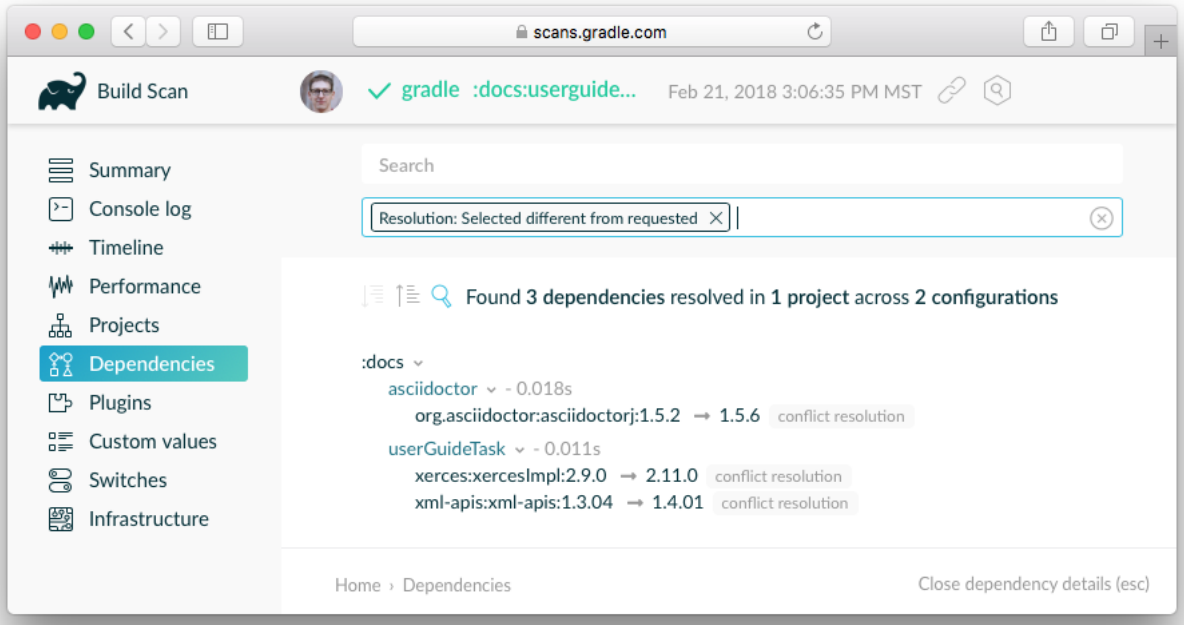


Figure 4. Debugging dependency conflicts with build scans

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

Troubleshooting slow Gradle builds

For build performance issues (including “slow sync time”), see the guide to 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.

```
gradle help -Dorg.gradle.debug=true --no-daemon
```

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

NOTE	You must either stop running Gradle Daemons or run with `--no-daemon` when using debug mode.

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.

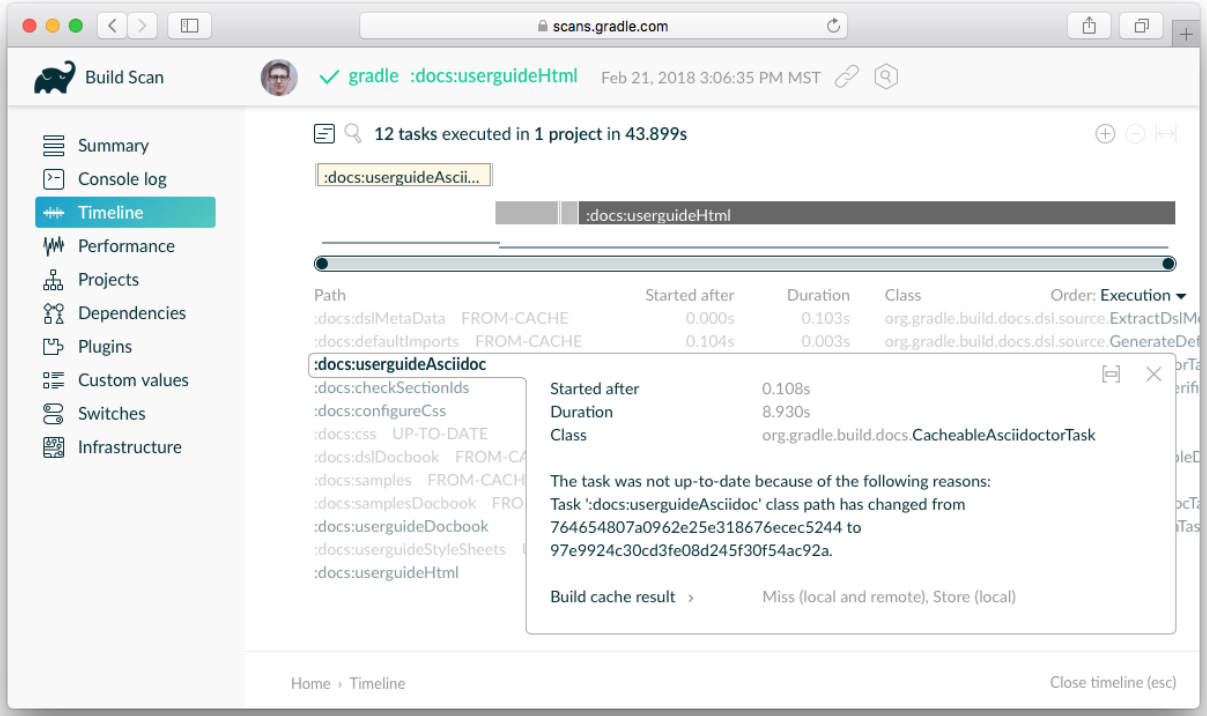


Figure 5. Debugging incremental build with a build scan

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.

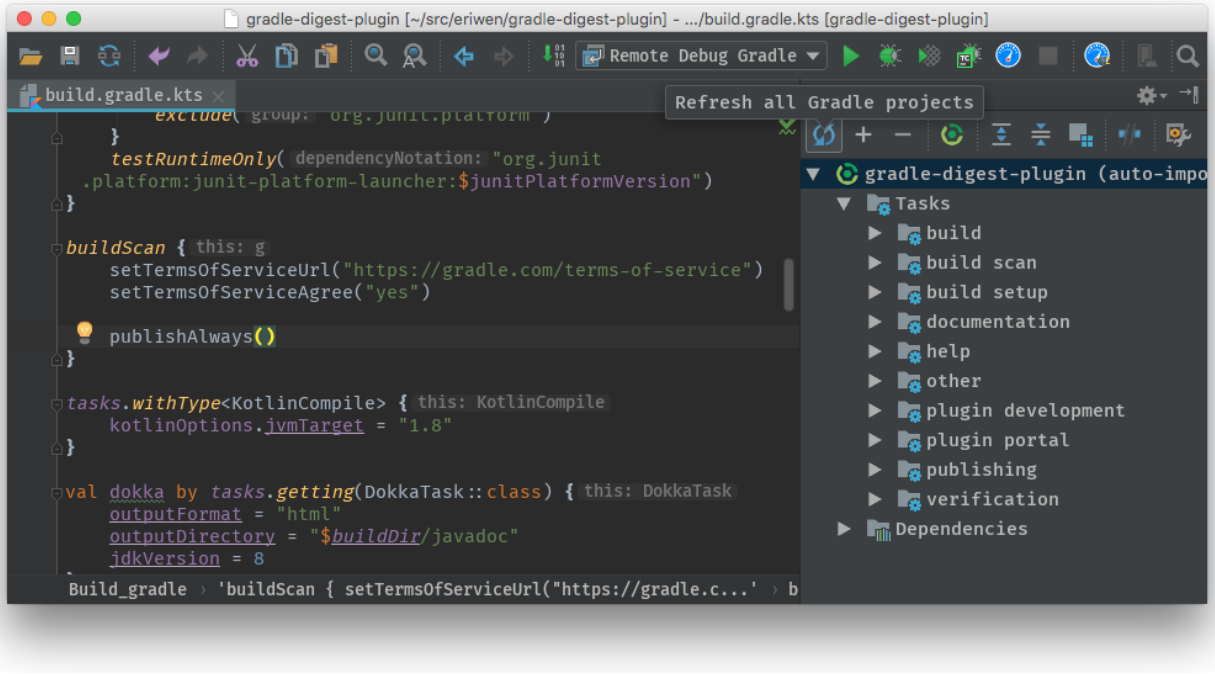


Figure 6. Refreshing a Gradle project in IntelliJ IDEA

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.

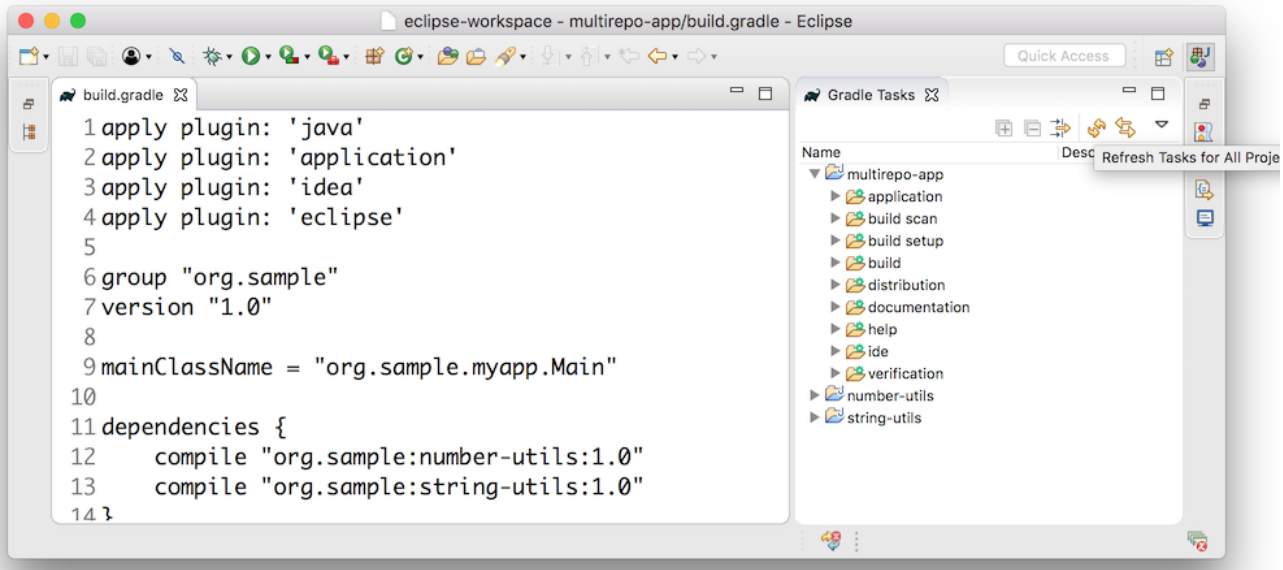


Figure 7. Refreshing a Gradle project in Eclipse Buildship

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.

Authoring Gradle Builds

The Feature Lifecycle

Gradle is under constant development and improvement. New versions are delivered on a regular and frequent basis (approximately every 6 weeks). Continuous improvement combined with frequent delivery allows new features to be made available to users early and for invaluable real world feedback to be incorporated into the development process. Getting new functionality into the hands of users regularly is a core value of the Gradle platform. At the same time, API and feature stability is taken very seriously and is also considered a core value of the Gradle platform. This is something that is engineered into the development process by design choices and automated testing, and is formalised by the section on backwards compatibility.

The Gradle *feature lifecycle* has been designed to meet these goals. It also serves to clearly communicate to users of Gradle what the state of a feature is. The term *feature* typically means an API or DSL method or property in this context, but it is not restricted to this definition. Command line arguments and modes of execution (e.g. the Build Daemon) are two examples of other kinds of features.

States

Features can be in one of 4 states:

- Internal
- Incubating
- Public
- Deprecated

Internal

Internal features are not designed for public use and are only intended to be used by Gradle itself. They can change in any way at any point in time without any notice. Therefore, we recommend avoiding the use of such features. Internal features are not documented. If it appears in this User Guide, the DSL Reference or the API Reference documentation then the feature is not internal.

Internal features may evolve into public features.

Incubating

Features are introduced in the *incubating* state to allow real world feedback to be incorporated into the feature before it is made public and locked down to provide backwards compatibility. It also gives users who are willing to accept potential future changes early access to the feature so they can put it into use immediately.

A feature in an incubating state may change in future Gradle versions until it is no longer incubating. Changes to incubating features for a Gradle release will be highlighted in the release notes for that release. The incubation period for new features varies depending on the scope,

complexity and nature of the feature.

Features in incubation are clearly indicated to be so. In the source code, all methods/properties/classes that are incubating are annotated with Incubating, which is also used to specially mark them in the DSL and API references. If an incubating feature is discussed in this User Guide, it will be explicitly said to be in the incubating state.

Public

The default state for a non-internal feature is *public*. Anything that is documented in the User Guide, DSL Reference or API references that is not explicitly said to be incubating or deprecated is considered public. Features are said to be *promoted* from an incubating state to public. The release notes for each release indicate which previously incubating features are being promoted by the release.

A public feature will *never* be removed or intentionally changed without undergoing deprecation. All public features are subject to the backwards compatibility policy.

Deprecated

Some features will become superseded or irrelevant due to the natural evolution of Gradle. Such features will eventually be removed from Gradle after being *deprecated*. A deprecated feature will *never* be changed, until it is finally removed according to the backwards compatibility policy.

Deprecated features are clearly indicated to be so. In the source code, all methods/properties/classes that are deprecated are annotated with "`@java.lang.Deprecated`" which is reflected in the DSL and API references. In most cases, there is a replacement for the deprecated element, and this will be described in the documentation. Using a deprecated feature will also result in a runtime warning in Gradle's output.

Use of deprecated features should be avoided. The release notes for each release indicate any features that are being deprecated by the release.

Backwards Compatibility Policy

Gradle provides backwards compatibility across major versions (e.g. `1.x`, `2.x`, etc.). Once a public feature is introduced or promoted in a Gradle release it will remain indefinitely or until it is deprecated. Once deprecated, it may be removed in the next major release. Deprecated features may be supported across major releases, but this is not guaranteed.

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 testability also becomes a valid option.

Example 37. A build script using conditional logic to create a task

build.gradle

```
if (project.findProperty('releaseEngineer') != null) {
    task release {
        doLast {
            logger.quiet 'Releasing to production...'

            // release the artifact to production
        }
    }
}
```

build.gradle.kts

```
if (project.findProperty("releaseEngineer") != null) {
    task("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 38. 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;

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 39. 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 guide.

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.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 40. Definition of tasks following best practices

build.gradle

```
import com.enterprise.DocsGenerate

task generateHtmlDocs(type: DocsGenerate) {
    group = JavaBasePlugin.DOCUMENTATION_GROUP
    description = 'Generates the HTML documentation for this project.'
    title = 'Project docs'
    outputDir = file("$buildDir/docs")
}

task 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

task<DocsGenerate>("generateHtmlDocs") {
    group = JavaBasePlugin.DOCUMENTATION_GROUP
    description = "Generates the HTML documentation for this project."
    title = "Project docs"
    outputDir = file("$buildDir/docs")
}

task("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 41. A task declaring the group and description

build.gradle

```
task generateDocs {
    group = 'Documentation'
    description = 'Generates the HTML documentation for this project.'

    doLast {
        // action implementation
    }
}
```

build.gradle.kts

```
task("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 42. Executing logic during configuration should be avoided

build.gradle

```
dependencies {
    implementation 'log4j:log4j:1.2.17'
}

task printArtifactNames {
    // always executed
    def libraryNames = configurations.compileClasspath.collect { it.name }

    doLast {
        logger.quiet libraryNames
    }
}
```

build.gradle.kts

```
dependencies {
    implementation("log4j:log4j:1.2.17")
}

task("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 43. Executing logic during execution phase is preferred

build.gradle

```
dependencies {
    implementation 'log4j:log4j:1.2.17'
}

task printArtifactNames {
    doLast {
        def libraryNames = configurations.compileClasspath.collect { it.name
}
        logger.quiet libraryNames
    }
}
```

build.gradle.kts

```
dependencies {
    implementation("log4j:log4j:1.2.17")
}

task("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.

Avoid storing passwords in plain text

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 a `gradle.properties` file. Those files usually live in a version control repository and can be viewed by anyone that has access to it.

Passwords should be stored in encrypted fashion. 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.

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
├─── settings.gradle
└─── src
    └─── main
        ├─── java
        │    └─── HelloWorld.java
        └─── kotlin
            └─── Utils.kt
```

```
.
├─── build.gradle.kts
├─── settings.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.

The following source tree demonstrates how to separate unit from integration tests in a Java-based project.

```
.
├─── build.gradle
├─── gradle
│    └─── integration-test.gradle
├─── settings.gradle
└─── src
     ├─── integTest
     │    └─── java
     │         └─── DefaultFileReaderIntegrationTest.java
     ├─── main
     │    └─── java
     │         ├─── DefaultFileReader.java
     │         ├─── FileReader.java
     │         └─── StringUtils.java
     └─── test
          └─── java
               └─── StringUtilsTest.java
```

```
.
├─── build.gradle.kts
├─── gradle
│    └─── integration-test.gradle.kts
├─── settings.gradle.kts
└─── src
     ├─── integTest
     │    └─── java
     │         └─── DefaultFileReaderIntegrationTest.java
     ├─── main
     │    └─── java
     │         ├─── DefaultFileReader.java
     │         ├─── FileReader.java
     │         └─── StringUtils.java
     └─── test
          └─── java
               └─── StringUtilsTest.java
```

Gradle models source code directories with the help of the source set concept. By pointing an instance of a source set to one or many source code directories, Gradle will automatically create a corresponding compilation task out-of-the-box.

Example 44. Integration test source set

gradle/integration-test.gradle

```
sourceSets {
    integTest {
        java.srcDir file('src/integTest/java')
        resources.srcDir file('src/integTest/resources')
        compileClasspath += sourceSets.main.output + configurations
.testRuntimeClasspath
        runtimeClasspath += output + compileClasspath
    }
}
```

gradle/integration-test.gradle.kts

```
sourceSets {
    create("integTest") {
        java.srcDir(file("src/integTest/java"))
        resources.srcDir(file("src/integTest/resources"))
        compileClasspath += sourceSets["main"].output +
configurations["testRuntimeClasspath"]
        runtimeClasspath += output + compileClasspath
    }
}
```

Source sets are only responsible for compiling source code, but do not deal with executing the byte code. For the purpose of test execution, a corresponding task of type Test needs to be established.

Example 45. Integration test task

gradle/integration-test.gradle

```
task integTest(type: Test) {
    description = 'Runs the integration tests.'
    group = 'verification'
    testClassesDirs = sourceSets.integTest.output.classesDirs
    classpath = sourceSets.integTest.runtimeClasspath
    mustRunAfter test
}

check.dependsOn integTest
```

gradle/integration-test.gradle.kts

```
task<Test>("integTest") {
    description = "Runs the integration tests."
    group = "verification"
    testClassesDirs = sourceSets["integTest"].output.classesDirs
    classpath = sourceSets["integTest"].runtimeClasspath
    mustRunAfter(tasks["test"])
}

tasks["check"].dependsOn("integTest")
```

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. This recommendation applies to single project builds as well as multi-project builds. The file can either be empty or define the desired name of the project.

A typical Gradle project with a settings file look as such:

```
.
├─── build.gradle
└─── settings.gradle
```

```
.
├─── build.gradle.kts
└─── settings.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 46. Custom buildSrc build script

buildSrc/build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'junit:junit:4.12'
}
```

buildSrc/build.gradle.kts

```
repositories {
    mavenCentral()
}

dependencies {
    testImplementation("junit:junit:4.12")
}
```

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).

```
.
├── build.gradle
├── buildSrc
│   ├── build.gradle
│   └── src
│       ├── main
│       │   └── java
│       │       └── com
│       │           └── enterprise
│       │               ├── Deploy.java
│       │               └── DeploymentPlugin.java
│       └── test
│           └── java
│               └── com
│                   └── enterprise
│                       └── DeploymentPluginTest.java
└── settings.gradle
```

```
.
├── 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
```

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 define 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.

```
.
├─── build.gradle
├─── gradle.properties
└─── settings.gradle
```

```
.
├─── build.gradle.kts
├─── gradle.properties
└─── settings.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.

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.

You can find a sample project that covers steps one to three in the `samples` directory of the standard `-all` Gradle distribution.

Build Cache

NOTE The build cache feature described here is different from the Android plugin build cache.

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 task output caching, we expect other features to use the build cache in the future.

NOTE A complete guide is available about using the build cache. It covers the different scenarios caching can improve, and 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, FindBugs, JDepend, Pmd
- JaCoCo: JacocoMerge, JacocoReport
- Other tasks: AntlrTask, ValidateTaskProperties, 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.

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 link:https://docs.gradle.org/5.0-rc-2/javadoc/org/gradle/api/tasks/CacheableTask.html] 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 47. Custom cacheable BundleTask

build.gradle

```
@CacheableTask                                          ①
class BundleTask extends NpmTask {

    @Override @Internal                                 ②
    ListProperty<String> getArgs() {
        super.getArgs()
    }

    @InputDirectory
    @SkipWhenEmpty
    @PathSensitive(PathSensitivity.RELATIVE)            ③
    final DirectoryProperty scripts = project.objects.directoryProperty()

    @InputFiles
    @PathSensitive(PathSensitivity.RELATIVE)            ④
    final ConfigurableFileCollection configFiles = project.files()

    @OutputFile
    final RegularFileProperty bundle = project.objects.fileProperty()

    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"))
    }
}

task bundle(type: BundleTask)
```

build.gradle.kts

```
@CacheableTask                                          ①
open class BundleTask : NpmTask() {

    @get:Internal                                       ②
    override val args
        get() = super.args


    @get:InputDirectory
    @get:SkipWhenEmpty
    @get:PathSensitive(PathSensitivity.RELATIVE)        ③
    val scripts: DirectoryProperty = project.objects.directoryProperty()

    @get:InputFiles
    @get:PathSensitive(PathSensitivity.RELATIVE)        ④
    val configFiles: ConfigurableFileCollection = project.files()

    @get:OutputFile
    val bundle: RegularFileProperty = project.objects.fileProperty()

    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"))
    }
}

val bundle by tasks.creating(BundleTask::class)
```

- (1) Add `@Cacheable` 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 48. Make the bundle task cacheable

build.gradle

```
task bundle(type: 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

```
task<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.

For more details on the configuration options refer to the DSL documentation of DirectoryBuildCache. Here is an example of the configuration.

Example 49. Configure the local cache

settings.gradle

```
buildCache {
    local(DirectoryBuildCache) {
        directory = new File(rootDir, 'build-cache')
        removeUnusedEntriesAfterDays = 30
    }
}
```

settings.gradle.kts

```
buildCache {
    local<DirectoryBuildCache> {
        directory = File(rootDir, "build-cache")
        removeUnusedEntriesAfterDays = 30
    }
}
```

Remote HTTP build cache

Gradle has built-in support for connecting to a remote build cache backend via HTTP. For more details on what the protocol looks like see HttpBuildCache. Note that by using 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 50. 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/")
    }
}
```

You can configure the credentials the HttpBuildCache uses to access the build cache server as shown in the following example.

Example 51. Configure remote HTTP cache

settings.gradle

```
buildCache {
    remote(HttpBuildCache) {
        url = 'http://example.com:8123/cache/'
        credentials {
            username = 'build-cache-user'
            password = 'some-complicated-password'
        }
    }
}
```

settings.gradle.kts

```
buildCache {
    remote<HttpBuildCache> {
        url = uri("http://example.com:8123/cache/")
        credentials {
            username = "build-cache-user"
            password = "some-complicated-password"
        }
    }
}
```

NOTE

You may encounter problems with an untrusted SSL certificate when you try to use a build cache backend with an HTTPS URL. The ideal solution is for someone to add a valid SSL certificate to the build cache backend, but we recognize that you may not be able to do that. In that case, set HttpBuildCache.isAllowUntrustedServer() to `true`.

This is a convenient workaround, but you shouldn't use it as a long-term solution.

Example 52. 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
    }
}
```

Configuration use cases

The recommended use case for the build cache is that your continuous integration server populates the remote build cache from clean builds while developers load from the remote build cache and store in the local build cache. The configuration would then look as follows.

Example 53. Recommended setup for CI push use case

settings.gradle

```
boolean isCiServer = System.getenv().containsKey("CI")

buildCache {
    local {
        enabled = !isCiServer
    }
    remote(HttpBuildCache) {
        url = 'https://example.com:8123/cache/'
        push = isCiServer
    }
}
```

settings.gradle.kts

```
val isCiServer = System.getenv().containsKey("CI")

buildCache {
    local {
        isEnabled = !isCiServer
    }
    remote<HttpBuildCache> {
        url = uri("https://example.com:8123/cache/")
        isPush = isCiServer
    }
}
```

If you use a `buildSrc` directory, you should make sure that it uses the same build cache configuration as the main build. This can be achieved by applying the same script to `buildSrc/settings.gradle` and `settings.gradle` as shown in the following example.

Example 54. Consistent setup for buildSrc and main build

settings.gradle

```
apply from: new File(settingsDir, 'gradle/buildCacheSettings.gradle')
```

gradle/buildCacheSettings.gradle

```
boolean isCiServer = System.getenv().containsKey("CI")

buildCache {
    local {
        enabled = !isCiServer
    }
    remote(HttpBuildCache) {
        url = 'https://example.com:8123/cache/'
        push = isCiServer
    }
}
```

buildSrc/settings.gradle

```
apply from: new File(settingsDir, '../gradle/buildCacheSettings.gradle')
```

settings.gradle.kts

```
apply(from = File(settingsDir, "gradle/buildCacheSettings.gradle.kts"))
```

gradle/buildCacheSettings.gradle.kts

```
val isCiServer = System.getenv().containsKey("CI")

buildCache {
    local {
        isEnabled = !isCiServer
    }
    remote<HttpBuildCache> {
        url = uri("https://example.com:8123/cache/")
        isPush = isCiServer
    }
}
```

buildSrc/settings.gradle.kts

```
apply(from = File(settingsDir, "../gradle/buildCacheSettings.gradle.kts"))
```

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 55. 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 and composite builds

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.

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.

Build Init Plugin

The Gradle Build Init plugin can be used to create a new Gradle build. It supports creating brand new projects of different types as well as converting existing builds (e.g. An Apache Maven build) to be Gradle builds.

Gradle plugins typically need to be *applied* to a project before they can be used (see Using plugins). The Build Init plugin is an automatically applied plugin, which means you do not need to apply it explicitly. To use the plugin, 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.

It also leverages the `wrapper` task to generate the Gradle Wrapper files for the project.

Tasks

The plugin adds the following tasks to the project:

`init` — *InitBuild*

Depends on: `wrapper`

Generates a Gradle project.

`wrapper` — *Wrapper*

Generates Gradle wrapper files.

What to set up

The `init` task supports different build init *types*, which determines exactly what files will be generated. The available types are listed below.

The simplest, and recommended, way to use the `init` task is to run `gradle init` from an interactive console. Gradle will list the available init 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 init type can be specified by using the `--type` command-line option. For example, to create

a Java library project simply 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 init 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 simply execute: `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.

All build setup types include the setup of the Gradle Wrapper.

Build init types

`pom` build init 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.

Note that the migration from Maven builds currently only supports the Groovy DSL for generated build scripts.

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, java 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 and tests
- Supports TestNG runner
- Generates global exclusions from Maven enforcer plugin settings

`java-application` build init type

The "`java-application`" build init 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 using Java
- Uses the "`jcenter`" dependency repository
- Uses JUnit 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 spock`: Uses Spock for testing instead of JUnit
- `gradle init --type java-application --test-framework testng`: Uses TestNG for testing instead of JUnit

`java-library` build init type

The "`java-library`" build init type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`java`" plugin to produce a library Jar
- Uses the "`jcenter`" dependency repository
- Uses JUnit 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 spock`: Uses Spock for testing instead of JUnit
- `gradle init --type java-library --test-framework testng`: Uses TestNG for testing instead of JUnit

`kotlin-application` build init type

The "`kotlin-application`" build init 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 "`jcenter`" 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 init type

The "`kotlin-library`" build init 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 Jar
- Uses the "`jcenter`" 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

`scala-library` build init type

The "`scala-library`" build init type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`scala`" plugin to produce a library Jar
- Uses the "`jcenter`" dependency repository
- Uses Scala 2.11
- 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
- Uses the Zinc Scala compiler by default

`groovy-library` build init type

The "`groovy-library`" build init type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`groovy`" plugin to produce a library Jar
- Uses the "`jcenter`" 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 init type

The "`groovy-application`" build init type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the "`groovy`" plugin
- Uses the "`application`" plugin to produce a command-line application implemented using Groovy
- Uses the "`jcenter`" 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

`basic` build init type

The "`basic`" build init type is useful for creating a fresh new Gradle project. 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.

Build Lifecycle

We said earlier that the core of Gradle is a language for dependency based programming. In Gradle terms this means that you can define tasks and dependencies between tasks. Gradle guarantees that these tasks are executed in the order of their dependencies, and that each task is executed only once. These tasks form a Directed Acyclic Graph. There are build tools that build up such a dependency graph as they execute their tasks. Gradle builds the complete dependency graph *before* any task is executed. This lies at the heart of Gradle and makes many things possible which would not be possible otherwise.

Your build scripts configure this dependency graph. Therefore they are strictly speaking *build configuration scripts*.

Build phases

A Gradle build has three distinct phases.

Initialization

Gradle supports single and multi-project builds. During the initialization phase, Gradle determines which projects are going to take part in the build, and creates a Project instance for each of these projects.

Configuration

During this phase the project objects are configured. The build scripts of *all* projects which are part of the build are executed.

Execution

Gradle determines the subset of the tasks, created and configured during the configuration phase, to be executed. The subset is determined by the task name arguments passed to the `gradle` command and the current directory. Gradle then executes each of the selected tasks.

Settings file

Beside the build script files, Gradle defines a settings file. The settings file is determined by Gradle via a naming convention. The default name for this file is `settings.gradle`. Later in this chapter we explain how Gradle looks for a settings file.

The settings file is executed during the initialization phase. A multi-project build must have a `settings.gradle` file in the root project of the multi-project hierarchy. It is required because the settings file defines which projects are taking part in the multi-project build (see Authoring Multi-Project Builds). For a single-project build, a settings file is optional. Besides defining the included projects, you might need it to add libraries to your build script classpath (see Organizing Gradle Projects). Let's first do some introspection with a single project build:

Example 56. Single project build

settings.gradle

```
println 'This is executed during the initialization phase.'
```

build.gradle

```
println 'This is executed during the configuration phase.'

task configured {
    println 'This is also executed during the configuration phase.'
}

task test {
    doLast {
        println 'This is executed during the execution phase.'
    }
}

task 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.'
}
```

settings.gradle.kts

```
println("This is executed during the initialization phase.")
```

build.gradle.kts

```
println("This is executed during the configuration phase.")

task("configured") {
    println("This is also executed during the configuration phase.")
}

task("test") {
    doLast {
        println("This is executed during the execution phase.")
    }
}

task("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.")
}
```

Output of `gradle test testBoth`

```
> gradle test testBoth
This is executed during the initialization phase.

> Configure project :
This is executed during the configuration phase.
This is also executed during the configuration phase.
This is executed during the configuration phase as well.

> 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
```

For a build script, the property access and method calls are delegated to a project object. Similarly property access and method calls within the settings file is delegated to a settings object. Look at the Settings class in the API documentation for more information.

Multi-project builds

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. There is much more to say about multi-project builds in the chapter dedicated to this topic (see Authoring Multi-Project Builds).

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. By default it is assumed that the location of the settings file is also the location of the root project. But you can redefine the location of the root project in the settings file.

Building the tree

In the settings file you can use a set of methods to build the project tree. Hierarchical and flat physical layouts get special support.

Hierarchical layouts

Example 57. Hierarchical 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[]).

Flat layouts

Example 58. Flat layout

settings.gradle

```
includeFlat 'project3', 'project4'
```

settings.gradle.kts

```
includeFlat("project3", "project4")
```

The `includeFlat` method takes directory names as an argument. These directories need to exist as siblings of the root project directory. The location of these directories are considered as child projects of the root project in the multi-project tree.

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 59. Lookup of elements of the project tree

settings.gradle

```
println rootProject.name
println project(':projectA').name
```

settings.gradle.kts

```
println(rootProject.name)
println(project(":projectA").name)
```

Using this descriptor you can change the name, project directory and build file of a project.

Example 60. Modification of elements of the project tree

settings.gradle

```
rootProject.name = 'main'
project(':projectA').projectDir = new File(settingsDir, '../my-project-a')
project(':projectA').buildFileName = 'projectA.gradle'
```

settings.gradle.kts

```
rootProject.name = "main"
project(":projectA").projectDir = File(settingsDir, "../my-project-a")
project(":projectA").buildFileName = "projectA.gradle"
```

Look at the ProjectDescriptor class in the API documentation for more information.

Initialization

How does Gradle know whether to do a single or multi-project build? If you trigger a multi-project build from a directory with a settings file, things are easy. But Gradle also allows you to execute the build from within any subproject taking part in the build. [2: Gradle supports partial multi-project builds (see Authoring Multi-Project Builds).] If you execute Gradle from within a project with no `settings.gradle` file, Gradle looks for a `settings.gradle` file in the following way:

- It looks in a directory called `master` which has the same nesting level as the current dir.

- If not found yet, it searches parent directories.
- If not found yet, the build is executed as a single project build.
- If a `settings.gradle` file is found, Gradle checks if the current project is part of the multi-project hierarchy defined in the found `settings.gradle` file. If not, the build is executed as a single project build. Otherwise a multi-project build is executed.

What is the purpose of this behavior? Gradle needs to determine whether the project you are in is a subproject of a multi-project build or not. Of course, if it is a subproject, only the subproject and its dependent projects are built, but Gradle needs to create the build configuration for the whole multi-project build (see Authoring Multi-Project Builds). If the current project contains a `settings.gradle` file, the build is always executed as:

- a single project build, if the `settings.gradle` file does not define a multi-project hierarchy
- a multi-project build, if the `settings.gradle` file does define a multi-project hierarchy.

The automatic search for a `settings.gradle` file only works for multi-project builds with a physical hierarchical or flat layout. For a flat layout you must additionally follow the naming convention described above ("`master`"). Gradle supports arbitrary physical layouts for a multi-project build, but for such arbitrary layouts you need to execute the build from the directory where the settings file is located. For information on how to run partial builds from the root, see Running tasks by their absolute path.

Gradle creates a Project object for every project taking part in the build. For a multi-project build these are the projects specified in the Settings object (plus the root project). Each project object has by default a name equal to the name of its top level directory, and every project except the root project has a parent project. Any project may have child projects.

Configuration and execution of a single project build

For a single project build, the workflow of the *after initialization* phases are pretty simple. The build script is executed against the project object that was created during the initialization phase. Then Gradle looks for tasks with names equal to those passed as command line arguments. If these task names exist, they are executed as a separate build in the order you have passed them. The configuration and execution for multi-project builds is discussed in Authoring Multi-Project Builds.

Responding to the lifecycle in the build script

Your build script can receive notifications as the build progresses through its lifecycle. These notifications generally take two forms: You can either implement a particular listener interface, or you can provide a closure to execute when the notification is fired. The examples below use closures. For details on how to use the listener interfaces, refer to the API documentation.

Project evaluation

You can receive a notification immediately before and after a project is evaluated. This can be used to do things like performing additional configuration once all the definitions in a build script have been applied, or for some custom logging or profiling.

Below is an example which adds a `test` task to each project which has a `hasTests` property value of true.

Example 61. Adding of test task to each project which has certain property set

build.gradle

```
allprojects {
    afterEvaluate { project ->
        if (project.hasTests) {
            println "Adding test task to $project"
            project.task('test') {
                doLast {
                    println "Running tests for $project"
                }
            }
        }
    }
}
```

projectA.gradle

```
hasTests = true
```

build.gradle.kts

```
allprojects {
    afterEvaluate {
        if (extra["hasTests"] as Boolean) {
            println("Adding test task to $project")
            task("test") {
                doLast {
                    println("Running tests for $project")
                }
            }
        }
    }
}
```

projectA.gradle.kts

```
extra["hasTests"] = true
```

Output of `gradle -q test`

```
> gradle -q test
Adding test task to project ':projectA'
Running tests for project ':projectA'
```

This example uses method `Project.afterEvaluate()` to add a closure which is executed after the project is evaluated.

It is also possible to receive notifications when any project is evaluated. This example performs some custom logging of project evaluation. Notice that the `afterProject` notification is received regardless of whether the project evaluates successfully or fails with an exception.

Example 62. Notifications

build.gradle

```
gradle.afterProject { project ->
    if (project.state.failure) {
        println "Evaluation of $project FAILED"
    } else {
        println "Evaluation of $project succeeded"
    }
}
```

build.gradle.kts

```
gradle.afterProject {
    if (state.failure != null) {
        println("Evaluation of $project FAILED")
    } else {
        println("Evaluation of $project succeeded")
    }
}
```

Output of `gradle -q test`

```
> gradle -q test
Evaluation of root project 'buildProjectEvaluateEvents' succeeded
Evaluation of project ':projectA' succeeded
Evaluation of project ':projectB' FAILED

FAILURE: Build failed with an exception.

* Where:
Build file '/home/user/gradle/samples/groovy/projectB.gradle' line: 1

* What went wrong:
A problem occurred evaluating project ':projectB'.
> 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 test
Evaluation of root project 'buildProjectEvaluateEvents' succeeded
Evaluation of project ':projectA' succeeded
Evaluation of project ':projectB' FAILED

FAILURE: Build failed with an exception.

* Where:
Build file '/home/user/gradle/samples/kotlin/projectB.gradle.kts' line: 1

* What went wrong:
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
```

You can also add a ProjectEvaluationListener to the Gradle to receive these events.

Task creation

You can receive a notification immediately after a task is added to a project. This can be used to set

some default values or add behaviour before the task is made available in the build file.

The following example sets the `srcDir` property of each task as it is created.

Example 63. Setting of certain property to all tasks

build.gradle

```
tasks.whenTaskAdded { task ->
    task.ext.srcDir = 'src/main/java'
}

task a

println "source dir is $a.srcDir"
```

build.gradle.kts

```
tasks.whenTaskAdded {
    extra["srcDir"] = "src/main/java"
}

val a = task("a")

println("source dir is ${a.extra["srcDir"]}")
```

Output of `gradle -q a`

```
> gradle -q a
source dir is src/main/java
```

You can also add an Action to a TaskContainer to receive these events.

Task execution graph ready

You can receive a notification immediately after the task execution graph has been populated (See Configure by DAG).

You can also add a TaskExecutionGraphListener to the TaskExecutionGraph to receive these events.

Task execution

You can receive a notification immediately before and after any task is executed.

The following example logs the start and end of each task execution. Notice that the `afterTask`

notification is received regardless of whether the task completes successfully or fails with an exception.

Example 64. Logging of start and end of each task execution

build.gradle

```
task ok

task 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

```
task("ok")

task("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")
    }
}
```

Output of `gradle -q broken`

```
> gradle -q broken
executing task ':ok' ...
done
executing task ':broken' ...
FAILED

FAILURE: Build failed with an exception.

* Where:
Build file '/home/user/gradle/samples/groovy/build.gradle' line: 5

* 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
```

You can also use a TaskExecutionListener to the TaskExecutionGraph to receive these events.

Build Script Basics

Projects and tasks

Everything in Gradle sits on top of two basic concepts: *projects* 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.

Each project is made up of 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.

For now, we will look at defining some simple tasks in a build with one project. Later chapters will look at working with multiple projects and more about working with projects and tasks.

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. [3: There are command line switches to change this behavior. See Command-Line Interface)] 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. [4: There are command line switches to change this behavior. See Command-Line Interface)] 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 65. Your first build script

build.gradle

```
task hello {
    doLast {
        println 'Hello world!'
    }
}
```

build.gradle.kts

```
task("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 66. 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 67. Using Groovy or Kotlin in Gradle's tasks

build.gradle

```
task upper {
    doLast {
        String someString = 'mY_nAmE'
        println "Original: $someString"
        println "Upper case: ${someString.toUpperCase()}"
    }
}
```

build.gradle.kts

```
task("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 68. Using Groovy or Kotlin in Gradle's tasks

build.gradle

```
task count {
    doLast {
        4.times { print "$it " }
    }
}
```

build.gradle.kts

```
task("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 69. Declaration of task that depends on other task

build.gradle

```
task hello {
    doLast {
        println 'Hello world!'
    }
}
task intro {
    dependsOn hello
    doLast {
        println "I'm Gradle"
    }
}
```

build.gradle.kts

```
task("hello") {
    doLast {
        println("Hello world!")
    }
}
task("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 70. Lazy dependsOn - the other task does not exist (yet)

build.gradle

```
task taskX {
    dependsOn 'taskY'
    doLast {
        println 'taskX'
    }
}
task taskY {
    doLast {
        println 'taskY'
    }
}
```

build.gradle.kts

```
task("taskX") {
    dependsOn("taskY")
    doLast {
        println("taskX")
    }
}
task("taskY") {
    doLast {
        println("taskY")
    }
}
```

Output of `gradle -q taskX`

```
> gradle -q taskX
taskY
taskX
```

The dependency of `taskX` to `taskY` is declared before `taskY` is defined. This is very important for multi-project builds. Task dependencies are discussed in more detail in Adding dependencies to a task.

Please notice that you can't use shortcut notation when referring to a task that is not yet defined.

Dynamic tasks

The power of Groovy or Kotlin can be used for more than defining what a task does. For example, you can also use it to dynamically create tasks.

Example 71. Dynamic creation of a task

build.gradle

```
4.times { counter ->
    task "task$counter" {
        doLast {
            println "I'm task number $counter"
        }
    }
}
```

build.gradle.kts

```
repeat(4) { counter ->
    task("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 created 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 72. Accessing a task via API - adding a dependency

build.gradle

```
4.times { counter ->
    task "task$counter" {
        doLast {
            println "I'm task number $counter"
        }
    }
}
task0.dependsOn task2, task3
```

build.gradle.kts

```
repeat(4) { counter ->
    task("task$counter") {
        doLast {
            println("I'm task number $counter")
        }
    }
}
tasks["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 73. Accessing a task via API - adding behaviour

build.gradle

```
task hello {
    doLast {
        println 'Hello Earth'
    }
}
hello.doFirst {
    println 'Hello Venus'
}
hello.doLast {
    println 'Hello Mars'
}
hello {
    doLast {
        println 'Hello Jupiter'
    }
}
```

build.gradle.kts

```
val hello = task("hello") {
    doLast {
        println("Hello Earth")
    }
}
hello.doFirst {
    println("Hello Venus")
}
hello.doLast {
    println("Hello Mars")
}
hello.apply {
    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.

Groovy DSL shortcut notations

There is a convenient notation for accessing an *existing* task. Each task is available as a property of the build script:

Example 74. Accessing task as a property of the build script

build.gradle

```
task hello {
    doLast {
        println 'Hello world!'
    }
}
hello.doLast {
    println "Greetings from the $hello.name task."
}
```

Output of `gradle -q hello`

```
> gradle -q hello
Hello world!
Greetings from the hello task.
```

This enables very readable code, especially when using the tasks provided by the plugins, like the `compile` task.

Extra task properties

You can add your own properties to a task. To add a property named `myProperty`, set `ext.myProperty` to an initial value. From that point on, the property can be read and set like a predefined task property.

Example 75. Adding extra properties to a task

build.gradle

```
task myTask {
    ext.myProperty = "myValue"
}

task printTaskProperties {
    doLast {
        println myTask.myProperty
    }
}
```

build.gradle.kts

```
task("myTask") {
    extra["myProperty"] = "myValue"
}

task("printTaskProperties") {
    doLast {
        println(tasks["myTask"].extra["myProperty"])
    }
}
```

Output of `gradle -q printTaskProperties`

```
> gradle -q printTaskProperties
myValue
```

Extra properties aren't limited to tasks. You can read more about them in Extra properties.

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 76. Using AntBuilder to execute ant.loadfile target

build.gradle

```
task 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

```
task("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 77. Using methods to organize your build logic

build.gradle

```
task checksum {
    doLast {
        fileList('./antLoadfileResources').each { File file ->
            ant.checksum(file: file, property: "cs_$file.name")
            println "$file.name Checksum: ${ant.properties["cs_$file.name"]}"
        }
    }
}

task 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

```
task("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}"]}")
        }
    }
}

task("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 78. Defining a default task

build.gradle

```
defaultTasks 'clean', 'run'

task clean {
    doLast {
        println 'Default Cleaning!'
    }
}

task run {
    doLast {
        println 'Default Running!'
    }
}

task other {
    doLast {
        println "I'm not a default task!"
    }
}
```

build.gradle.kts

```
defaultTasks("clean", "run")

task("clean") {
    doLast {
        println("Default Cleaning!")
    }
}

task("run") {
    doLast {
        println("Default Running!")
    }
}

task("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).

Configure by DAG

As we later describe in full detail (see Build Lifecycle), Gradle has a configuration phase and an execution phase. After the configuration phase, Gradle knows all tasks that should be executed. Gradle offers you a hook to make use of this information. A use-case for this would be to check if the release task is among the tasks to be executed. Depending on this, you can assign different values to some variables.

In the following example, execution of the `distribution` and `release` tasks results in different value of the `version` variable.

Example 79. Different outcomes of build depending on chosen tasks

build.gradle

```
task distribution {
    doLast {
        println "We build the zip with version=$version"
    }
}

task release {
    dependsOn 'distribution'
    doLast {
        println 'We release now'
    }
}

gradle.taskGraph.whenReady { taskGraph ->
    if (taskGraph.hasTask(":release")) {
        version = '1.0'
    } else {
        version = '1.0-SNAPSHOT'
    }
}
```

build.gradle.kts

```
task("distribution") {
    doLast {
        println("We build the zip with version=$version")
    }
}

task("release") {
    dependsOn("distribution")
    doLast {
        println("We release now")
    }
}

gradle.taskGraph.whenReady {
    version =
        if (hasTask(":release")) "1.0"
        else "1.0-SNAPSHOT"
}
```

Output of `gradle -q distribution`

```
> gradle -q distribution
We build the zip with version=1.0-SNAPSHOT
```

Output of `gradle -q release`

```
> gradle -q release
We build the zip with version=1.0
We release now
```

The important thing is that `whenReady` affects the release task *before* the release task is executed. This works even when the release task is not the *primary* task (i.e., the task passed to the `gradle` command).

External dependencies for the build script

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 80. 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 81. 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'
    }
}

task 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")
    }
}

task("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.

Where to next?

In this chapter, we have had a first look at tasks. But this is not the end of the story for tasks. If you want to jump into more of the details, have a look at More About Tasks.

Otherwise, continue on to the tutorials and Dependency Management for Java Projects.

Composite 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.

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 82. Dependencies of my-app

my-app/build.gradle

```
plugins {
    id 'java'
    id 'application'
    id 'idea'
}

group "org.sample"
version "1.0"

application {
    mainClassName = "org.sample.myapp.Main"
}

dependencies {
    implementation "org.sample:number-utils:1.0"
    implementation "org.sample:string-utils:1.0"
}

repositories {
    jcenter()
}
```

my-app/build.gradle.kts

```
plugins {
    java
    application
    idea
}

group = "org.sample"
version = "1.0"

application {
    mainClassName = "org.sample.myapp.Main"
}

dependencies {
    implementation("org.sample:number-utils:1.0")
    implementation("org.sample:string-utils:1.0")
}

repositories {
    jcenter()
}
```

NOTE	The code for this example can be found at `samples/compositeBuilds/basic` in the '-all' distribution of Gradle.

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.

Example: Declaring a command-line composite

Output of `gradle --include-build ../my-utils run`

```
> gradle --include-build ../my-utils run
> Task :processResources NO-SOURCE
> Task :my-utils:string-utils:compileJava
> Task :my-utils:string-utils:processResources NO-SOURCE
> Task :my-utils:string-utils:classes
> Task :my-utils:string-utils:jar
> Task :my-utils:number-utils:compileJava
> Task :my-utils:number-utils:processResources NO-SOURCE
> Task :my-utils:number-utils:classes
> Task :my-utils:number-utils:jar
> Task :compileJava
> Task :classes

> Task :run
The answer is 42


BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

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 83. Declaring a separate composite

composite/settings.gradle

```
rootProject.name = 'adhoc'

includeBuild '../my-app'
includeBuild '../my-utils'
```

composite/settings.gradle.kts

```
rootProject.name = "adhoc"

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 84. Depending on task from included build

composite/build.gradle

```
task run {
    dependsOn gradle.includedBuild('my-app').task(':run')
}
```

composite/build.gradle.kts

```
task("run") {
    dependsOn(gradle.includedBuild("my-app").task(":run"))
}
```

More details tasks that depend on included build tasks below.

Restrictions on included builds

Most builds can be included into a composite, including other composite builds. However there are some limitations.

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 the composite build can be executed from the command line, or from you IDE. Executing a task will result in direct task dependencies being executed, as well as those tasks required to build dependency artifacts from included builds.

NOTE	There is not (yet) any means to directly execute a task from an included build via the command line. 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}`.

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 85. 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 86. Declaring the substitutions for an included build

settings.gradle

```
rootProject.name = 'app'

includeBuild('../anonymous-library') {
    dependencySubstitution {
        substitute module('org.sample:number-utils') with project(':')
    }
}
```

settings.gradle.kts

```
rootProject.name = "app"

includeBuild("../anonymous-library") {
    dependencySubstitution {
        substitute(module("org.sample:number-utils")).with(project(":"))
    }
}
```

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".

Cases where included build substitutions must be declared

Many builds that use the `uploadArchives` task to publish artifacts 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: this usually means a task other than `uploadArchives` is used.
- 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 87. Depending on a single task from an included build

composite/build.gradle

```
task run {
    dependsOn gradle.includedBuild('my-app').task(':run')
}
```

composite/build.gradle.kts

```
task("run") {
    dependsOn(gradle.includedBuild("my-app").task(":run"))
}
```

Example 88. Depending on a task with path in all included builds

build.gradle

```
task publishDeps {
    dependsOn gradle.includedBuilds*.task(':uploadArchives')
}
```

build.gradle.kts

```
task("publishDeps") {
    dependsOn(gradle.includedBuilds.map { it.task(":uploadArchives") })
}
```

Current limitations and future plans for composite builds

We think composite builds are pretty useful already. However, there are some things that don't yet work the way we'd like, and other improvements that we think will make things work even better.

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.
- Native builds are not supported. (Binary dependencies are not yet supported for native builds).
- Substituting plugins only works with the `buildscript` block but not with the `plugins` block.

Improvements we have planned for upcoming releases include:

- Better detection of dependency substitution, for build that publish with custom coordinates, builds that produce multiple components, etc. This will reduce the cases where dependency substitution needs to be explicitly declared for an included build.
- The ability to target a task or tasks in an included build directly from the command line. We are currently exploring syntax options for allowing this functionality, which will remove many cases where a delegating task is required in the composite.
- Make the `plugins {}` block consider included builds when locating plugins and their dependencies.
- Making the implicit `buildSrc` project an included build.

Authoring Multi-Project Builds

The powerful support for multi-project builds is one of Gradle's unique selling points. This topic is also the most intellectually challenging.

A multi-project build in gradle consists of one root project, and one or more subprojects that may also have subprojects.

Cross project configuration

While each subproject could configure itself in complete isolation of the other subprojects, it is common that subprojects share common traits. It is then usually preferable to share configurations among projects, so the same configuration affects several subprojects.

Let's start with a very simple multi-project build. Gradle is a general purpose build tool at its core, so the projects don't have to be Java projects. Our first examples are about marine life.

Configuration and execution

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 executing the `build.gradle` (or `build.gradle.kts`) file of a project, which implies e.g. downloading all plugins that were declared using '`apply plugin`' or a `plugins` block. 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 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.

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. Scalability is an important requirement for Gradle. Hence, starting from version 1.4 a new incubating 'configuration on demand' mode is introduced.

Configuration on demand mode attempts to configure only projects that are relevant for requested tasks, i.e. it only executes the `build.gradle[.kts]` file of projects that are participating in the build. This way, the configuration time of a large multi-project build can be reduced. In the long term, this mode will become the default mode, possibly the only mode for Gradle build execution. 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. This way the typical common configuration is supported (allprojects or subprojects script blocks).
- 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(":someOtherProject:someOtherTask")`

- A task requested via task path from the command line (or Tooling API) causes the relevant project to be configured. For example, building 'projectA:projectB:someTask' causes configuration of projectB.

Eager to try out this new feature? 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.

Defining common behavior

Let's look at some examples with the following project tree. This is a multi-project build with a root project named `water` and a subproject named `bluewhale`.

Example 89. Multi-project tree - water & bluewhale projects

Project layout

```
.
├─── bluewhale/
├─── build.gradle
└─── settings.gradle
```

Project layout

```
.
├─── bluewhale/
├─── build.gradle.kts
└─── settings.gradle.kts
```

NOTE	The code for this example can be found at `samples/userguide/multiproject/firstExample/water` in the '-all' distribution of Gradle.

settings.gradle

```
rootProject.name = 'water'
include 'bluewhale'
```

settings.gradle.kts

```
rootProject.name = "water"
include("bluewhale")
```

And where is the build script for the `bluewhale` project? In Gradle build scripts are optional. Obviously for a single project build, a project without a build script doesn't make much sense. For multiproject builds the situation is different. Let's look at the build script for the `water` project and execute it:

Example 90. Build script of water (parent) project

build.gradle

```
Closure cl = { task -> println "I'm $task.project.name" }
task('hello').doLast(cl)
project(':bluewhale') {
    task('hello').doLast(cl)
}
```

build.gradle.kts

```
val cl = Action<Task> { println("I'm ${this.project.name}") }
task("hello").doLast(cl)
project(":bluewhale") {
    task("hello").doLast(cl)
}
```

Output of `gradle -q hello`

```
> gradle -q hello
I'm water
I'm bluewhale
```

Gradle allows you to access any project of the multi-project build from any build script. The Project API provides a method called `project()`, which takes a path as an argument and returns the Project object for this path. The capability to configure a project build from any build script we call *cross project configuration*. Gradle implements this via *configuration injection*.

We are not that happy with the build script of the `water` project. It is inconvenient to add the task explicitly for every project. We can do better. Let's first add another project called `krill` to our multi-project build.

Example 91. Multi-project tree - water, bluewhale & krill projects

Project layout

```
.
├─── bluewhale/
├─── build.gradle
├─── krill/
└─── settings.gradle
```

Project layout

```
.
├─── bluewhale/
├─── build.gradle.kts
├─── krill/
└─── settings.gradle.kts
```

NOTE The code for this example can be found at `samples/userguide/multiproject/addKrill/water` in the '-all' distribution of Gradle.

settings.gradle

```
rootProject.name = 'water'

include 'bluewhale', 'krill'
```

settings.gradle.kts

```
rootProject.name = "water"

include("bluewhale", "krill")
```

Now we rewrite the `water` build script and boil it down to a single line.

Example 92. Water project build script

build.gradle

```
allprojects {
    task hello {
        doLast { task ->
            println "I'm $task.project.name"
        }
    }
}
```

build.gradle.kts

```
allprojects {
    task("hello") {
        doLast {
            println("I'm ${this.project.name}")
        }
    }
}
```

Output of `gradle -q hello`

```
> gradle -q hello
I'm water
I'm bluewhale
I'm krill
```

Is this cool or is this cool? And how does this work? The Project API provides a property `allprojects` which returns a list with the current project and all its subprojects underneath it. If you call `allprojects` with a closure, the statements of the closure are delegated to the projects associated with `allprojects`. You could also do an iteration via `allprojects.each` (in Groovy) or `allprojects.forEach` (in Kotlin), but that would be more verbose.

Other build systems use inheritance as the primary means for defining common behavior. We also offer inheritance for projects as you will see later. But Gradle uses configuration injection as the usual way of defining common behavior. We think it provides a very powerful and flexible way of configuring multiproject builds.

Another possibility for sharing configuration is to use a common external script.

Subproject configuration

The Project API also provides a property for accessing the subprojects only.

Defining common behavior

Example 93. Defining common behavior of all projects and subprojects

build.gradle

```
allprojects {
    task hello {
        doLast { task ->
            println "I'm $task.project.name"
        }
    }
}
subprojects {
    hello {
        doLast {
            println "- I depend on water"
        }
    }
}
```

build.gradle.kts

```
allprojects {
    task("hello") {
        doLast {
            println("I'm ${this.project.name}")
        }
    }
}
subprojects {
    tasks.getByName("hello") {
        doLast {
            println("- I depend on water")
        }
    }
}
```

Output of `gradle -q hello`

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
I'm krill
- I depend on water
```

You may notice that there are two code snippets referencing the “hello” task. The first one, which uses the “task” keyword (in Groovy) or the task() function (in Kotlin), constructs the task and provides it’s base configuration. The second piece doesn’t use the “task” keyword or function, as it is further configuring the existing “hello” task. You may only construct a task once in a project, but you may add any number of code blocks providing additional configuration.

Adding specific behavior

You can add specific behavior on top of the common behavior. Usually we put the project specific behavior in the build script of the project where we want to apply this specific behavior. But as we have already seen, we don’t have to do it this way. We could add project specific behavior for the bluewhale project like this:

Example 94. Defining specific behaviour for particular project

build.gradle

```
allprojects {
    task hello {
        doLast { task ->
            println "I'm $task.project.name"
        }
    }
}
subprojects {
    hello {
        doLast {
            println "- I depend on water"
        }
    }
}
project(':bluewhale').hello {
    doLast {
        println "- I'm the largest animal that has ever lived on this
planet."
    }
}
```

build.gradle.kts

```
allprojects {
    task("hello") {
        doLast {
            println("I'm ${this.project.name}")
        }
    }
}
subprojects {
    tasks.getByName("hello") {
        doLast {
            println("- I depend on water")
        }
    }
}
project(":bluewhale").tasks.getByName("hello") {
    doLast {
        println("- I'm the largest animal that has ever lived on this
planet.")
    }
}
```

Output of `gradle -q hello`

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
I'm krill
- I depend on water
```

As we have said, we usually prefer to put project specific behavior into the build script of this project. Let's refactor and also add some project specific behavior to the `krill` project.

Example 95. Defining specific behaviour for project krill

Project layout

```
.
├─── bluewhale
│    └─── build.gradle
├─── build.gradle
├─── krill
│    └─── build.gradle
└─── settings.gradle
```

Project layout

```
.
├─── bluewhale
│    └─── build.gradle.kts
├─── build.gradle.kts
├─── krill
│    └─── build.gradle.kts
└─── settings.gradle.kts
```

NOTE The code for this example can be found at `samples/userguide/multiproject/spreadSpecifics/water` in the '-all' distribution of Gradle.

settings.gradle

```
rootProject.name = 'water'
include 'bluewhale', 'krill'
```

bluewhale/build.gradle

```
hello.doLast {
  println "- I'm the largest animal that has ever lived on this planet."
}
```

krill/build.gradle

```
hello.doLast {
  println "- The weight of my species in summer is twice as heavy as all
human beings."
}
```

build.gradle

```
allprojects {
    task hello {
        doLast { task ->
            println "I'm $task.project.name"
        }
    }
}
subprojects {
    hello {
        doLast {
            println "- I depend on water"
        }
    }
}
```

settings.gradle.kts

```
rootProject.name = "water"
include("bluewhale", "krill")
```

bluewhale/build.gradle.kts

```
tasks.getByName("hello").doLast {
    println("- I'm the largest animal that has ever lived on this planet.")
}
```

krill/build.gradle.kts

```
tasks.getByName("hello").doLast {
    println("- The weight of my species in summer is twice as heavy as all
human beings.")
}
```

build.gradle.kts

```
allprojects {
    task("hello") {
        doLast {
            println("I'm ${this.project.name}")
        }
    }
}
subprojects {
    tasks.getByName("hello") {
        doLast {
            println("- I depend on water")
        }
    }
}
```

Output of `gradle -q hello`

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
I'm krill
- I depend on water
- The weight of my species in summer is twice as heavy as all human beings.
```

Project filtering

To show more of the power of configuration injection, let's add another project called `tropicalFish` and add more behavior to the build via the build script of the `water` project.

Filtering by name

Example 96. Adding custom behaviour to some projects (filtered by project name)

Project layout

```
.
├─── bluewhale/
│    └─── build.gradle
├─── build.gradle
├─── krill/
│    └─── build.gradle
├─── settings.gradle
└─── tropicalFish/
```

Project layout

```
.
├─── bluewhale/
│    └─── build.gradle.kts
├─── build.gradle.kts
├─── krill/
│    └─── build.gradle.kts
├─── settings.gradle.kts
└─── tropicalFish/
```

NOTE The code for this example can be found at `samples/userguide/multiproject/addTropical/water` in the '-all' distribution of Gradle.

settings.gradle

```
rootProject.name = 'water'
include 'bluewhale', 'krill', 'tropicalFish'
```

build.gradle

```
allprojects {
    task hello {
        doLast { task ->
            println "I'm $task.project.name"
        }
    }
}
subprojects {
    hello {
        doLast {
            println "- I depend on water"
        }
    }
}
configure(subprojects.findAll {it.name != 'tropicalFish'}) {
    hello {
        doLast {
            println '- I love to spend time in the arctic waters.'
        }
    }
}
```

settings.gradle.kts

```
rootProject.name = "water"
include("bluewhale", "krill", "tropicalFish")
```

build.gradle.kts

```
allprojects {
    task("hello") {
        doLast {
            println("I'm ${this.project.name}")
        }
    }
}
subprojects {
    tasks.getByName("hello") {
        doLast {
            println("- I depend on water")
        }
    }
}
configure(subprojects.filter { it.name != "tropicalFish" }) {
    tasks.getByName("hello") {
        doLast {
            println("- I love to spend time in the arctic waters.")
        }
    }
}
```

Output of `gradle -q hello`

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I love to spend time in the arctic waters.
- I'm the largest animal that has ever lived on this planet.
I'm krill
- I depend on water
- I love to spend time in the arctic waters.
- The weight of my species in summer is twice as heavy as all human beings.
I'm tropicalFish
- I depend on water
```

The `configure()` method takes a list as an argument and applies the configuration to the projects in this list.

Filtering by properties

Using the project name for filtering is one option. Using extra project properties is another.

Example 97. Adding custom behaviour to some projects (filtered by project properties)

Project layout

```
.
├─── bluewhale
│    └─── build.gradle
├─── build.gradle
├─── krill
│    └─── build.gradle
├─── settings.gradle
└─── tropicalFish
     └─── build.gradle
```

Project layout

```
.
├─── bluewhale
│    └─── build.gradle.kts
├─── build.gradle.kts
├─── krill
│    └─── build.gradle.kts
├─── settings.gradle.kts
└─── tropicalFish
     └─── build.gradle.kts
```

NOTE The code for this example can be found at `samples/userguide/multiproject/tropicalWithProperties/water` in the '-all' distribution of Gradle.

settings.gradle

```
rootProject.name = 'water'
include 'bluewhale', 'krill', 'tropicalFish'
```

bluewhale/build.gradle

```
ext.arctic = true
hello.doLast {
  println "- I'm the largest animal that has ever lived on this planet."
}
```

krill/build.gradle

```
ext.arctic = true
hello.doLast {
    println "- The weight of my species in summer is twice as heavy as all
human beings."
}
```

build.gradle

```
allprojects {
    task hello {
        doLast { task ->
            println "I'm $task.project.name"
        }
    }
}
subprojects {
    hello {
        doLast {println "- I depend on water"}
        afterEvaluate { Project project ->
            if (project.arctic) { doLast {
                println '- I love to spend time in the arctic waters.' }
            }
        }
    }
}
```

tropicalFish/build.gradle

```
ext.arctic = false
```

settings.gradle.kts

```
rootProject.name = "water"
include("bluewhale", "krill", "tropicalFish")
```

bluewhale/build.gradle.kts

```
extra["arctic"] = true
tasks.getByName("hello").doLast {
  println("- I'm the largest animal that has ever lived on this planet.")
}
```

krill/build.gradle.kts

```
extra["arctic"] = true
tasks.getByName("hello").doLast {
    println("- The weight of my species in summer is twice as heavy as all
human beings.")
}
```

build.gradle.kts

```
allprojects {
    task("hello") {
        doLast {
            println("I'm ${this.project.name}")
        }
    }
}
subprojects {
    tasks.getByName("hello") {
        doLast { println("- I depend on water") }
        afterEvaluate {
            if (extra["arctic"] as Boolean) {
                doLast {
                    println("- I love to spend time in the arctic waters.")
                }
            }
        }
    }
}
```

tropicalFish/build.gradle.kts

```
extra["arctic"] = false
```

Output of `gradle -q hello`

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
- I love to spend time in the arctic waters.
I'm krill
- I depend on water
- The weight of my species in summer is twice as heavy as all human beings.
- I love to spend time in the arctic waters.
I'm tropicalFish
- I depend on water
```

In the build file of the `water` project we use an `afterEvaluate` notification. This means that the closure we are passing gets evaluated *after* the build scripts of the subproject are evaluated. As the property `arctic` is set in those build scripts, we have to do it this way. You will find more on this topic in Dependencies — Which Dependencies?

Execution rules for multi-project builds

When we executed the `hello` task from the root project dir, things behaved in an intuitive way. All the `hello` tasks of the different projects were executed. Let's switch to the `bluewhale` dir and see what happens if we execute Gradle from there.

Running build from subproject

```
> gradle -q hello
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
- I love to spend time in the arctic waters.
```

The basic rule behind Gradle's behavior is simple. Gradle looks down the hierarchy, starting with the *current dir*, for tasks with the name `hello` 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 dir, 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. We will have a closer look at this in the next section. Let's now have our last marine example. Let's add a task to `bluewhale` and `krill`.

Example 98. Evaluation and execution of projects

bluewhale/build.gradle

```
ext.arctic = true
hello {
    doLast {
        println "- I'm the largest animal that has ever lived on this
planet."
    }
}

task distanceToIceberg {
    doLast {
        println '20 nautical miles'
    }
}
```

krill/build.gradle

```
ext.arctic = true
hello {
    doLast {
        println "- The weight of my species in summer is twice as heavy as
all human beings."
    }
}

task distanceToIceberg {
    doLast {
        println '5 nautical miles'
    }
}
```

bluewhale/build.gradle.kts

```
extra["arctic"] = true
tasks.getByName("hello") {
    doLast {
        println("- I'm the largest animal that has ever lived on this
planet.")
    }
}

task("distanceToIceberg") {
    doLast {
        println("20 nautical miles")
    }
}
```

krill/build.gradle.kts

```
extra["arctic"] = true
tasks.getByName("hello") {
    doLast {
        println("- The weight of my species in summer is twice as heavy as
all human beings.")
    }
}

task("distanceToIceberg") {
    doLast {
        println("5 nautical miles")
    }
}
```

Output of `gradle -q distanceToIceberg`

```
> gradle -q distanceToIceberg
20 nautical miles
5 nautical miles
```

Here's the output without the `-q` option:

Output of `gradle distanceToIceberg`

```
> gradle distanceToIceberg

> Task :bluewhale:distanceToIceberg
20 nautical miles

> Task :krill:distanceToIceberg
5 nautical miles

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

The build is executed from the `water` project. Neither `water` nor `tropicalFish` have a task with the name `distanceToIceberg`. Gradle does not care. The simple rule mentioned already above is: Execute all tasks down the hierarchy which have this name. Only complain if there is *no* such task!

Running tasks by their absolute path

As we have seen, you can run a multi-project build by entering any subproject dir and execute the build from there. All matching task names of the project hierarchy starting with the current dir are executed. But Gradle also offers to execute tasks by their absolute path (see also Project and task paths):

Running tasks by their absolute path

```
> gradle -q :hello :krill:hello hello
I'm water
I'm krill
- I depend on water
- The weight of my species in summer is twice as heavy as all human beings.
- I love to spend time in the arctic waters.
I'm tropicalFish
- I depend on water
```

The build is executed from the `tropicalFish` project. We execute the `hello` tasks of the `water`, the `krill` and the `tropicalFish` project. The first two tasks are specified by their absolute path, the last task is executed using the name matching mechanism described above.

Project and task paths

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.

The path of a task is simply its project path plus the task name, like "`:bluewhale:hello`". Within a project you can address a task of the same project just by its name. This is interpreted as a relative

path.

Dependencies - Which dependencies?

The examples from the last section were special, as the projects had no *Execution Dependencies*. They had only *Configuration Dependencies*. The following sections illustrate the differences between these two types of dependencies.

Execution dependencies

Dependencies and execution order

Example 99. Dependencies and execution order

Project layout

```
.
├─── build.gradle
├─── consumer
│    └─── build.gradle
├─── producer
│    └─── build.gradle
└─── settings.gradle
```

Project layout

```
.
├─── build.gradle.kts
├─── consumer
│    └─── build.gradle.kts
├─── producer
│    └─── build.gradle.kts
└─── settings.gradle.kts
```

NOTE The code for this example can be found at `samples/userguide/multiproject/dependencies/firstMessages/messages` in the '-all' distribution of Gradle.

build.gradle

```
ext.producerMessage = null
```

settings.gradle

```
include 'consumer', 'producer'
```

consumer/build.gradle

```
task action {
    doLast {
        println("Consuming message: ${rootProject.producerMessage}")
    }
}
```

producer/build.gradle

```
task action {
    doLast {
        println "Producing message:"
        rootProject.producerMessage = 'Watch the order of execution.'
    }
}
```

build.gradle.kts

```
extra["producerMessage"] = null
```

settings.gradle.kts

```
include("consumer", "producer")
```

consumer/build.gradle.kts

```
task("action") {
    doLast {
        println("Consuming message: ${rootProject.extra["producerMessage"]}")
    }
}
```

producer/build.gradle.kts

```
task("action") {
    doLast {
        println("Producing message:")
        rootProject.extra["producerMessage"] = "Watch the order of
execution."
    }
}
```

Output of `gradle -q action`

```
> gradle -q action
Consuming message: null
Producing message:
```

This didn't quite do what we want. If nothing else is defined, Gradle executes the task in alphanumeric order. Therefore, Gradle will execute "`:consumer:action`" before "`:producer:action`". Let's try to solve this with a hack and rename the producer project to "`aProducer`".

Example 100. Dependencies and execution order

Project layout

```
.
├──── aProducer
│     └──── build.gradle
├──── build.gradle
├──── consumer
│     └──── build.gradle
└──── settings.gradle
```

Project layout

```
.
├──── aProducer
│     └──── build.gradle.kts
├──── build.gradle.kts
├──── consumer
│     └──── build.gradle.kts
└──── settings.gradle.kts
```

build.gradle

```
ext.producerMessage = null
```

settings.gradle

```
include 'consumer', 'aProducer'
```

consumer/build.gradle

```
task action {
    doLast {
        println("Consuming message: ${rootProject.producerMessage}")
    }
}
```

aProducer/build.gradle

```
task action {
    doLast {
        println "Producing message:"
        rootProject.producerMessage = 'Watch the order of execution.'
    }
}
```

build.gradle.kts

```
extra["producerMessage"] = null
```

settings.gradle.kts

```
include("consumer", "aProducer")
```

consumer/build.gradle.kts

```
task("action") {
    doLast {
        println("Consuming message: ${rootProject.extra["producerMessage"]}")
    }
}
```

aProducer/build.gradle.kts

```
task("action") {
    doLast {
        println("Producing message:")
        rootProject.extra["producerMessage"] = "Watch the order of
execution."
    }
}
```

Output of `gradle -q action`

```
> gradle -q action
Producing message:
Consuming message: Watch the order of execution.
```

We can show where this hack doesn't work if we now switch to the `consumer` dir and execute the build.

Output of `gradle -q action` *from the* `consumer` *dir*

```
> gradle -q action
Consuming message: null
```

The problem is that the two "`action`" tasks are unrelated. If you execute the build from the "`messages`" project Gradle executes them both because they have the same name and they are down the hierarchy. In the last example only one "`action`" task was down the hierarchy and therefore it was the only task that was executed. We need something better than this hack.

Real life examples

Gradle's multi-project features are driven by real life use cases. One good example consists of two web application projects and a parent project that creates a distribution including the two web applications. [5: The real use case we had, was using http://lucene.apache.org/solr, where you need a separate war for each index you are accessing. That was one reason why we have created a distribution of webapps. The Resin servlet container allows us, to let such a distribution point to a base installation of the servlet container.] For the example we use only one build script and do *cross project configuration*.

Example 101. Dependencies - real life example - crossproject configuration

Project layout

```
.
├─── build.gradle
├─── date
│    └─── src
│        └─── main
│            ├─── java
│            │    └─── org
│            │        └─── gradle
│            │            └─── sample
│            │                └─── DateServlet.java
│            └─── webapp
│                └─── web.xml
├─── hello
│    └─── src
│        └─── main
│            ├─── java
│            │    └─── org
│            │        └─── gradle
│            │            └─── sample
│            │                └─── HelloServlet.java
│            └─── webapp
│                └─── web.xml
└─── settings.gradle
```

Project layout

```
.
├──── build.gradle.kts
├──── date
│    └──── src
│        └──── main
│            ├──── java
│            │    └──── org
│            │        └──── gradle
│            │            └──── sample
│            │                └──── DateServlet.java
│            └──── webapp
│                └──── web.xml
├──── hello
│    └──── src
│        └──── main
│            ├──── java
│            │    └──── org
│            │        └──── gradle
│            │            └──── sample
│            │                └──── HelloServlet.java
│            └──── webapp
│                └──── web.xml
└──── settings.gradle.kts
```

NOTE The code for this example can be found at `samples/userguide/multiproject/dependencies/webDist` in the '-all' distribution of Gradle.

settings.gradle

```
rootProject.name = 'webDist'
include 'date', 'hello'
```

build.gradle

```
allprojects {
    apply plugin: 'java'
    group = 'org.gradle.sample'
    version = '1.0'
}

subprojects {
    apply plugin: 'war'
    repositories {
        mavenCentral()
    }
    dependencies {
        compile "javax.servlet:servlet-api:2.5"
    }
}

task explodedDist(type: Copy) {
    into "$buildDir/explodedDist"
    subprojects {
        from tasks.withType(War)
    }
}
```

settings.gradle.kts

```
rootProject.name = "webDist"
include("date", "hello")
```

build.gradle.kts

```
allprojects {
    apply(plugin = "java")
    group = "org.gradle.sample"
    version = "1.0"
}

subprojects {
    apply(plugin = "war")
    repositories {
        mavenCentral()
    }
    dependencies {
        "providedCompile"("javax.servlet:servlet-api:2.5")
    }
}

task<Copy>("explodedDist") {
    into("$buildDir/explodedDist")
    subprojects {
        from(tasks.withType<War>())
    }
}
```

We have an interesting set of dependencies. Obviously the `date` and `hello` projects have a *configuration* dependency on `webDist`, as all the build logic for the webapp projects is injected by `webDist`. The *execution* dependency is in the other direction, as `webDist` depends on the build artifacts of `date` and `hello`. There is even a third dependency. `webDist` has a *configuration* dependency on `date` and `hello` because it needs to know the `archivePath`. But it asks for this information at *execution time*. Therefore we have no circular dependency.

Such dependency patterns are daily bread in the problem space of multi-project builds. If a build system does not support these patterns, you either can't solve your problem or you need to do ugly hacks which are hard to maintain and massively impair your productivity as a build master.

Project lib dependencies

What if one project needs the jar produced by another project in its compile path, and not just the jar but also the transitive dependencies of this jar? Obviously this is a very common use case for Java multi-project builds. As mentioned in Project dependencies, Gradle offers project lib

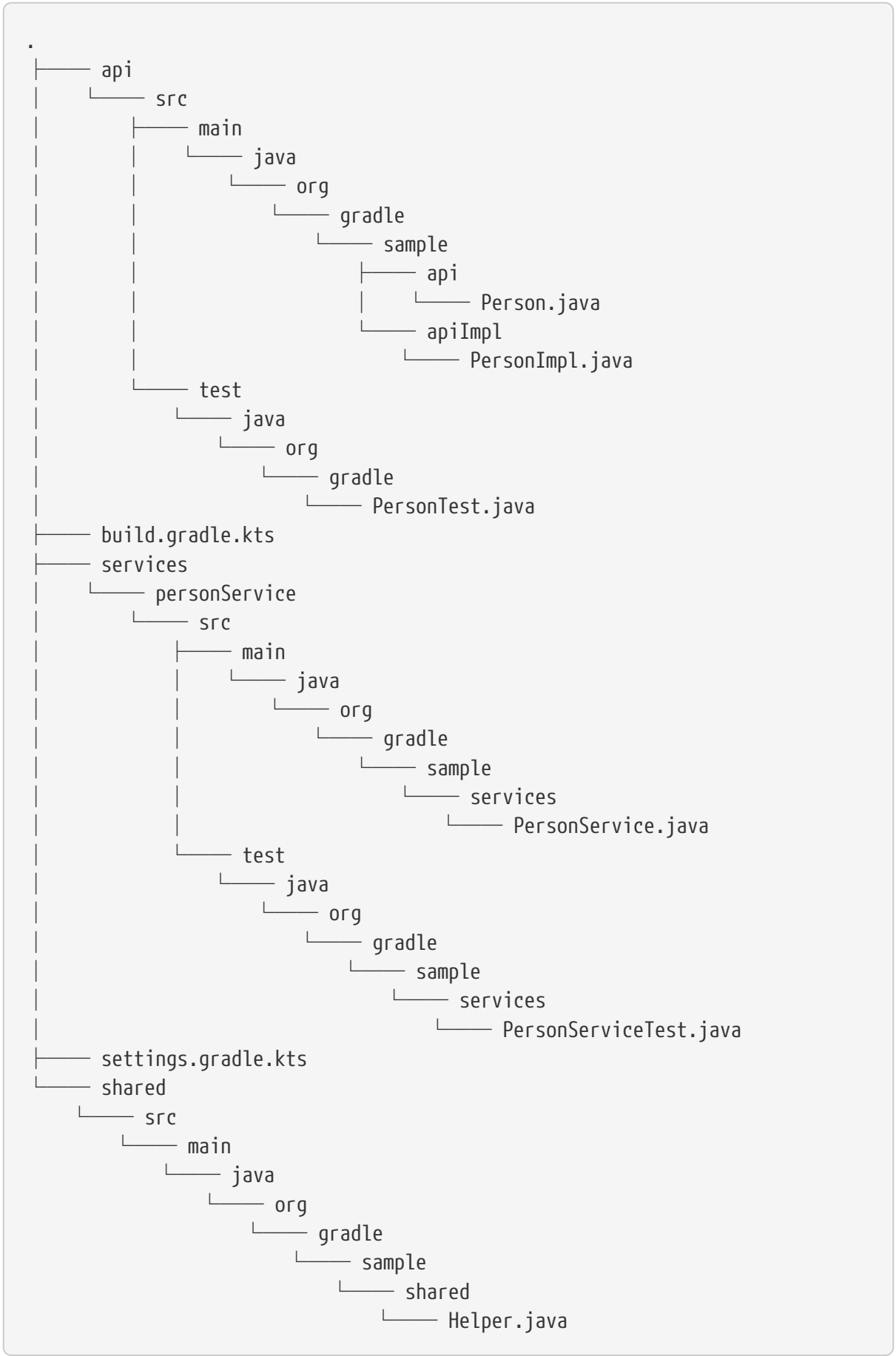
dependencies for this.

Example 102. Project lib dependencies

Project layout

```
.
├─── api
│    └─── src
│        ├─── main
│        │    └─── java
│        │        └─── org
│        │            └─── gradle
│        │                └─── sample
│        │                    ├─── api
│        │                    │    └─── Person.java
│        │                    └─── apiImpl
│        │                        └─── PersonImpl.java
│        └─── test
│            └─── java
│                └─── org
│                    └─── gradle
│                        └─── PersonTest.java
├─── build.gradle
├─── services
│    └─── personService
│        └─── src
│            ├─── main
│            │    └─── java
│            │        └─── org
│            │            └─── gradle
│            │                └─── sample
│            │                    └─── services
│            │                        └─── PersonService.java
│            └─── test
│                └─── java
│                    └─── org
│                        └─── gradle
│                            └─── sample
│                                └─── services
│                                    └─── PersonServiceTest.java
├─── settings.gradle
└─── shared
    └─── src
        └─── main
            └─── java
                └─── org
                    └─── gradle
                        └─── sample
                            └─── shared
                                └─── Helper.java
```

Project layout



NOTE	The code for this example can be found at `samples/userguide/multiproject/dependencies/java` in the '-all' distribution of Gradle.

We have the projects "`shared`", "`api`" and "`personService`". The "`personService`" project has a lib dependency on the other two projects. The "`api`" project has a lib dependency on the "`shared`" project. "`services`" is also a project, but we use it just as a container. 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

```
include 'api', 'shared', 'services:personService'
```

build.gradle

```
/*
 * Copyright 2018 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

subprojects {
    apply plugin: 'java'
    group = 'org.gradle.sample'
    version = '1.0'
    repositories {
        mavenCentral()
    }
    dependencies {
        testImplementation "junit:junit:4.12"
    }
}

project(':api') {
    dependencies {
        implementation project(':shared')
    }
}

project(':services:personService') {
    dependencies {
        implementation project(':shared'), project(':api')
    }
}
```

settings.gradle.kts

```
include("api", "shared", "services:personService")
```

build.gradle.kts

```
subprojects {
    apply(plugin = "java")
    group = "org.gradle.sample"
    version = "1.0"
    repositories {
        mavenCentral()
    }
    dependencies {
        "testImplementation"("junit:junit:4.12")
    }
}

project(":api") {
    dependencies {
        "implementation"(project(":shared"))
    }
}

project(":services:personService") {
    dependencies {
        "implementation"(project(":shared"))
        "implementation"(project(":api"))
    }
}
```

All the build logic is in the build script of the root project. [6: We do this here, as it makes the layout a bit easier. We usually put the project specific stuff into the build script of the respective projects.] A "*lib*" 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. So you can enter the "`api`" directory and trigger a "`gradle compile`". First the "`shared`" project is built and then the "`api`" project is built. Project dependencies enable partial multi-project builds.

If you come from Maven land you might be perfectly happy with this. If you come from Ivy land, you might expect some more fine grained control. Gradle offers this to you:

Example 103. Fine grained control over dependencies

build.gradle

```
subprojects {
    apply plugin: 'java'
    group = 'org.gradle.sample'
    version = '1.0'
}

project(':api') {
    configurations {
        spi
    }
    dependencies {
        compile project(':shared')
    }
    task spiJar(type: Jar) {
        baseName = 'api-spi'
        from sourceSets.main.output
        include('org/gradle/sample/api/**')
    }
    artifacts {
        spi spiJar
    }
}

project(':services:personService') {
    dependencies {
        compile project(':shared')
        compile project(path: ':api', configuration: 'spi')
        testCompile "junit:junit:4.12", project(':api')
    }
}
```

build.gradle.kts

```
subprojects {
    apply(plugin = "java")
    group = "org.gradle.sample"
    version = "1.0"
}

project(":api") {
    configurations {
        create("spi")
    }
    dependencies {
        "implementation"(project(":shared"))
    }
    task<Jar>("spiJar") {
        baseName = "api-spi"
        from(project.the<SourceSetContainer>()["main"].output)
        include("org/gradle/sample/api/**")
    }
    artifacts {
        add("spi", tasks["spiJar"])
    }
}

project(":services:personService") {
    dependencies {
        "implementation"(project(":shared"))
        "implementation"(project(path = ":api", configuration = "spi"))
        "testImplementation"("junit:junit:4.12")
        "testImplementation"(project(":api"))
    }
}
```

The Java plugin adds per default a jar to your project libraries which contains all the classes. In this example we create an *additional* library containing only the interfaces of the "`api`" project. We assign this library to a new *dependency configuration*. For the person service we declare that the project should be compiled only against the "`api`" interfaces but tested with all classes from "`api`".

Depending on the task output 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. The attribute `builtBy` takes care of establishing an inferred task dependency. For more information on `builtBy`, see SourceSetOutput.

Example 104. Task generating a property file containing build information

build.gradle

```
task buildInfo(type: BuildInfo) {
    version = project.version
    outputFile = file("$buildDir/generated-resources/build-info.properties")
}

sourceSets {
    main {
        output.dir(buildInfo.outputFile.parentFile, builtBy: buildInfo)
    }
}
```

build.gradle.kts

```
val buildInfo = task<BuildInfo>("buildInfo") {
    version = project.version.toString()
    outputFile = file("$buildDir/generated-resources/build-info.properties")
}

sourceSets {
    main {
        output.dir(buildInfo.outputFile.parentFile, "builtBy" to buildInfo)
    }
}
```

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 105. 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.

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.

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:personservice`" 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 106. 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
9 actionable tasks: 9 executed
```

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 lib dependencies of the testRuntime configuration.

Example 107. 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
12 actionable tasks: 12 executed
```

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 lib dependency (in the testRuntime configuration) on the specified project.

Example 108. 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:personService:compileJava
> Task :services:personService:processResources
> Task :services:personService:classes
> Task :services:personService:jar
> Task :services:personService:assemble
> Task :services:personService:compileTestJava
> Task :services:personService:processTestResources
> Task :services:personService:testClasses
> Task :services:personService:test
> Task :services:personService:check
> Task :services:personService:build
> Task :services:personService:buildDependents
> Task :api:buildDependents

BUILD SUCCESSFUL in 0s
17 actionable tasks: 17 executed
```

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.

Multi Project and buildSrc

Using buildSrc to organize build logic tells us that we can place build logic to be compiled and tested in the special `buildSrc` directory. In a multi project build, there can only be one `buildSrc` directory which must be located in the root directory.

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 *enhanced tasks*, which are tasks that have their own properties and methods. This is really different from what you are used to with Ant targets. Such enhanced 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	This API is likely to be replaced in the near future. See the task configuration avoidance chapter for more information.

Example 109. Defining tasks using strings for task names

build.gradle

```
task('hello') {
    doLast {
        println "hello"
    }
}

task('copy', type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

build.gradle.kts

```
task("hello") {
    doLast {
        println("hello")
    }
}

task<Copy>("copy") {
    from(file("srcDir"))
    into(buildDir)
}
```

There is an alternative syntax for defining tasks, which you may prefer to use:

Example 110. Defining tasks using the `tasks` container

build.gradle

```
tasks.create('hello') {
    doLast {
        println "hello"
    }
}

tasks.create('copy', Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

build.gradle.kts

```
tasks.create("hello") {
    doLast {
        println("hello")
    }
}

tasks {
    create<Copy>("copy") {
        from(file("srcDir"))
        into(buildDir)
    }
}
```

Here we add tasks to the `tasks` collection. Have a look at TaskContainer for more variations of the `create()` method.

And finally, there are language specific syntaxes for the Groovy and Kotlin DSL:

Example 111. Defining tasks using a DSL specific syntax

build.gradle

```
// Using Groovy dynamic keywords

task(hello) {
    doLast {
        println "hello"
    }
}

task(copy, type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

build.gradle.kts

```
// Using Kotlin delegated properties

val hello by tasks.creating {
    doLast {
        println("hello")
    }
}

val copy by tasks.creating(Copy::class) {
    from(file("srcDir"))
    into(buildDir)
}
```

Note that the Kotlin delegated properties syntax is particularly useful if you need the created task for further reference.

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:

Example 112. Accessing tasks using a DSL specific syntax

build.gradle

```
task hello
task copy(type: Copy)

// Access tasks using Groovy dynamic properties on Project

println hello.name
println project.hello.name

println copy.destinationDir
println project.copy.destinationDir
```

build.gradle.kts

```
task("hello")
task<Copy>("copy")

// Access tasks using Kotlin delegated properties

val hello by tasks.getting
println(hello.name)

val copy by tasks.getting(Copy::class)
println(copy.destinationDir)
```

Tasks are also available through the `tasks` collection.

Example 113. Accessing tasks via tasks collection

build.gradle

```
task hello
task copy(type: Copy)

println tasks['hello'].name
println tasks.hello.name

println tasks['copy'].destinationDir
println tasks.copy.destinationDir
```

build.gradle.kts

```
task("hello")
task<Copy>("copy")

println(tasks["hello"].name)
println(tasks.getByName("hello").name)

println(tasks.getByName<Copy>("copy").destinationDir)
```

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 114. Accessing tasks by path

build.gradle

```
project(':projectA') {
    task hello
}

task hello

println tasks.getByPath('hello').path
println tasks.getByPath(':hello').path
println tasks.getByPath('projectA:hello').path
println tasks.getByPath(':projectA:hello').path
```

build.gradle.kts

```
project(":projectA") {
    task("hello")
}

task("hello")

println(tasks.getByPath("hello").path)
println(tasks.getByPath(":hello").path)
println(tasks.getByPath("projectA:hello").path)
println(tasks.getByPath(":projectA:hello").path)
```

Output of `gradle -q hello`

```
> gradle -q hello
:hello
:hello
:projectA:hello
:projectA: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 create a `Copy` task for your build, you can declare in your build script:

Example 115. Creating a copy task

build.gradle

```
task myCopy(type: Copy)
```

build.gradle.kts

```
task<Copy>("myCopy")
```

This creates 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 116. Configuring a task using the API

build.gradle

```
Copy myCopy = tasks.getByName("myCopy")
myCopy.from 'resources'
myCopy.into 'target'
myCopy.include('**/*.txt', '**/*.xml', '**/*.properties')
```

build.gradle.kts

```
val myCopy = tasks.getByName<Copy>("myCopy")
myCopy.from("resources")
myCopy.into("target")
myCopy.include("**/*.txt", "**/*.xml", "**/*.properties")
```

This is similar to the way we would configure objects in Java. You have to repeat the context (`myCopy`) in the configuration statement every time. This is a redundancy and not very nice to read.

There is another way of configuring a task. It also preserves the context and it is arguably the most readable. It is usually our favorite.

Example 117. Configuring a task using a DSL specific syntax

build.gradle

```
// Configure task using Groovy dynamic task configuration block
myCopy {
   from 'resources'
   into 'target'
}
myCopy.include('**/*.txt', '**/*.xml', '**/*.properties')
```

build.gradle.kts

```
// Configure task using Kotlin delegated properties and a lambda
val myCopy by tasks.getting(Copy::class) {
    from("resources")
    into("target")
}
myCopy.include("**/*.txt", "**/*.xml", "**/*.properties")
```

This works for *any* task. Task access is just a shortcut for the `tasks.getByName()` method. It is important to note that if you pass a block to the `getByName()` method, this block is applied to *configure* the task, not when the task executes.

Have a look at TaskContainer for more options for configuring tasks.

You can also use a configuration block when you define a task.

Example 118. Defining a task with a configuration block

build.gradle

```
task copy(type: Copy) {
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

build.gradle.kts

```
task<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 119. Task class with `@Inject` constructor

build.gradle

```
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

```
open 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 120. Creating a task with constructor arguments using TaskContainer

build.gradle

```
tasks.create('myTask', CustomTask, 'hello', 42)
```

build.gradle.kts

```
tasks.create<CustomTask>("myTask", "hello", 42)
```

You can also create the task using a `constructorArgs` Map argument using the Project API:

Example 121. Creating a task with constructor arguments using Map

build.gradle

```
task myTask(type: CustomTask, constructorArgs: ['hello', 42])
```

build.gradle.kts

```
task("myTask", "type" to CustomTask::class.java, "constructorArgs" to
listOf("hello", 42))
```

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 `projectA:taskX` to `projectB:taskY`:

Example 122. Adding dependency on task from another project

build.gradle

```
project('projectA') {
    task taskX {
        dependsOn ':projectB:taskY'
        doLast {
            println 'taskX'
        }
    }
}

project('projectB') {
    task taskY {
        doLast {
            println 'taskY'
        }
    }
}
```

build.gradle.kts

```
project("projectA") {
    task("taskX") {
        dependsOn(":projectB:taskY")
        doLast {
            println("taskX")
        }
    }
}

project("projectB") {
    task("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 `Task` object, as shown in this example:

Example 123. Adding dependency using task object

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
    }
}

task taskY {
    doLast {
        println 'taskY'
    }
}

taskX.dependsOn taskY
```

build.gradle.kts

```
val taskX by tasks.creating {
    doLast {
        println("taskX")
    }
}

val taskY by tasks.creating {
    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 124. Adding dependency using a lazy block

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
    }
}

// Using a Groovy Closure
taskX.dependsOn {
    tasks.findAll { task -> task.name.startsWith('lib') }
}

task lib1 {
    doLast {
        println 'lib1'
    }
}

task lib2 {
    doLast {
        println 'lib2'
    }
}

task notALib {
    doLast {
        println 'notALib'
    }
}
```

build.gradle.kts

```
val taskX by tasks.creating {
    doLast {
        println("taskX")
    }
}

// Using a Gradle Provider
taskX.dependsOn(provider {
    tasks.filter { task -> task.name.startsWith("lib") }
})

task("lib1") {
    doLast {
        println("lib1")
    }
}

task("lib2") {
    doLast {
        println("lib2")
    }
}

task("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 125. Adding a 'must run after' task ordering

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
    }
}
task taskY {
    doLast {
        println 'taskY'
    }
}
taskY.mustRunAfter taskX
```

build.gradle.kts

```
val taskX by tasks.creating {
    doLast {
        println("taskX")
    }
}
val taskY by tasks.creating {
    doLast {
        println("taskY")
    }
}
taskY.mustRunAfter(taskX)
```

Output of `gradle -q taskY taskX`

```
> gradle -q taskY taskX
taskX
taskY
```

Example 126. Adding a 'should run after' task ordering

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
    }
}
task taskY {
    doLast {
        println 'taskY'
    }
}
taskY.shouldRunAfter taskX
```

build.gradle.kts

```
val taskX by tasks.creating {
    doLast {
        println("taskX")
    }
}
val taskY by tasks.creating {
    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 127. 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 128. A 'should run after' task ordering is ignored if it introduces an ordering cycle

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
    }
}
task taskY {
    doLast {
        println 'taskY'
    }
}
task taskZ {
    doLast {
        println 'taskZ'
    }
}
taskX.dependsOn taskY
taskY.dependsOn taskZ
taskZ.shouldRunAfter taskX
```

build.gradle.kts

```
val taskX by tasks.creating {
    doLast {
        println("taskX")
    }
}
val taskY by tasks.creating {
    doLast {
        println("taskY")
    }
}
val taskZ by tasks.creating {
    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 129. Adding a description to a task

build.gradle

```
task copy(type: Copy) {
   description 'Copies the resource directory to the target directory.'
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

build.gradle.kts

```
task<Copy>("copy") {
   description = "Copies the resource directory to the target directory."
   from("resources")
   into("target")
   include("**/*.txt", "**/*.xml", "**/*.properties")
}
```

Replacing tasks

Sometimes you want to replace a task. For example, if you want to exchange a task added by the Java plugin with a custom task of a different type. You can achieve this with:

Example 130. Overwriting a task

build.gradle

```
task copy(type: Copy)

task copy(overwrite: true) {
    doLast {
        println('I am the new one.')
    }
}
```

build.gradle.kts

```
task("copy")

task("copy", "overwrite" to true).apply {
    doLast {
        println("I am the new one.")
    }
}
```

Output of `gradle -q copy`

```
> gradle -q copy
I am the new one.
```

This will replace a task of type `Copy` with the task you've defined, because it uses the same name. When you define the new task, you have to set the `overwrite` property to true. Otherwise Gradle throws an exception, saying that a task with that name already exists.

Skipping tasks

Gradle offers multiple ways to skip the execution of a task.

Using a predicate

You can use the `onlyIf()` method to attach a predicate to a task. The task's actions are only executed if the predicate evaluates to true. You implement the predicate as a closure. The closure 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 due to be executed.

Example 131. Skipping a task using a predicate

build.gradle

```
task hello {
    doLast {
        println 'hello world'
    }
}

hello.onlyIf { !project.hasProperty('skipHello') }
```

build.gradle.kts

```
val hello by tasks.creating {
    doLast {
        println("hello world")
    }
}

hello.onlyIf { !project.hasProperty("skipHello") }
```

Output of `gradle hello -PskipHello`

```
> gradle hello -PskipHello
> Task :hello SKIPPED

BUILD SUCCESSFUL in 0s
```

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 132. Skipping tasks with StopExecutionException

build.gradle

```
task compile {
    doLast {
        println 'We are doing the compile.'
    }
}

compile.doFirst {
    // Here you would put arbitrary conditions in real life.
    // But this is used in an integration test so we want defined behavior.
    if (true) { throw new StopExecutionException() }
}
task myTask {
    dependsOn('compile')
    doLast {
        println 'I am not affected'
    }
}
```

build.gradle.kts

```
val compile by tasks.creating {
    doLast {
        println("We are doing the compile.")
    }
}

compile.doFirst {
    // Here you would put arbitrary conditions in real life.
    // But this is used in an integration test so we want defined behavior.
    if (true) { throw StopExecutionException() }
}
task ("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. [7: 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).]

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 133. Enabling and disabling tasks

build.gradle

```
task disableMe {
    doLast {
        println 'This should not be printed if the task is disabled.'
    }
}
disableMe.enabled = false
```

build.gradle.kts

```
val disableMe by tasks.creating {
    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 134. Specifying task timeouts

build.gradle

```
task hangingTask() {
    doLast {
        Thread.sleep(100000)
    }
    timeout = Duration.ofMillis(500)
}
```

build.gradle.kts

```
import java.time.Duration

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.

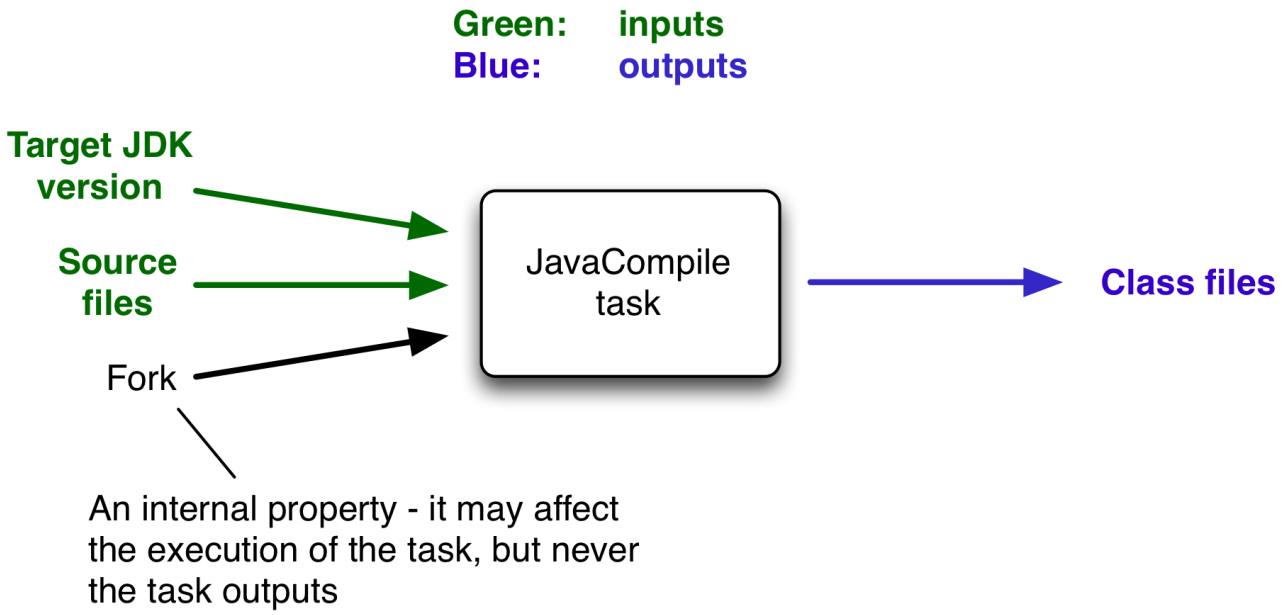


Figure 8. Example task inputs and outputs

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 have 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 three 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 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...), ProjectLayout.files(java.lang.Object...), and ProjectLayout.configurableFiles(java.lang.Object...) methods.

- 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 135. Custom task class

buildSrc/src/main/java/org/example/ProcessTemplates.java

```
package org.example;

import java.io.File;
import java.util.HashMap;
import org.gradle.api.*;
import org.gradle.api.file.*;
import org.gradle.api.tasks.*;

public class ProcessTemplates extends DefaultTask {
    private TemplateEngineType templateEngine;
    private FileCollection sourceFiles;
    private TemplateData templateData;
    private File outputDir;

    @Input
    public TemplateEngineType getTemplateEngine() {
        return this.templateEngine;
    }

    @InputFiles
    public FileCollection getSourceFiles() {
        return this.sourceFiles;
    }

    @Nested
    public TemplateData getTemplateData() {
        return this.templateData;
    }

    @OutputDirectory
    public File getOutputDir() { return this.outputDir; }

    // + setter methods for the above - assume we've defined them

    @TaskAction
    public void processTemplates() {
        // ...
    }
}
```

buildSrc/src/main/java/org/example/TemplateData.java

```
package org.example;

import java.util.HashMap;
import java.util.Map;
import org.gradle.api.tasks.Input;

public class TemplateData {
    private String name;
    private Map<String, String> variables;

    public TemplateData(String name, Map<String, String> variables) {
        this.name = name;
        this.variables = new HashMap<>(variables);
    }

    @Input
    public String getName() { return this.name; }

    @Input
    public Map<String, String> getVariables() {
        return this.variables;
    }
}
```

Output of `gradle processTemplates`

```
> gradle processTemplates
> Task :processTemplates


BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Output of `gradle processTemplates` *(run again)*

```
> gradle processTemplates
> Task :processTemplates UP-TO-DATE

BUILD SUCCESSFUL in 0s
1 actionable task: 1 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 have 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 2. Incremental build property type annotations

Annotation	Expected property type	Description
`@link:{javadocPath}/org/gradle/api/tasks/Input.html[Input]`	Any `Serializable` type	A simple input value
`@link:{javadocPath}/org/gradle/api/tasks/InputFile.html[InputFile]`	`File`*	A single input file (not directory)

Annotation	Expected property type	Description
`@link:{javadocPath}/org/gradle/api/tasks/InputDirectory.html[InputDirectory]`	`File`*	A single input directory (not file)
`@link:{javadocPath}/org/gradle/api/tasks/InputFiles.html[InputFiles]`	`Iterable<File>`*	An iterable of input files and directories
`@link:{javadocPath}/org/gradle/api/tasks/Classpath.html[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 <<#sec:task_input_using_classpath_annotations,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
`@link:{javadocPath}/org/gradle/api/tasks/CompileClasspath.html[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 <<#sec:task_input_using_classpath_annotations,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. [NOTE] ==== The `@CompileClasspath` annotation was introduced in Gradle 3.4. To stay compatible with Gradle 3.3 and 3.2, compile classpath properties should also be annotated with `@Classpath`. For compatibility with Gradle versions before 3.2 the property should also be annotated with `@InputFiles`. ====
`@link:{javadocPath}/org/gradle/api/tasks/OutputFile.html[OutputFile]`	`File`*	A single output file (not directory)
`@link:{javadocPath}/org/gradle/api/tasks/OutputDirectory.html[OutputDirectory]`	`File`*	A single output directory (not file)

Annotation	Expected property type	Description
`@link:{javadocPath}/org/gradle/api/tasks/OutputFiles.html[OutputFiles]`	`Map<String, File>`+++**+++ or `Iterable<File>`+++*+++	An iterable or map of output files. Using a file tree turns <<build_cache.adoc#sec:task_output_caching, caching>> off for the task.
`@link:{javadocPath}/org/gradle/api/tasks/OutputDirectories.html[OutputDirectories]`	`Map<String, File>`+++**+++ or `Iterable<File>`+++*+++	An iterable of output directories. Using a file tree turns <<build_cache.adoc#sec:task_output_caching, caching>> off for the task.
`@link:{javadocPath}/org/gradle/api/tasks/Destroys.html[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.
`@link:{javadocPath}/org/gradle/api/tasks/LocalState.html[LocalState]`	`File` or `Iterable<File>`+++*+++	Specifies one or more files that represent the <<custom_tasks.adoc#sec:storing_incremental_task_state,local state of the task>>. These files are removed when the task is loaded from cache.
`@link:{javadocPath}/org/gradle/api/tasks/Nested.html[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`.
`@link:{javadocPath}/org/gradle/api/tasks/Console.html[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.
`@link:{javadocPath}/org/gradle/api/tasks/Internal.html[Internal]`	Any type	Indicates that the property is used internally but is neither an input nor an output.

Annotation	Expected property type	Description
[#skip-when-empty]`@link:{javadocPath}/org/gradle/api/tasks/SkipWhenEmpty.html[SkipWhenEmpty]`	`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.
`@link:{javadocPath}/org/gradle/api/tasks/Optional.html[Optional]`	Any type	Used with any of the property type annotations listed in the link:{javadocPath}/org/gradle/api/tasks/Optional.html[Optional] API documentation. This annotation disables validation checks on the corresponding property. See <<#sec:task_input_output_validation,the section on validation>> for more details.
`@link:{javadocPath}/org/gradle/api/tasks/PathSensitive.html[PathSensitive]`	`File`+++*+++	[[inputs_path_sensitivity]]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.

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...), ProjectLayout.files(java.lang.Object...), or ProjectLayout.configurableFiles(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 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 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

In the same way, this kind of modelling is available to custom tasks.

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 136. Ad-hoc task

build.gradle

```
task processTemplatesAdHoc {
    inputs.property("engine", TemplateEngineType.FREEMARKER)
    inputs.files(fileTree("src/templates"))
    inputs.property("templateData.name", "docs")
    inputs.property("templateData.variables", [year: 2013])
    outputs.dir("$buildDir/genOutput2")

    doLast {
        // Process the templates here
    }
}
```

build.gradle.kts

```
task("processTemplatesAdHoc") {
    inputs.property("engine", TemplateEngineType.FREEMARKER)
    inputs.files(fileTree("src/templates"))
    inputs.property("templateData.name", "docs")
    inputs.property("templateData.variables", mapOf("year" to "2013"))
    outputs.dir("$buildDir/genOutput2")

    doLast {
        // Process the templates here
    }
}
```

Output of `gradle processTemplatesAdHoc`

```
> gradle processTemplatesAdHoc
> Task :processTemplatesAdHoc

BUILD SUCCESSFUL in 0s
1 actionable task: 1 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 137. Ad-hoc task declaring a destroyable

build.gradle

```
task removeTempDir {
    destroyables.register("$projectDir/tmpDir")
    doLast {
        delete("$projectDir/tmpDir")
    }
}
```

build.gradle.kts

```
task("removeTempDir") {
    destroyables.register("$projectDir/tmpDir")
    doLast {
        delete("$projectDir/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.

Using it for custom task types

Another type of example involves adding input and output definitions to instances of a custom task class that lacks the requisite annotations. For example, imagine that the `ProcessTemplates` task is provided by a plugin and that it's missing the incremental build annotations. In order to make up for that deficiency, you can use the runtime API:

Example 138. Using runtime API with custom task type

build.gradle

```
task processTemplatesRuntime(type: ProcessTemplatesNoAnnotations) {
    templateEngine = TemplateEngineType.FREEMARKER
    sourceFiles = fileTree("src/templates")
    templateData = new TemplateData("test", [year: 2014])
    outputDir = file("$buildDir/genOutput3")

    inputs.property("engine", templateEngine)
    inputs.files(sourceFiles)
    inputs.property("templateData.name", templateData.name)
    inputs.property("templateData.variables", templateData.variables)
    outputs.dir(outputDir)
}
```

build.gradle.kts

```
task<ProcessTemplatesNoAnnotations>("processTemplatesRuntime") {
    templateEngine = TemplateEngineType.FREEMARKER
    sourceFiles = fileTree("src/templates")
    templateData = TemplateData("test", mapOf("year" to "2014"))
    outputDir = file("$buildDir/genOutput3")

    inputs.property("engine", templateEngine)
    inputs.files(sourceFiles)
    inputs.property("templateData.name", templateData.name)
    inputs.property("templateData.variables", templateData.variables)
    outputs.dir(outputDir)
}
```

Output of `gradle processTemplatesRuntime`

```
> gradle processTemplatesRuntime
> Task :processTemplatesRuntime


BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Output of `gradle processTemplatesRuntime` *(run again)*

```
> gradle processTemplatesRuntime
> Task :processTemplatesRuntime UP-TO-DATE


BUILD SUCCESSFUL in 0s
1 actionable task: 1 up-to-date
```

As you can see, we can both configure the tasks properties and use those properties as arguments to the incremental build runtime API. 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. Note that if the task type is already using the incremental build annotations, the runtime API will add inputs and outputs rather than replace them.

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 139. Using skipWhenEmpty() via the runtime API

build.gradle

```
task processTemplatesRuntimeConf(type: ProcessTemplatesNoAnnotations) {
    // ...
    sourceFiles = fileTree("src/templates") {
        include "**/*.fm"
    }

    inputs.files(sourceFiles).skipWhenEmpty()
    // ...
}
```

build.gradle.kts

```
task<ProcessTemplatesNoAnnotations>("processTemplatesRuntimeConf") {
    // ...
    sourceFiles = fileTree("src/templates") {
        include("**/*.fm")
    }

    inputs.files(sourceFiles).skipWhenEmpty()
    // ...
}
```

Output of `gradle clean processTemplatesRuntimeConf`

```
> gradle clean processTemplatesRuntimeConf
> Task :processTemplatesRuntimeConf NO-SOURCE


BUILD SUCCESSFUL in 0s
1 actionable task: 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.

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 140. Inferred task dependency via task outputs

build.gradle

```
task packageFiles(type: Zip) {
    from processTemplates.outputs
}
```

build.gradle.kts

```
task<Zip>("packageFiles") {
    from(processTemplates.outputs)
}
```

Output of `gradle clean packageFiles`

```
> gradle clean packageFiles
> Task :processTemplates
> Task :packageFiles


BUILD SUCCESSFUL in 0s
3 actionable tasks: 2 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 141. Inferred task dependency via a task argument

build.gradle

```
task packageFiles2(type: Zip) {
    from processTemplates
}
```

build.gradle.kts

```
task<Zip>("packageFiles2") {
    from(processTemplates)
}
```

Output of `gradle clean packageFiles2`

```
> gradle clean packageFiles2
> Task :processTemplates
> Task :packageFiles2


BUILD SUCCESSFUL in 0s
3 actionable tasks: 2 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.

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.

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)

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 142. Declaring a method to add task inputs

build.gradle

```
task processTemplates(type: ProcessTemplates) {
    templateEngine = TemplateEngineType.FREEMARKER
    templateData = new TemplateData("test", [year: 2012])
    outputDir = file("$buildDir/genOutput")

    sources fileTree("src/templates")
}
```

build.gradle.kts

```
task<ProcessTemplates>("processTemplates") {
    templateEngine = TemplateEngineType.FREEMARKER
    templateData = TemplateData("test", mapOf("year" to "2012"))
    outputDir = file("$buildDir/genOutput")

    sources(fileTree("src/templates"))
}
```

ProcessTemplates.java

```
public class ProcessTemplates extends DefaultTask {
    // ...
    private FileCollection sourceFiles = getProject().getLayout().files();

    @SkipWhenEmpty
    @InputFiles
    @PathSensitive(PathSensitivity.NONE)
    public FileCollection getSourceFiles() {
        return this.sourceFiles;
    }

    public void sources(FileCollection sourceFiles) {
        this.sourceFiles = this.sourceFiles.plus(sourceFiles);
    }

    // ...
}
```

Output of `gradle processTemplates`

```
> gradle processTemplates
> Task :processTemplates


BUILD SUCCESSFUL in 0s
1 actionable task: 1 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 143. Declaring a method to add a task as an input

build.gradle

```
task copyTemplates(type: Copy) {
    into "$buildDir/tmp"
    from "src/templates"
}

task processTemplates2(type: ProcessTemplates) {
    // ...
    sources copyTemplates
}
```

build.gradle.kts

```
val copyTemplates by tasks.creating(Copy::class) {
    into("$buildDir/tmp")
    from("src/templates")
}

task<ProcessTemplates>("processTemplates2") {
    // ...
    sources(copyTemplates)
}
```

ProcessTemplates.java

```
    // ...
    public void sources(Task inputTask) {
        this.sourceFiles = this.sourceFiles.plus(getProject().getLayout().files
(inputTask));
    }
    // ...
```

Output of `gradle processTemplates2`

```
> gradle processTemplates2
> Task :copyTemplates
> Task :processTemplates2


BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 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 144. Failed attempt at setting up an inferred task dependency

build.gradle

```
plugins {
    id 'java'
}

task badInstrumentClasses(type: Instrument) {
    classFiles = fileTree(compileJava.destinationDir)
    destinationDir = file("$buildDir/instrumented")
}
```

build.gradle.kts

```
plugins {
    java
}

task<Instrument>("badInstrumentClasses") {
    classFiles = fileTree(tasks.compileJava.get().destinationDir)
    destinationDir = file("$buildDir/instrumented")
}
```

Output of `gradle clean badInstrumentClasses`

```
> gradle clean badInstrumentClasses
> Task :clean UP-TO-DATE
> Task :badInstrumentClasses NO-SOURCE


BUILD SUCCESSFUL in 0s
1 actionable task: 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 145. Setting up an inferred task dependency between output dir and input files

build.gradle

```
task instrumentClasses(type: Instrument) {
    classFiles = compileJava.outputs.files
    destinationDir = file("$buildDir/instrumented")
}
```

build.gradle.kts

```
task<Instrument>("instrumentClasses") {
    classFiles = tasks.compileJava.get().outputs.files
    destinationDir = file("$buildDir/instrumented")
}
```

Output of `gradle clean instrumentClasses`

```
> gradle clean instrumentClasses
> Task :clean UP-TO-DATE
> Task :compileJava
> Task :instrumentClasses


BUILD SUCCESSFUL in 0s
3 actionable tasks: 2 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.layout.configurableFiles()` in place of `project.fileTree()`:

Example 146. Setting up an inferred task dependency with filesFor()

build.gradle

```
task instrumentClasses2(type: Instrument) {
    classFiles = layout.files(compileJava)
    destinationDir = file("$buildDir/instrumented")
}
```

build.gradle.kts

```
task<Instrument>("instrumentClasses2") {
    classFiles = layout.files(tasks.compileJava.get())
    destinationDir = file("$buildDir/instrumented")
}
```

Output of `gradle clean instrumentClasses2`

```
> gradle clean instrumentClasses2
> Task :clean UP-TO-DATE
> Task :compileJava
> Task :instrumentClasses2


BUILD SUCCESSFUL in 0s
3 actionable tasks: 2 executed, 1 up-to-date
```

Remember that `files()`, `layout.files()` and `layout.configurableFiles()` 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 147. Setting up an inferred task dependency with builtBy()

build.gradle

```
task instrumentClassesBuiltBy(type: Instrument) {
    classFiles = fileTree(compileJava.destinationDir) {
        builtBy compileJava
    }
    destinationDir = file("$buildDir/instrumented")
}
```

build.gradle.kts

```
task<Instrument>("instrumentClassesBuiltBy") {
    classFiles = fileTree(tasks.compileJava.get().destinationDir) {
        builtBy(tasks.compileJava.get())
    }
    destinationDir = file("$buildDir/instrumented")
}
```

Output of `gradle clean instrumentClassesBuiltBy`

```
> gradle clean instrumentClassesBuiltBy
> Task :clean UP-TO-DATE
> Task :compileJava
> Task :instrumentClassesBuiltBy


BUILD SUCCESSFUL in 0s
3 actionable tasks: 2 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.

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. One use case is to disable up-to-date checks completely for a task, like so:

Example 148. Ignoring up-to-date checks

build.gradle

```
task alwaysInstrumentClasses(type: Instrument) {
    classFiles = layout.files(compileJava)
    destinationDir = file("$buildDir/instrumented")
    outputs.upToDateWhen { false }
}
```

build.gradle.kts

```
task<Instrument>("alwaysInstrumentClasses") {
    classFiles = layout.files(tasks.compileJava.get())
    destinationDir = file("$buildDir/instrumented")
    outputs.upToDateWhen { false }
}
```

Output of `gradle clean alwaysInstrumentClasses`

```
> gradle clean alwaysInstrumentClasses
> Task :compileJava
> Task :alwaysInstrumentClasses


BUILD SUCCESSFUL in 0s
3 actionable tasks: 2 executed, 1 up-to-date
```

Output of `gradle alwaysInstrumentClasses`

```
> gradle alwaysInstrumentClasses
> Task :compileJava UP-TO-DATE
> Task :alwaysInstrumentClasses


BUILD SUCCESSFUL in 0s
2 actionable tasks: 1 executed, 1 up-to-date
```

The `{ false }` closure ensures that `alwaysInstrumentClasses` will always be executed, irrespective of whether there is no change in the inputs or outputs.

You can of course put more complex logic into the closure. 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 an up-to-date check at all. 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.

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.

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):

Example 149. Runtime classpath normalization

build.gradle

```
normalization {
    runtimeClasspath {
        ignore 'build-info.properties'
    }
}
```

build.gradle.kts

```
normalization {
    runtimeClasspath {
        ignore("build-info.properties")
    }
}
```

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.

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 150. 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 151. 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')
            }
        }
    }
}

task 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", "")))
            }
        }
    }
}

task("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 152. Adding a task finalizer

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
    }
}
task taskY {
    doLast {
        println 'taskY'
    }
}

taskX.finalizedBy taskY
```

build.gradle.kts

```
val taskX by tasks.creating {
    doLast {
        println("taskX")
    }
}
val taskY by tasks.creating {
    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.

Example 153. Task finalizer for a failing task

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
        throw new RuntimeException()
    }
}
task taskY {
    doLast {
        println 'taskY'
    }
}

taskX.finalizedBy taskY
```

build.gradle.kts

```
val taskX by tasks.creating {
    doLast {
        println("taskX")
        throw RuntimeException()
    }
}
val taskY by tasks.creating {
    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/groovy/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
```

On the other hand, finalizer tasks are not executed if the finalized task didn't do any work, for example if it is considered up to date or if a dependent task fails.

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`)

Many Gradle plug-ins define their own lifecycle tasks to make it convenient to do specific things. When developing your own plugins, you should consider using your own lifecycle tasks or hooking

into some of the tasks already provided by Gradle. See the Java plugin tasks for an example.

Unless a lifecycle task has actions, its outcome is determined by its dependencies. If any of the task's dependencies are executed, the lifecycle task will be considered executed. If all of the task's 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.

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 3. 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)

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.

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 154. 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 155. 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.')
```

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.")
```

Use the typical SLF4J pattern to replace a placeholder with an actual value as part of the log message.

Example 156. 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 157. 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 158. 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 159. Configuring standard output capture for a task

build.gradle

```
task logInfo {
    logging.captureStandardOutput LogLevel.INFO
    doFirst {
        println 'A task message which is logged at INFO level'
    }
}
```

build.gradle.kts

```
task("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 160. 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.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

Standard Gradle plugins

There are a number of plugins included in the Gradle distribution. These are listed below.

Language plugins

These plugins add support for various languages which can be compiled for and executed in the JVM.

Plugin Id	Automatically applies	Description
`java`	`java-base`	Adds Java compilation, testing and bundling capabilities to a project. It serves as the basis for many of the other Gradle plugins. See also this tutorial on Java projects.
`groovy`	`java`, `groovy-base`	Adds support for building Groovy projects. See also this tutorial for Groovy projects.
`scala`	`java`, `scala-base`	Adds support for building Scala projects.
`antlr`	`java`	Adds support for generating parsers using Antlr.

Incubating language plugins

These plugins add support for various languages:

Plugin Id	Automatically applies	Description
`assembler`	-	Adds native assembly language capabilities to a project.
`c`	-	Adds C source compilation capabilities to a project.
`cpp`	-	Adds C++ source compilation capabilities to a project.
`objective-c`	-	Adds Objective-C source compilation capabilities to a project.
`objective-cpp`	-	Adds Objective-C++ source compilation capabilities to a project.
`windows-resources`	-	Adds support for including Windows resources in native binaries.

Integration plugins

These plugins provide some integration with various runtime technologies.

Plugin Id	Automatically applies	Works with	Description
application	java, distribution	-	Adds tasks for running and bundling a Java project as a command-line application.
distribution	-	-	Adds support for building ZIP and TAR distributions.
ear	-	java	Adds support for building J2EE applications.
ivy-publish	-	application, distribution, java, war	Provides a new DSL to support publishing artifacts to Ivy repositories, which improves on the existing DSL.
maven-publish	-	application, distribution, java, war	Provides a new DSL to support publishing artifacts to Maven repositories, which improves on the existing DSL.
maven	-	java, war	Adds support for publishing artifacts to Maven repositories using the *original* publishing mechanism available in Gradle 1.0. See also Legacy Publishing.
osgi	java-base	java	Adds support for building OSGi bundles.
war	java	-	Adds support for assembling web application WAR files.

Incubating integration plugins

These plugins provide some integration with various runtime technologies.

Plugin Id	Automatically applies	Description
java-library-distribution	java, distribution	Adds support for building ZIP and TAR distributions for a Java library.

Software development plugins

These plugins provide help with your software development process.

Plugin Id	Automatically applies	Works with	Description
announce	-	-	Publish messages to your favourite platforms, such as Twitter or Growl.
build-announcements	announce	-	Sends local announcements to your desktop about interesting events in the build lifecycle.

Plugin Id	Automatically applies	Works with	Description
build-dashboard	reporting-base	-	Generates build dashboard report.
checkstyle	java-base	-	Performs quality checks on your project's Java source files using Checkstyle and generates reports from these checks.
codenarc	groovy-base	-	Performs quality checks on your project's Groovy source files using CodeNarc and generates reports from these checks.
eclipse	-	java,groovy, scala	Generates files that are used by Eclipse IDE, thus making it possible to import the project into Eclipse. See also this tutorial for Java projects.
eclipse-wtp	-	ear, war	Does the same as the eclipse plugin plus generates eclipse WTP (Web Tools Platform) configuration files. After importing to eclipse your war/ear projects should be configured to work with WTP. See also this tutorial for Java projects.
findbugs	java-base	-	Performs quality checks on your project's Java source files using FindBugs and generates reports from these checks.
idea	-	java	Generates files that are used by Intellij IDEA IDE, thus making it possible to import the project into IDEA.
jacoco	reporting-base	java	Provides integration with the JaCoCo code coverage library for Java.
java-gradle-plugin	java		Assists with development of Gradle plugins by providing standard plugin build configuration and validation.
jdepend	java-base	-	Performs quality checks on your project's source files using JDepend and generates reports from these checks.
pmd	java-base	-	Performs quality checks on your project's Java source files using PMD and generates reports from these checks.
project-report	reporting-base	-	Generates reports containing useful information about your Gradle build.
signing	base	-	Adds the ability to digitally sign built files and artifacts.

Incubating software development plugins

These plugins provide help with your software development process.

Table 4. Software development plugins

Plugin Id	Automatically applies	Works with	Description
cunit	-	-	Adds support for running CUnit tests.

Plugin Id	Automatically applies	Works with	Description
`visual-studio`	-	native language plugins	Adds integration with Visual Studio.

Base plugins

These plugins form the basic building blocks which the other plugins are assembled from. They are available for you to use in your build files, and are listed here for completeness. However, be aware that they are not yet considered part of Gradle's public API. As such, these plugins are not documented in the user guide. You might refer to their API documentation to learn more about them.

Plugin Id	Description
`base`	Adds the standard lifecycle tasks and configures reasonable defaults for the archive tasks. See Base Plugin.
`java-base`	Adds the source sets concept to the project. Does not add any particular source sets.
`groovy-base`	Adds the Groovy source sets concept to the project.
`scala-base`	Adds the Scala source sets concept to the project.
`reporting-base`	Adds some shared convention properties to the project, relating to report generation.

Third party plugins

You can find a list of external plugins at the Gradle Plugins site.

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 161. 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 162. Declaring the JUnit dependency

build.gradle

```
dependencies {
    testImplementation 'junit:junit:4.12'
}
```

build.gradle.kts

```
dependencies {
    testImplementation("junit:junit:4.12")
}
```

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 Gradle runner in a Java JUnit test:

Example: Using GradleRunner with JUnit

BuildLogicFunctionalTest.java

```
import org.gradle.testkit.runner.BuildResult;
import org.gradle.testkit.runner.GradleRunner;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.TemporaryFolder;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Collections;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import static org.gradle.testkit.runner.TaskOutcome.*;

public class BuildLogicFunctionalTest {
    @Rule public final TemporaryFolder testProjectDir = new TemporaryFolder();
    private File settingsFile;
    private File buildFile;

    @Before
    public void setup() throws IOException {
        settingsFile = testProjectDir.newFile("settings.gradle");
        buildFile = testProjectDir.newFile("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.getRoot())
            .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 are written in the Groovy programming language, and as many plugins are implemented in Groovy, 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 Gradle runner in a Groovy Spock test:

Example: Using GradleRunner with Spock

BuildLogicFunctionalTest.groovy

```
import org.gradle.testkit.runner.GradleRunner
import static org.gradle.testkit.runner.TaskOutcome.*
import org.junit.Rule
import org.junit.rules.TemporaryFolder
import spock.lang.Specification

class BuildLogicFunctionalTest extends Specification {
    @Rule TemporaryFolder testProjectDir = new TemporaryFolder()
    File settingsFile
    File buildFile

    def setup() {
        settingsFile = testProjectDir.newFile('settings.gradle')
        buildFile = testProjectDir.newFile('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.root)
            .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.

For earlier versions of Gradle (before 2.13), it is possible to manually make the code under test available via some extra configuration. The following example demonstrates having the build generate a file containing the implementation classpath of the code under test, and making it available at test runtime.

Example 163. Making the code under test classpath available to the tests

build.gradle

```
// Write the plugin's classpath to a file to share with the tests
task createClasspathManifest {
    def outputDir = file("$buildDir/$name")

    inputs.files sourceSets.main.runtimeClasspath
    outputs.dir outputDir

    doLast {
        outputDir.mkdirs()
        file("$outputDir/plugin-classpath.txt").text = sourceSets.main
.runtimeClasspath.join("\n")
    }
}

// Add the classpath file to the test runtime classpath
dependencies {
    testRuntimeOnly files(createClasspathManifest)
}
```

build.gradle.kts

```
// Write the plugin's classpath to a file to share with the tests
task("createClasspathManifest") {
    val outputDir = file("$buildDir/$name")

    inputs.files(sourceSets.main.get().runtimeClasspath)
    outputs.dir(outputDir)

    doLast {
        outputDir.mkdirs()
        file("$outputDir/plugin-
classpath.txt").writeText(sourceSets.main.get().runtimeClasspath.joinToString
("\n"))
    }
}

// Add the classpath file to the test runtime classpath
dependencies {
    testRuntimeOnly(files(tasks["createClasspathManifest"]))
}
```

NOTE	The code for this example can be found at `samples/testKit/gradleRunner/manualClasspathInjection` in the '-all' distribution of Gradle.

The tests can then read this value, and inject the classpath into the test build by using the method GradleRunner.withPluginClasspath(java.lang.Iterable). This classpath is then available to use to locate plugins in a test build via the plugins DSL (see Plugins). Applying plugins with the plugins DSL requires the definition of a plugin identifier. The following is an example (in Groovy) of doing this from within a Spock Framework `setup()` method, which is analogous to a JUnit `@Before` method.

Example: Injecting the code under test classes into test builds

src/test/groovy/org/gradle/sample/BuildLogicFunctionalTest.groovy

```
    List<File> pluginClasspath

    def setup() {
        settingsFile = testProjectDir.newFile('settings.gradle')
        buildFile = testProjectDir.newFile('build.gradle')

        def pluginClasspathResource = getClass().classLoader.findResource("plugin-
classpath.txt")
        if (pluginClasspathResource == null) {
            throw new IllegalStateException("Did not find plugin classpath resource,
run `testClasses` build task.")
        }

        pluginClasspath = pluginClasspathResource.readLines().collect { new File(it) }
    }

    def "hello world task prints hello world"() {
        given:
        buildFile << """
            plugins {
                id 'org.gradle.sample.helloworld'
            }
        """

        when:
        def result = GradleRunner.create()
            .withProjectDir(testProjectDir.root)
            .withArguments('helloWorld')
            .withPluginClasspath(pluginClasspath)
            .build()

        then:
        result.output.contains('Hello world!')
        result.task(":helloWorld").outcome == SUCCESS
    }
```

NOTE	The code for this example can be found at `samples/testKit/gradleRunner/manualClasspathInjection` in the '-all' distribution of Gradle.

This approach works well when executing the functional tests as part of the Gradle build. When executing the functional tests from an IDE, there are extra considerations. Namely, the classpath manifest file points to the class files etc. generated by Gradle and not the IDE. This means that after making a change to the source of the code under test, the source must be recompiled by Gradle. Similarly, if the effective classpath of the code under test changes, the manifest must be regenerated. In either case, executing the `testClasses` task of the build will ensure that things are up to date.

Some IDEs provide a convenience option to delegate the "test classpath generation and execution" to the build. In IntelliJ you can find this option under Preferences… > Build, Execution, Deployment > Build Tools > Gradle > Runner > Delegate IDE build/run actions to gradle. Please consult the documentation of your IDE for more information.

Working with Gradle versions prior to 2.8

The GradleRunner.withPluginClasspath(java.lang.Iterable) method will not work when executing the build with a Gradle version earlier than 2.8 (see The version used to test), as this feature is not supported on such Gradle versions.

Instead, the code must be injected via the build script itself. The following sample demonstrates how this can be done.

Example: Injecting the code under test classes into test builds for Gradle versions prior to 2.8

src/test/groovy/org/gradle/sample/BuildLogicFunctionalTest.groovy

```
    List<File> pluginClasspath

    def setup() {
        settingsFile = testProjectDir.newFile('settings.gradle')
        buildFile = testProjectDir.newFile('build.gradle')

        def pluginClasspathResource = getClass().classLoader.findResource("plugin-
classpath.txt")
        if (pluginClasspathResource == null) {
            throw new IllegalStateException("Did not find plugin classpath resource,
run `testClasses` build task.")
        }

        pluginClasspath = pluginClasspathResource.readLines().collect { new File(it) }
    }

    def "hello world task prints hello world with pre Gradle 2.8"() {
        given:
        def classpathString = pluginClasspath
            .collect { it.absolutePath.replace('\\', '\\\\') } // escape backslashes
in Windows paths
            .collect { "'$it'" }
            .join(", ")

        buildFile << """
            buildscript {
                dependencies {
                    classpath files($classpathString)
                }
            }
            apply plugin: "org.gradle.sample.helloworld"
        """

        when:
        def result = GradleRunner.create()
            .withProjectDir(testProjectDir.root)
            .withArguments('helloWorld')
            .withGradleVersion("2.7")
            .build()

        then:
        result.output.contains('Hello world!')
        result.task(":helloWorld").outcome == SUCCESS
    }
```

NOTE	The code for this example can be found at `samples/testKit/gradleRunner/manualClasspathInjection` in the '-all' distribution of Gradle.

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 test compile 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 164. 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:1.1-groovy-2.4') {
        exclude module: 'groovy-all'
    }
}
```

build.gradle.kts

```
plugins {
    groovy
    `java-gradle-plugin`
}

dependencies {
    testImplementation("org.spockframework:spock-core:1.1-groovy-2.4") {
        exclude(module = "groovy-all")
    }
}
```

NOTE The code for this example can be found at `samples/testKit/gradleRunner/automaticClasspathInjectionQuickstart` in the '-all' distribution of Gradle.

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.root)
        .withArguments('helloWorld')
        .withPluginClasspath()
        .build()

    then:
    result.output.contains('Hello world!')
    result.task(":helloWorld").outcome == SUCCESS
}
```

NOTE	The code for this example can be found at `samples/testKit/gradleRunner/automaticClasspathInjectionQuickstart` in the '-all' distribution of Gradle.

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.

Example 165. Reconfiguring the classpath generation conventions of the Java Gradle Development plugin

build.gradle

```
plugins {
    id 'groovy'
    id 'java-gradle-plugin'
}

sourceSets {
    functionalTest {
        groovy {
            srcDir file('src/functionalTest/groovy')
        }
        resources {
            srcDir file('src/functionalTest/resources')
        }
        compileClasspath += sourceSets.main.output + configurations
.testRuntimeClasspath
        runtimeClasspath += output + compileClasspath
    }
}

task functionalTest(type: Test) {
    testClassesDirs = sourceSets.functionalTest.output.classesDirs
    classpath = sourceSets.functionalTest.runtimeClasspath
}

check.dependsOn functionalTest

gradlePlugin {
    testSourceSets sourceSets.functionalTest
}

dependencies {
    functionalTestCompile('org.spockframework:spock-core:1.1-groovy-2.4') {
        exclude module: 'groovy-all'
    }
}
```

build.gradle.kts

```
plugins {
    groovy
    `java-gradle-plugin`
}

sourceSets {
    create("functionalTest") {
        withConvention(GroovySourceSet::class) {
            groovy {
                srcDir(file("src/functionalTest/groovy"))
            }
        }
        resources {
            srcDir(file("src/functionalTest/resources"))
        }
        compileClasspath += sourceSets.main.get().output +
configurations.testRuntimeClasspath
        runtimeClasspath += output + compileClasspath
    }
}

task<Test>("functionalTest") {
    testClassesDirs = sourceSets["functionalTest"].output.classesDirs
    classpath = sourceSets["functionalTest"].runtimeClasspath
}

tasks.check { dependsOn(tasks["functionalTest"]) }

gradlePlugin {
    testSourceSets(sourceSets["functionalTest"])
}

dependencies {
    "functionalTestImplementation"("org.spockframework:spock-core:1.1-groovy-
2.4") {
        exclude(module = "groovy-all")
    }
}
```

NOTE The code for this example can be found at `samples/testKit/gradleRunner/automaticClasspathInjectionCustomTestSourceSet` in the '-all' distribution of Gradle.

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 environment variables etc. 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 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 org.junit.Rule
import org.junit.rules.TemporaryFolder
import spock.lang.Specification
import spock.lang.Unroll

class BuildLogicFunctionalTest extends Specification {
    @Rule final TemporaryFolder testProjectDir = new TemporaryFolder()
    File settingsFile
    File buildFile

    def setup() {
        settingsFile = testProjectDir.newFile('settings.gradle')
        buildFile = testProjectDir.newFile('build.gradle')
    }

    @Unroll
    def "can execute hello world task with Gradle version #gradleVersion"() {
        given:
        buildFile << """
            task helloWorld {
                doLast {
                    logger.quiet 'Hello world!'
                }
            }
        """

        when:
        def result = GradleRunner.create()
            .withGradleVersion(gradleVersion)
            .withProjectDir(testProjectDir.root)
            .withArguments('helloWorld')
            .build()

        then:
        result.output.contains('Hello world!')
        result.task(":helloWorld").outcome == SUCCESS

        where:
        gradleVersion << ['2.6', '2.7']
    }
}
```

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 5. 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.

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.root, '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

```
@Rule final TemporaryFolder testProjectDir = new TemporaryFolder()
File buildFile
File localBuildCacheDirectory

def setup() {
    localBuildCacheDirectory = testProjectDir.newFolder('local-cache')
    testProjectDir.newFile('settings.gradle') << """
        buildCache {
            local {
                directory '${localBuildCacheDirectory.toURI()}'
            }
        }
    """
    buildFile = testProjectDir.newFile('build.gradle')
}
```

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, *script* plugins and *binary* plugins. 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. Binary plugins are classes that implement the Plugin interface and adopt a programmatic approach to manipulating the build. Binary plugins can reside within a build script, within the project hierarchy or externally in a plugin jar.

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 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.

Script plugins

Example 166. 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.

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 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 167. 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 168. Applying a community plugin

build.gradle

```
plugins {
    id 'com.jfrog.bintray' version '0.4.1'
}
```

build.gradle.kts

```
plugins {
    id("com.jfrog.bintray") version "0.4.1"
}
```

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.
- Allow different plugins to use different versions of dependencies.

- 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).

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

The plugins {} block can currently only be used in a project's build script. It cannot be used in script plugins, the settings.gradle file 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 plugins 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 or master 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 apply plugin: «plugin id» in the subprojects block or use the plugins {} block in sub projects build scripts:

Example 169. Applying plugins only on certain subprojects

settings.gradle

```
include 'helloA'
include 'helloB'
include 'goodbyeC'
```

build.gradle

```
plugins {
    id 'org.gradle.sample.hello' version '1.0.0' apply false
    id 'org.gradle.sample.goodbye' version '1.0.0' apply false
}

subprojects {
    if (name.startsWith('hello')) {
        apply plugin: 'org.gradle.sample.hello'
    }
}
```

goodbyeC/build.gradle

```
plugins {
    id 'org.gradle.sample.goodbye'
}
```

settings.gradle.kts

```
include("helloA")
include("helloB")
include("goodbyeC")
```

build.gradle.kts

```
plugins {
    id("org.gradle.sample.hello") version "1.0.0" apply false
    id("org.gradle.sample.goodbye") version "1.0.0" apply false
}

subprojects {
    if (name.startsWith("hello")) {
        apply(plugin = "org.gradle.sample.hello")
    }
}
```

goodbyeC/build.gradle.kts

```
plugins {
    id("org.gradle.sample.goodbye")
}
```

If you then run `gradle hello` you'll see that only the helloA and helloB subprojects had the hello plugin applied.

Example 170. Output of `gradle hello`

```
> gradle hello
:helloA:hello
:helloB:hello
Hello!
Hello!

BUILD SUCCEEDED
```

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 171. Defining a buildSrc plugin with an ID

buildSrc/build.gradle

```
plugins {
    id 'java'
    id 'java-gradle-plugin'
}

gradlePlugin {
    plugins {
        myPlugins {
            id = 'my-plugin'
            implementationClass = 'my.MyPlugin'
        }
    }
}

dependencies {
    compileOnly gradleApi()
}
```

buildSrc/build.gradle.kts

```
plugins {
    java
    `java-gradle-plugin`
}

gradlePlugin {
    plugins {
        create("myPlugins") {
            id = "my-plugin"
            implementationClass = "my.MyPlugin"
        }
    }
}

dependencies {
    compileOnly(gradleApi())
}
```

The plugin can then be applied by ID as normal:

Example 172. 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 173. Configuring pluginManagement per-project and globally

settings.gradle

```
pluginManagement {
    resolutionStrategy {
    }
    repositories {
    }
}
```

init.gradle

```
settingsEvaluated { settings ->
    settings.pluginManagement {
        resolutionStrategy {
        }
        repositories {
        }
    }
}
```

settings.gradle.kts

```
pluginManagement {
    resolutionStrategy {
    }
    repositories {
    }
}
```

init.gradle.kts

```
settingsEvaluated { settings ->
    settings.pluginManagement {
        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 174. 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 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 175. Plugin resolution strategy.

settings.gradle

```
pluginManagement {
    resolutionStrategy {
        eachPlugin {
            if (requested.id.namespace == 'org.gradle.sample') {
                useModule('org.gradle.sample: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 == "org.gradle.sample") {
                useModule("org.gradle.sample: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 `org.gradle.sample.hello` plugin and a `org.gradle.sample.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 176. Complete Plugin Publishing Sample

build.gradle

```
plugins {
    id 'java-gradle-plugin'
    id 'maven-publish'
    id 'ivy-publish'
}

group 'org.gradle.sample'
version '1.0.0'

gradlePlugin {
    plugins {
        hello {
            id = 'org.gradle.sample.hello'
            implementationClass = 'org.gradle.sample.hello.HelloPlugin'
        }
        goodbye {
            id = 'org.gradle.sample.goodbye'
            implementationClass = 'org.gradle.sample.goodbye.GoodbyePlugin'
        }
    }
}

publishing {
    repositories {
        maven {
            url '../../consuming/maven-repo'
        }
        ivy {
            url '../../consuming/ivy-repo'
        }
    }
}
```

build.gradle.kts

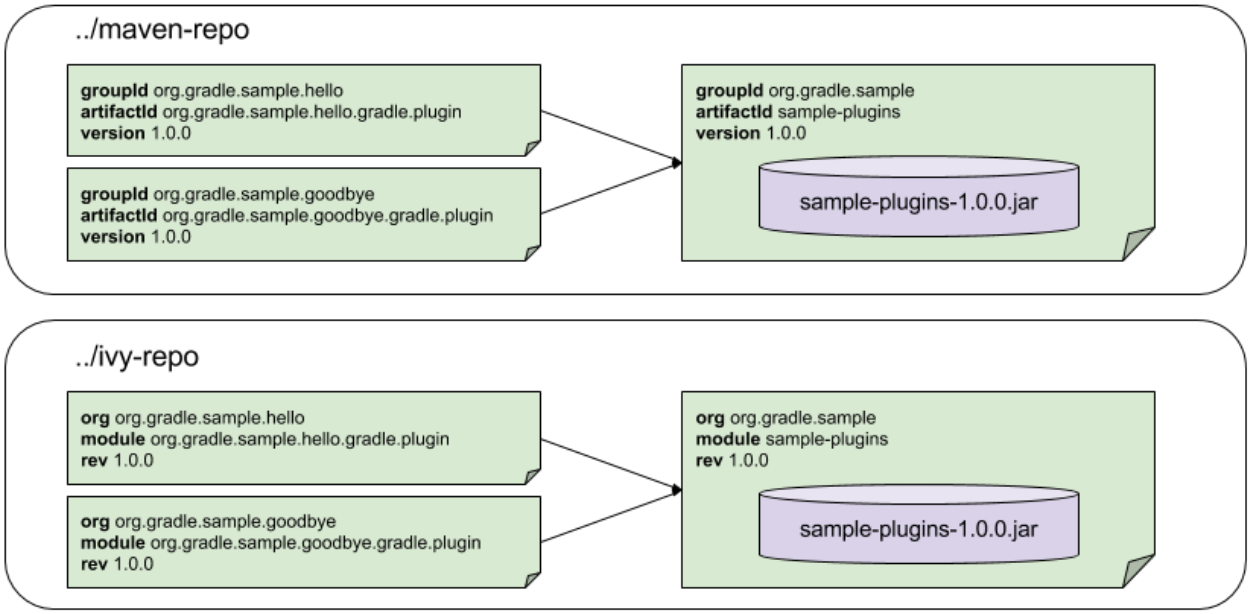
```
plugins {
    `java-gradle-plugin`
    `maven-publish`
    `ivy-publish`
}

group = "org.gradle.sample"
version = "1.0.0"

gradlePlugin {
    plugins {
        create("hello") {
            id = "org.gradle.sample.hello"
            implementationClass = "org.gradle.sample.hello.HelloPlugin"
        }
        create("goodbye") {
            id = "org.gradle.sample.goodbye"
            implementationClass = "org.gradle.sample.goodbye.GoodbyePlugin"
        }
    }
}

publishing {
    repositories {
        maven {
            url = uri("../../consuming/maven-repo")
        }
        ivy {
            url = uri("../../consuming/ivy-repo")
        }
    }
}
```

Running `gradle publish` in the sample directory causes the following repo layouts to exist:



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 177. 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 178. 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 179. Applying a plugin with the buildscript block

build.gradle

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.jfrog.bintray.gradle:gradle-bintray-plugin:0.4.1'
    }
}

apply plugin: 'com.jfrog.bintray'
```

build.gradle.kts

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath("com.jfrog.bintray.gradle:gradle-bintray-plugin:0.4.1")
    }
}

apply(plugin = "com.jfrog.bintray")
```

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.

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 180. How to copy a single file

build.gradle

```
task copyReport(type: Copy) {
    from file("$buildDir/reports/my-report.pdf")
    into file("$buildDir/toArchive")
}
```

build.gradle.kts

```
task<Copy>("copyReport") {
    from(file("$buildDir/reports/my-report.pdf"))
    into(file("$buildDir/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 181. Using implicit string paths

build.gradle

```
task copyReport2(type: Copy) {
    from "$buildDir/reports/my-report.pdf"
    into "$buildDir/toArchive"
}
```

build.gradle.kts

```
task<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 182. Prefer task/project properties over hard-coded paths

build.gradle

```
task copyReport3(type: Copy) {
    from myReportTask.outputFile
    into archiveReportsTask.dirToArchive
}
```

build.gradle.kts

```
task<Copy>("copyReport3") {
    val outputFile: File by myReportTask.extra
    val dirToArchive: File by archiveReportsTask.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 183. Using multiple arguments with from()

build.gradle

```
task copyReportsForArchiving(type: Copy) {
    from "$buildDir/reports/my-report.pdf", "src/docs/manual.pdf"
    into "$buildDir/toArchive"
}
```

build.gradle.kts

```
task<Copy>("copyReportsForArchiving") {
    from("$buildDir/reports/my-report.pdf", "src/docs/manual.pdf")
    into("$buildDir/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 184. Using a flat filter

build.gradle

```
task copyPdfReportsForArchiving(type: Copy) {
    from "$buildDir/reports"
    include "*.pdf"
    into "$buildDir/toArchive"
}
```

build.gradle.kts

```
task<Copy>("copyPdfReportsForArchiving") {
    from("$buildDir/reports")
    include("*.pdf")
    into("$buildDir/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:

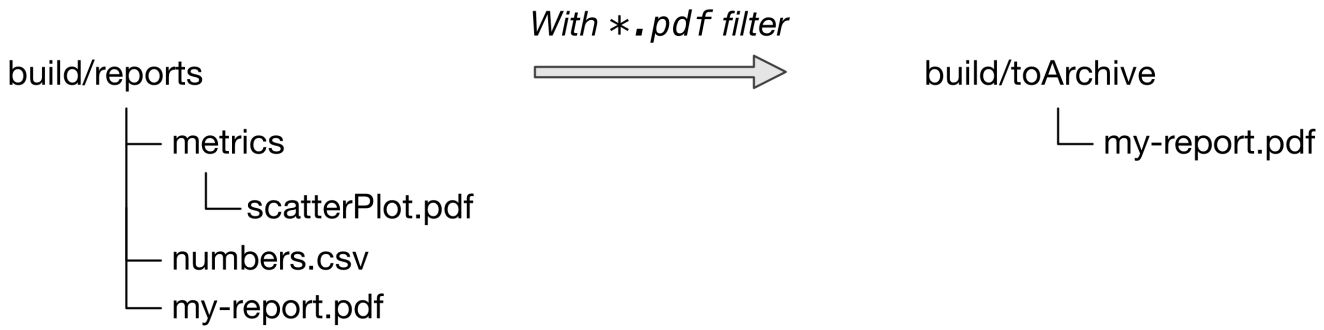


Figure 9. The effect of a flat filter on copying

You can include files in subdirectories by using an Ant-style glob pattern (`**/*`), as done in this updated example:

Example 185. Using a deep filter

build.gradle

```
task copyAllPdfReportsForArchiving(type: Copy) {
    from "$buildDir/reports"
    include "**/*.pdf"
    into "$buildDir/toArchive"
}
```

build.gradle.kts

```
task<Copy>("copyAllPdfReportsForArchiving") {
    from("$buildDir/reports")
    include("**/*.pdf")
    into("$buildDir/toArchive")
}
```

This task has the following effect:

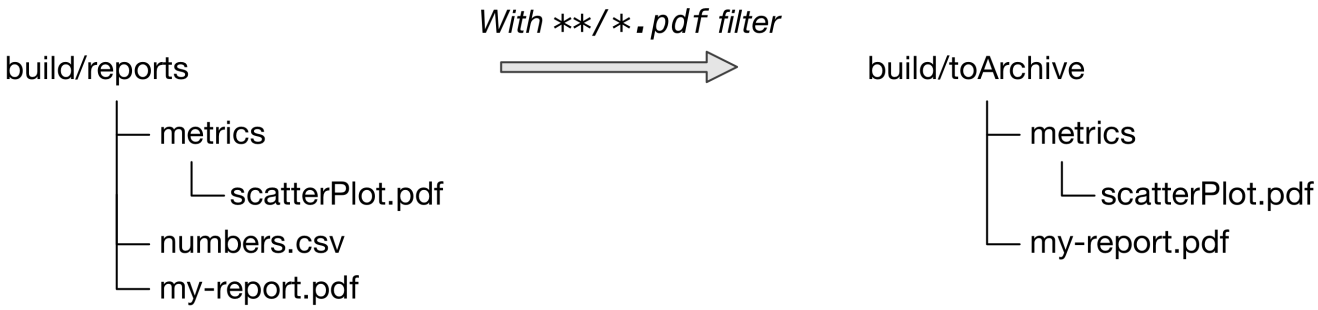


Figure 10. The effect of a deep filter on copying

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 186. Copying an entire directory

build.gradle

```
task copyReportsDirForArchiving(type: Copy) {
    from "$buildDir/reports"
    into "$buildDir/toArchive"
}
```

build.gradle.kts

```
task<Copy>("copyReportsDirForArchiving") {
    from("$buildDir/reports")
    into("$buildDir/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 187. Copying an entire directory, including itself

build.gradle

```
task copyReportsDirForArchiving2(type: Copy) {
    from("$buildDir") {
        include "reports/**"
    }
    into "$buildDir/toArchive"
}
```

build.gradle.kts

```
task<Copy>("copyReportsDirForArchiving2") {
    from("$buildDir") {
        include("reports/**")
    }
    into("$buildDir/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 188. Archiving a directory as a ZIP

build.gradle

```
task packageDistribution(type: Zip) {
    archiveName = "my-distribution.zip"
    destinationDir = file("$buildDir/dist")

    from "$buildDir/toArchive"
}
```

build.gradle.kts

```
task<Zip>("packageDistribution") {
    archiveName = "my-distribution.zip"
    destinationDir = file("$buildDir/dist")

    from("$buildDir/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 189. Using the Base Plugin for its archive name convention

build.gradle

```
plugins {
    id 'base'
}

version = "1.0.0"

task packageDistribution(type: Zip) {
    from("$buildDir/toArchive") {
        exclude "**/*.pdf"
    }

    from("$buildDir/toArchive") {
        include "**/*.pdf"
        into "docs"
    }
}
```

build.gradle.kts

```
plugins {
    base
}

version = "1.0.0"

task<Zip>("packageDistribution") {
    from("$buildDir/toArchive") {
        exclude("**/*.pdf")
    }

    from("$buildDir/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 190. Unpacking a ZIP file

build.gradle

```
task unpackFiles(type: Copy) {
    from zipTree("src/resources/thirdPartyResources.zip")
    into "$buildDir/resources"
}
```

build.gradle.kts

```
task<Copy>("unpackFiles") {
    from(zipTree("src/resources/thirdPartyResources.zip"))
    into("$buildDir/resources")
}
```

As with a normal copy, you can control which files are unpacked via filters and even rename files as they are unpacked.

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 191. 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'
}

task uberJar(type: Jar) {
    appendix = '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")
}

task<Jar>("uberJar") {
    appendix = "uber"

    from(sourceSets.main.get().output)

    dependsOn(configurations.runtimeClasspath)
    from(Callable {
        configurations.runtimeClasspath.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 guide. 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 192. Manually creating a directory

build.gradle

```
task ensureDirectory {
    doLast {
        mkdir "images"
    }
}
```

build.gradle.kts

```
task("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 193. Moving a directory using the Ant task

build.gradle

```
task moveReports {
    doLast {
        ant.move file: "${buildDir}/reports",
                 todir: "${buildDir}/toArchive"
    }
}
```

build.gradle.kts

```
task("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 194. Renaming files as they are copied

build.gradle

```
task copyFromStaging(type: Copy) {
    from "src/main/webapp"
    into "$buildDir/explodedWar"

    rename '(.+)-staging(.+)', '$1$2'
}
```

build.gradle.kts

```
task<Copy>("copyFromStaging") {
    from("src/main/webapp")
    into("$buildDir/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 195. Truncating filenames as they are copied

build.gradle

```
task copyWithTruncate(type: Copy) {
    from "$buildDir/reports"
    rename { String filename ->
        if (filename.size() > 10) {
            return filename[0..7] + "~" + filename.size()
        }
        else return filename
    }
    into "$buildDir/toArchive"
}
```

build.gradle.kts

```
task<Copy>("copyWithTruncate") {
    from("$buildDir/reports")
    rename { filename: String ->
        if (filename.length > 10) {
            filename.slice(0..7) + "~" + filename.length
        }
        else filename
    }
    into("$buildDir/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...), ProjectLayout.files(java.lang.Object...), and ProjectLayout.configurableFiles(java.lang.Object...) methods.

For example, the following task deletes the entire contents of a build's output directory:

Example 196. Deleting a directory

build.gradle

```
task myClean(type: Delete) {
    delete buildDir
}
```

build.gradle.kts

```
task<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 197. Deleting files matching a specific pattern

build.gradle

```
task cleanTempFiles(type: Delete) {
    delete fileTree("src").matching {
        include "**/*.tmp"
    }
}
```

build.gradle.kts

```
task<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 198. How to minimize the number of hard-coded paths in your build

build.gradle

```
ext {
    archivesDirPath = "$buildDir/archives"
}

task packageClasses(type: Zip) {
    appendix = "classes"
    destinationDir = file(archivesDirPath)

    from compileJava
}
```

build.gradle.kts

```
val archivesDirPath by extra { "$buildDir/archives" }

task<Zip>("packageClasses") {
    appendix = "classes"
    destinationDir = file(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 `destinationDir` 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)`, as this creates a path relative to the current working directory, which could be anywhere.

Here are some examples of using the `file()` method with different types of argument:

Example 199. 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 200. 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.

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 201. 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 202. Implementing a file collection

build.gradle

```
task 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

```
task("list") {
    doLast {
        var srcDir: File? = null

        val collection = layout.files(provider {
            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) of `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 203. 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 204. 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:

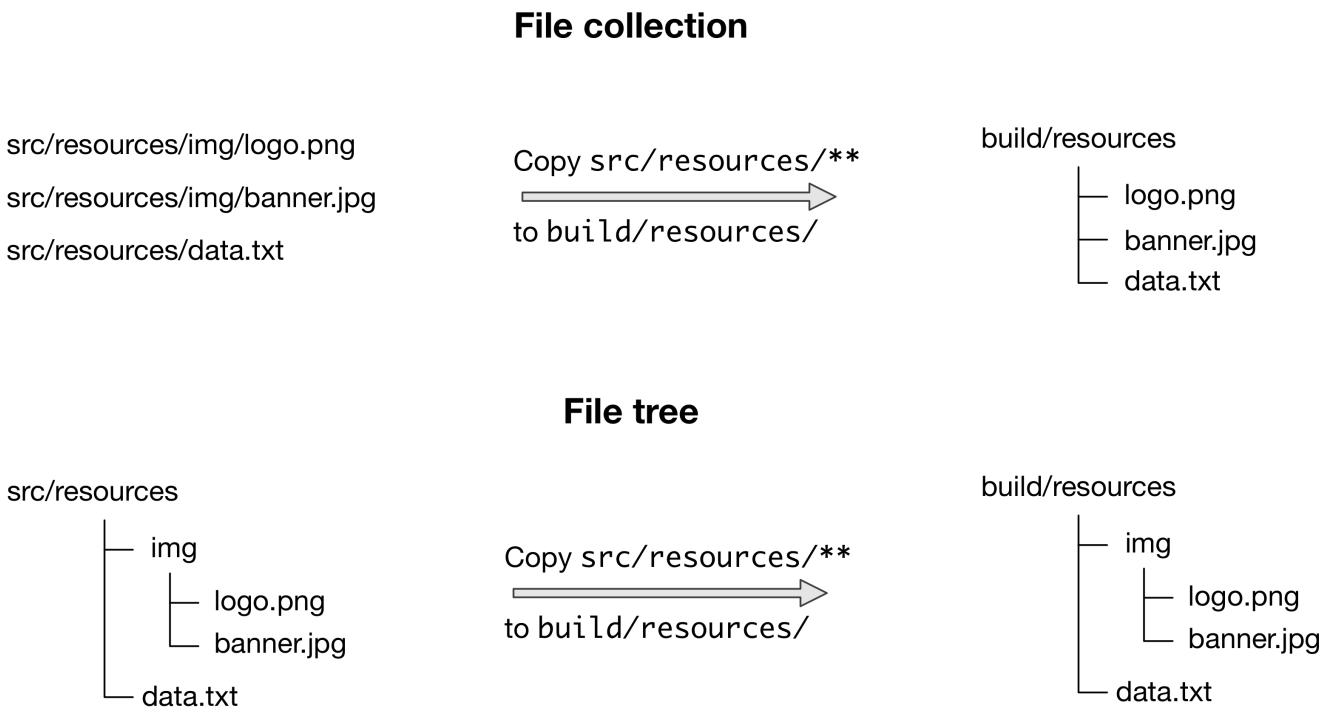


Figure 11. The differences in how file trees and file collections behave when copying files

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 205. 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 exclusion patterns for convenience — the same defaults as Ant in fact. For the complete default exclusion list, see the Ant manual.

If those default exclusions prove problematic, you can workaround the issue by using the `defaultexcludes` Ant task, as demonstrated in this example:

Example 206. Changing Ant default exclusions for a copy task

build.gradle

```
task forcedCopy(type: Copy) {
    into "$buildDir/inPlaceApp"
    from 'src/main/webapp'

    doFirst {
        ant.defaultexcludes remove: "**/.git"
        ant.defaultexcludes remove: "**/.git/**"
        ant.defaultexcludes remove: "**/*~"
    }

    doLast {
        ant.defaultexcludes default: true
    }
}
```

build.gradle.kts

```
task<Copy>("forcedCopy") {
    into("$buildDir/inPlaceApp")
    from("src/main/webapp")

    doFirst {
        ant.withGroovyBuilder {
            "defaultexcludes"("remove" to "**/.git")
            "defaultexcludes"("remove" to "**/.git/**")
            "defaultexcludes"("remove" to "**/*~")
        }
    }

    doLast {
        ant.withGroovyBuilder {
            "defaultexcludes"("default" to true)
        }
    }
}
```

In general, it's best to ensure that the default exclusions are reset whenever you change them as modifications are visible to the entire build. The above example is performing such a reset in its `doLast` action.

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 207. 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 208. 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 209. Specifying a set of files

build.gradle

```
task compile(type: 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

```
task<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(provider {
        // 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 210. 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.getByName<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(provider { 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…), ProjectLayout.files(java.lang.Object…), and ProjectLayout.configurableFiles(java.lang.Object…), so see those methods 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 211. Specifying copy task source files and destination directory

build.gradle

```
task anotherCopyTask(type: 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

```
task<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(Callable { 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 212. Selecting the files to copy

build.gradle

```
task copyTaskWithPatterns(type: Copy) {
    from 'src/main/webapp'
    into "$buildDir/explodedWar"
    include '**/*.html'
    include '**/*.jsp'
    exclude { FileTreeElement details ->
        details.file.name.endsWith('.html') &&
            details.file.text.contains('DRAFT')
    }
}
```

build.gradle.kts

```
task<Copy>("copyTaskWithPatterns") {
    from("src/main/webapp")
    into("$buildDir/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 213. Renaming files as they are copied

build.gradle

```
task rename(type: Copy) {
    from 'src/main/webapp'
    into "$buildDir/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

```
task<Copy>("rename") {
    from("src/main/webapp")
    into("$buildDir/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 214. Filtering files as they are copied

build.gradle

```
import org.apache.tools.ant.filters.FixCrLfFilter
import org.apache.tools.ant.filters.ReplaceTokens

task filter(type: Copy) {
    from 'src/main/webapp'
    into "$buildDir/explodedWar"
    // Substitute property tokens in files
    expand(copyright: '2009', version: '2.3.1')
    expand(project.properties)
    // Use some of the filters provided by Ant
    filter(FixCrLfFilter)
    filter(ReplaceTokens, tokens: [copyright: '2009', version: '2.3.1'])
    // Use a closure to filter each line
    filter { String line ->
        "[$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
task<Copy>("filter") {
    from("src/main/webapp")
    into("$buildDir/explodedWar")
    // Substitute property tokens in files
    expand("copyright" to "2009", "version" to "2.3.1")
    expand(project.properties)
    // Use some of the filters provided by Ant
    filter(FixCrLfFilter::class)
    filter(ReplaceTokens::class, "tokens" to mapOf("copyright" to "2009",
"version" to "2.3.1"))
    // Use a closure to filter each line
    filter { line: String ->
        "[$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 215. Sharing copy specifications

build.gradle

```
CopySpec webAssetsSpec = copySpec {
    from 'src/main/webapp'
    include '**/*.html', '**/*.png', '**/*.jpg'
    rename '(.+)-staging(.+)', '$1$2'
}

task copyAssets(type: Copy) {
    into "$buildDir/inPlaceApp"
    with webAssetsSpec
}

task distApp(type: Zip) {
    archiveName = 'my-app-dist.zip'
    destinationDir = file("$buildDir/dists")

    from appClasses
    with webAssetsSpec
}
```

build.gradle.kts

```
val webAssetsSpec: CopySpec = copySpec {
    from("src/main/webapp")
    include("**/*.html", "**/*.png", "**/*.jpg")
    rename("(.+)-staging(.+)", "$1$2")
}

task<Copy>("copyAssets") {
    into("$buildDir/inPlaceApp")
    with(webAssetsSpec)
}

task<Zip>("distApp") {
    archiveName = "my-app-dist.zip"
    destinationDir = file("$buildDir/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 216. Sharing copy patterns only

build.gradle

```
def webAssetPatterns = {
    include '**/*.html', '**/*.png', '**/*.jpg'
}

task copyAppAssets(type: Copy) {
    into "$buildDir/inPlaceApp"
    from 'src/main/webapp', webAssetPatterns
}

task archiveDistAssets(type: Zip) {
    archiveName = 'distribution-assets.zip'
    destinationDir = file("$buildDir/dists")

    from 'distResources', webAssetPatterns
}
```

build.gradle.kts

```
val webAssetPatterns = Action<CopySpec> {
    include("**/*.html", "**/*.png", "**/*.jpg")
}

task<Copy>("copyAppAssets") {
    into("$buildDir/inPlaceApp")
    from("src/main/webapp", webAssetPatterns)
}

task<Zip>("archiveDistAssets") {
    archiveName = "distribution-assets.zip"
    destinationDir = file("$buildDir/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:

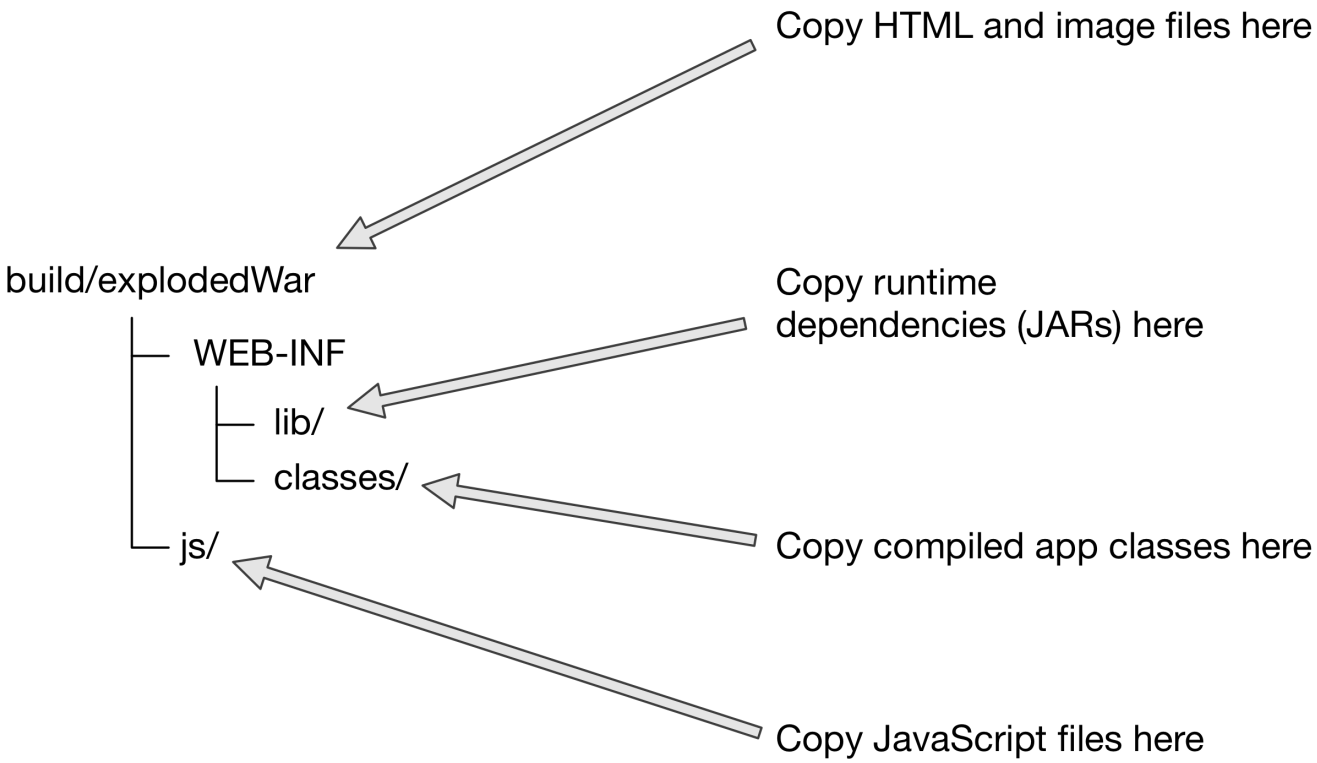


Figure 12. Creating an exploded WAR for a Servlet container

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 217. Nested copy specs

build.gradle

```
task nestedSpecs(type: Copy) {
    into "$buildDir/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

```
task<Copy>("nestedSpecs") {
    into("$buildDir/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 218. Copying files using the copy() method without up-to-date check

build.gradle

```
task copyMethod {
    doLast {
        copy {
            from 'src/main/webapp'
            into "$buildDir/explodedWar"
            include '**/*.html'
            include '**/*.jsp'
        }
    }
}
```

build.gradle.kts

```
task("copyMethod") {
    doLast {
        copy {
            from("src/main/webapp")
            into("$buildDir/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 219. Copying files using the copy() method with up-to-date check

build.gradle

```
task copyMethodWithExplicitDependencies{
    // up-to-date check for inputs, plus add copyTask as dependency
    inputs.files copyTask
    outputs.dir 'some-dir' // up-to-date check for outputs
    doLast{
        copy {
            // Copy the output of copyTask
            from copyTask
            into 'some-dir'
        }
    }
}
```

build.gradle.kts

```
task("copyMethodWithExplicitDependencies") {
    // up-to-date check for inputs, plus add copyTask as dependency
    inputs.files(copyTask)
    outputs.dir("some-dir") // up-to-date check for outputs
    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 220. Using the Sync task to copy dependencies

build.gradle

```
task libs(type: Sync) {
    from configurations.runtime
    into "$buildDir/libs"
}
```

build.gradle.kts

```
task<Sync>("libs") {
    from(configurations["runtime"])
    into("$buildDir/libs")
}
```

You can also perform the same function in your own tasks with the Project.sync(org.gradle.api.Action) method.

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 221. Archiving a directory as a ZIP

build.gradle

```
task packageDistribution(type: Zip) {
    archiveName = "my-distribution.zip"
    destinationDir = file("$buildDir/dist")

    from "$buildDir/toArchive"
}
```

build.gradle.kts

```
task<Zip>("packageDistribution") {
    archiveName = "my-distribution.zip"
    destinationDir = file("$buildDir/dist")

    from("$buildDir/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 'zipProject', hence the `myZip` task creates an archive named 'zipProject-1.0.zip':

Example 222. Creation of ZIP archive

build.gradle

```
plugins {
    id 'base'
}

version = 1.0

task myZip(type: Zip) {
    from 'somedir'

    doLast {
        println archiveName
        println relativePath(destinationDir)
        println relativePath(archivePath)
    }
}
```

build.gradle.kts

```
plugins {
    base
}

version = "1.0"

task<Zip>("myZip") {
    from("somedir")

    doLast {
        println(archiveName)
        println(relativePath(destinationDir))
        println(relativePath(archivePath))
    }
}
```

Output of `gradle -q myZip`

```
> gradle -q myZip
zipProject-1.0.zip
build/distributions
build/distributions/zipProject-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 `archiveName` and `destinationDir` 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.getArchiveName(): *[baseName]-[appendix]-[version]-[classifier].[extension]*. 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 *baseName*, project version for *version* and the archive type for *extension*. It does not provide values for the other properties.

This example — from the same project as the one above — configures just the `baseName` property, overriding the default value of the project name:

Example 223. Configuration of archive task - custom archive name

build.gradle

```
task myCustomZip(type: Zip) {
    baseName = 'customName'
    from 'somedir'

    doLast {
        println archiveName
    }
}
```

build.gradle.kts

```
task<Zip>("myCustomZip") {
    baseName = "customName"
    from("somedir")

    doLast {
        println(archiveName)
    }
}
```

Output of `gradle -q myCustomZip`

```
> gradle -q myCustomZip
customName-1.0.zip
```

You can also override the default `baseName` value for *all* the archive tasks in your build by using the

project property `archivesBaseName`, as demonstrated by the following example:

Example 224. Configuration of archive task - appendix & classifier

build.gradle

```
plugins {
    id 'base'
}

version = 1.0
archivesBaseName = "gradle"

task myZip(type: Zip) {
    from 'somedir'
}

task myOtherZip(type: Zip) {
    appendix = 'wrapper'
    classifier = 'src'
    from 'somedir'
}

task echoNames {
    doLast {
        println "Project name: ${project.name}"
        println myZip.archiveName
        println myOtherZip.archiveName
    }
}
```

build.gradle.kts

```
plugins {
    base
}

version = "1.0"
base.archivesBaseName = "gradle"

val myZip by tasks.creating(Zip::class) {
    from("somedir")
}

val myOtherZip by tasks.creating(Zip::class) {
    appendix = "wrapper"
    classifier = "src"
    from("somedir")
}

task("echoNames") {
    doLast {
        println("Project name: ${project.name}")
        println(myZip.archiveName)
        println(myOtherZip.archiveName)
    }
}
```

Output of `gradle -q echoNames`

```
> gradle -q echoNames
Project name: zipProject
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:

`archiveName` — `String`, *default:* `baseName-appendix-version-classifier.extension`

The complete file name of the generated archive. If any of the properties in the default value are empty, their '-' separator is dropped.

`archivePath` — `File`, *read-only, default:* `destinationDir/archiveName`

The absolute file path of the generated archive.

`destinationDir` — `File`, *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`.

`baseName` — `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.

`appendix` — `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.

`version` — `String`, *default:* `project.version`

The version portion of the archive file name, typically in the form of a normal project or product version.

`classifier` — `String`, *default:* `null`

The classifier portion of the archive file name. Often used to distinguish between archives that target different platforms.

`extension` — `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 archives

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 225. Activating reproducible archives

build.gradle

```
tasks.withType(AbstractArchiveTask) {
    preserveFileTimestamps = false
    reproducibleFileOrder = true
}
```

build.gradle.kts

```
tasks.withType<AbstractArchiveTask>() {
    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 Legacy Publishing.

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. [8: Any language element except for statement labels.] A Kotlin build script can contain any Kotlin language element. Gradle assumes that each build script is encoded using UTF-8.

The Project API

In the tutorial we used, for example, the `apply()` method. Where does this method come from? We said earlier that the build script defines a project in Gradle. For each project in the build, Gradle creates an object of type Project and associates this `Project` object with the build script. As the build script executes, it configures this `Project` object:

TIP

Getting help writing build scripts

Don't forget that your build script is simply Groovy or Kotlin code that drives the Gradle API. And the Project interface is your starting point for accessing everything in the Gradle API. So, if you're wondering what 'tags' are available in your build script, you can start with the documentation for the `Project` interface.

- Any method you call in your build script which *is not defined* in the build script, is delegated to the `Project` object.
- Any property you access in your build script, which *is not defined* in the build script, is delegated to the `Project` object.

Let's try this out and try to access the `name` property of the `Project` object.

Example 226. 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
projectApi
projectApi
```

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 6. 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 227. Using local variables

build.gradle

```
def dest = "dest"

task copy(type: Copy) {
    from "source"
    into dest
}
```

build.gradle.kts

```
val dest = "dest"

task<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 228. Using extra properties

build.gradle

```
plugins {
    id 'java'
}

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"
    }
}

task printProperties {
    doLast {
        println springVersion
        println emailNotification
        sourceSets.matching { it.purpose == "production" }.each { println it
.name }
    }
}
```

build.gradle.kts

```
plugins {
    java
}

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"
    }
}

task("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 229. Configuring arbitrary objects

build.gradle

```
import java.text.FieldPosition

task 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

task("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 230. Configuring arbitrary objects using a script

build.gradle

```
task 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 231. Groovy JDK methods

build.gradle

```
// Iterable gets an each() method
configurations.runtime.each { File f -> println f }
```

Have a look at http://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 232. 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 233. 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 234. 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 235. 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 236. Closure delegates

build.gradle

```
dependencies {
    assert delegate == project.dependencies
    testCompile('junit:junit:4.12')
    delegate.testCompile('junit:junit:4.12')
}
```

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.attributes.*
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.dsl.*
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.invocation.*
```

```
import org.gradle.api.java.archives.*
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.announce.*
import org.gradle.api.plugins.antlr.*
import org.gradle.api.plugins.buildcomparison.gradle.*
import org.gradle.api.plugins.osgi.*
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.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.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.authentication.*
import org.gradle.authentication.aws.*
import org.gradle.authentication.http.*
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.platform.*
import org.gradle.jvm.plugins.*
import org.gradle.jvm.tasks.*
import org.gradle.jvm.tasks.api.*
import org.gradle.jvm.test.*
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.coffeescript.*
import org.gradle.language.cpp.*
import org.gradle.language.cpp.plugins.*
import org.gradle.language.cpp.tasks.*
import org.gradle.language.java.*
import org.gradle.language.java.artifact.*
import org.gradle.language.java.plugins.*
import org.gradle.language.java.tasks.*
import org.gradle.language.javascript.*
import org.gradle.language.jvm.*
import org.gradle.language.jvm.plugins.*
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.routes.*
import org.gradle.language.scala.*
import org.gradle.language.scala.plugins.*
import org.gradle.language.scala.tasks.*
import org.gradle.language.scala.toolchain.*
import org.gradle.language.swift.*
import org.gradle.language.swift.plugins.*
import org.gradle.language.swift.tasks.*
import org.gradle.language.twirl.*
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.base.*
import org.gradle.platform.base.binary.*
import org.gradle.platform.base.component.*
import org.gradle.platform.base.plugins.*
import org.gradle.play.*
import org.gradle.play.distribution.*
import org.gradle.play.platform.*
import org.gradle.play.plugins.*
import org.gradle.play.plugins.ide.*
import org.gradle.play.tasks.*
import org.gradle.play.toolchain.*
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.javascript.base.*
import org.gradle.plugins.javascript.coffeescript.*
import org.gradle.plugins.javascript.envjs.*
import org.gradle.plugins.javascript.envjs.browser.*
import org.gradle.plugins.javascript.envjs.http.*
import org.gradle.plugins.javascript.envjs.http.simple.*
import org.gradle.plugins.javascript.jshint.*
import org.gradle.plugins.javascript.rhino.*
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.vcs.*
import org.gradle.vcs.git.*
import org.gradle.workers.*
```

Writing Custom Task Classes

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 is 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 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. 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 237. Defining a custom task

build.gradle

```
class GreetingTask extends DefaultTask {
}
```

build.gradle.kts

```
open 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 238. A hello world task

build.gradle

```
class GreetingTask extends DefaultTask {
    @TaskAction
    def greet() {
        println 'hello from GreetingTask'
    }
}

// Create a task using the task type
task hello(type: GreetingTask)
```

build.gradle.kts

```
open class GreetingTask : DefaultTask() {
    @TaskAction
    fun greet() {
        println("hello from GreetingTask")
    }
}

// Create a task using the task type
task<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 simply POGOs, 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 239. A customizable hello world task

build.gradle

```
class GreetingTask extends DefaultTask {
    String greeting = 'hello from GreetingTask'

    @TaskAction
    def greet() {
        println greeting
    }
}

// Use the default greeting
task hello(type: GreetingTask)

// Customize the greeting
task greeting(type: GreetingTask) {
    greeting = 'greetings from GreetingTask'
}
```

build.gradle.kts

```
open class GreetingTask : DefaultTask() {
    var greeting = "hello from GreetingTask"

    @TaskAction
    fun greet() {
        println(greeting)
    }
}

// Use the default greeting
task<GreetingTask>("hello")

// Customize the greeting
task<GreetingTask>("greeting") {
    greeting = "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 240. A build for a custom task

build.gradle

```
plugins {
    id 'groovy'
}

dependencies {
    implementation gradleApi()
    implementation localGroovy()
}
```

build.gradle.kts

```
plugins {
    groovy
}

dependencies {
    implementation(gradleApi())
    implementation(localGroovy())
}
```

NOTE The code for this example can be found at `samples/customPlugin` in the '-all' distribution of Gradle.

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

class GreetingTask extends DefaultTask {
    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 241. Using a custom task in another project

build.gradle

```
buildscript {
    repositories {
        maven {
            url = uri(repoLocation)
        }
    }
    dependencies {
        classpath 'org.gradle:customPlugin:1.0-SNAPSHOT'
    }
}

task greeting(type: org.gradle.GreetingTask) {
    greeting = 'howdy!'
}
```

build.gradle.kts

```
buildscript {
    repositories {
        maven {
            url = uri(repoLocation)
        }
    }
    dependencies {
        classpath("org.gradle:customPlugin:1.0-SNAPSHOT")
    }
}

task<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
    public 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 optimise your build so that only out-of-date inputs are processed, you can do so with an *incremental task*.

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 contains a single IncrementalTaskInputs parameter, which indicates to Gradle that the action will process the changed inputs only.

The incremental task action may supply an IncrementalTaskInputs.outOfDate(org.gradle.api.Action) action for processing any input file that is out-of-date, and a IncrementalTaskInputs.removed(org.gradle.api.Action) action that executes for any input file that has been removed since the previous execution.

Example 242. Defining an incremental task action

build.gradle

```
class IncrementalReverseTask extends DefaultTask {
    @InputDirectory
    def File inputDir

    @OutputDirectory
    def File outputDir

    @Input
    def inputProperty

    @TaskAction
    void execute(IncrementalTaskInputs inputs) {
        println inputs.incremental ? 'CHANGED inputs considered out of date'
                                   : 'ALL inputs considered out of date'
        if (!inputs.incremental)
            project.delete(outputDir.listFiles())

        inputs.outOfDate { change ->
            if (change.file.directory) return

            println "out of date: ${change.file.name}"
            def targetFile = new File(outputDir, change.file.name)
            targetFile.text = change.file.text.reverse()
        }

        inputs.removed { change ->
            if (change.file.directory) return

            println "removed: ${change.file.name}"
            def targetFile = new File(outputDir, change.file.name)
            targetFile.delete()
        }
    }
}
```

build.gradle.kts

```
open class IncrementalReverseTask : DefaultTask() {
    @InputDirectory
    lateinit var inputDir: File

    @OutputDirectory
    lateinit var outputDir: File

    @Input
    lateinit var inputProperty: String

    @TaskAction
    fun execute(inputs: IncrementalTaskInputs) {
        println(
            if (inputs.isIncremental) "CHANGED inputs considered out of date"
            else "ALL inputs considered out of date"
        )
        if (!inputs.isIncremental) {
            project.delete(outputDir.listFiles())
        }

        inputs.outOfDate {
            if (file.isDirectory) return@outOfDate

            println("out of date: ${file.name}")
            val targetFile = File(outputDir, file.name)
            targetFile.writeText(file.readText().reversed())
        }

        inputs.removed {
            if (file.isDirectory) return@removed

            println("removed: ${file.name}")
            val targetFile = File(outputDir, file.name)
            targetFile.delete()
        }
    }
}
```

NOTE The code for this example can be found at `samples/userguide/tasks/incrementalTask` in the '-all' distribution of Gradle.

If for some reason the task is not run incremental, e.g. by running with `--rerun-tasks`, only the outOfDate action is executed, even if there were deleted input files. You should consider handling this case at the beginning, as is done in the example above.

For a simple transformer task like this, the task action simply needs to generate output files for any out-of-date inputs, and delete output files for any removed inputs.

A task may only contain a single incremental task action.

Which inputs are considered out of date?

When Gradle has history of a previous task execution, and the only changes to the task execution context since that execution are to input files, then Gradle is able to determine which input files need to be reprocessed by the task. In this case, the IncrementalTaskInputs.outOfDate(org.gradle.api.Action) action will be executed for any input file that was *added* or *modified*, and the IncrementalTaskInputs.removed(org.gradle.api.Action) action will be executed for any *removed* input file.

However, there are many cases where Gradle is unable to determine which input files need to be reprocessed. 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` criteria added to the task returns `false`.
- An input property has changed since the previous execution.
- One or more output files have changed since the previous execution.

In any of these cases, Gradle will consider all of the input files to be `outOfDate`. The IncrementalTaskInputs.outOfDate(org.gradle.api.Action) action will be executed for every input file, and the IncrementalTaskInputs.removed(org.gradle.api.Action) action will not be executed at all.

You can check if Gradle was able to determine the incremental changes to input files with IncrementalTaskInputs.isIncremental().

An incremental task in action

Given the incremental task implementation above, we can explore the various change scenarios by example. Note that the various mutation tasks ('updateInputs', 'removeInput', etc) are only present for demonstration purposes: these would not normally be part of your build script.

First, consider the `IncrementalReverseTask` executed against a set of inputs for the first time. In this case, all inputs will be considered “out of date”:

Example 243. Running the incremental task for the first time

build.gradle

```
task incrementalReverse(type: IncrementalReverseTask) {
    inputDir = file('inputs')
    outputDir = file("$buildDir/outputs")
    inputProperty = project.properties['taskInputProperty'] ?: 'original'
}
```

build.gradle.kts

```
task<IncrementalReverseTask>("incrementalReverse") {
    inputDir = file("inputs")
    outputDir = file("$buildDir/outputs")
    inputProperty = project.properties["taskInputProperty"] as String? ?:
"original"
}
```

Build layout

```
.
├─── build.gradle
└─── inputs
    ├─── 1.txt
    ├─── 2.txt
    └─── 3.txt
```

Output of `gradle -q incrementalReverse`

```
> gradle -q incrementalReverse
ALL inputs considered out of date
out of date: 1.txt
out of date: 2.txt
out of date: 3.txt
```

Naturally when the task is executed again with no changes, then the entire task is up to date and no files are reported to the task action:

Example: Running the incremental task with unchanged inputs

Output of `gradle -q incrementalReverse`

```
> gradle -q incrementalReverse
```

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 reported to IncrementalTaskInputs.outOfDate(org.gradle.api.Action):

Example 244. Running the incremental task with updated input files

build.gradle

```
task updateInputs() {
    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

```
task("updateInputs") {
    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
CHANGED inputs considered out of date
out of date: 1.txt
out of date: 4.txt
```

When an existing input file is removed, then re-executing the task results in that file being reported to IncrementalTaskInputs.removed(org.gradle.api.Action):

Example 245. Running the incremental task with an input file removed

build.gradle

```
task removeInput() {
    doLast {
        file('inputs/3.txt').delete()
    }
}
```

build.gradle.kts

```
task("removeInput") {
    doLast {
        file("inputs/3.txt").delete()
    }
}
```

Output of `gradle -q removeInput incrementalReverse`

```
> gradle -q removeInput incrementalReverse
CHANGED inputs considered out of date
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, *all* input files are reported to the IncrementalTaskInputs.outOfDate(org.gradle.api.Action) action, and no input files are reported to the IncrementalTaskInputs.removed(org.gradle.api.Action) action:

Example 246. Running the incremental task with an output file removed

build.gradle

```
task removeOutput() {
    doLast {
        file("$buildDir/outputs/1.txt").delete()
    }
}
```

build.gradle.kts

```
task("removeOutput") {
    doLast {
        file("$buildDir/outputs/1.txt").delete()
    }
}
```

Output of `gradle -q removeOutput incrementalReverse`

```
> gradle -q removeOutput incrementalReverse
ALL inputs considered out of date
out of date: 1.txt
out of date: 2.txt
out of date: 3.txt
```

When a task input property is modified, Gradle is unable to determine how this property impacted the task outputs, so all input files are assumed to be out of date. So similar to the changed output file example, *all* input files are reported to the IncrementalTaskInputs.outOfDate(org.gradle.api.Action) action, and no input files are reported to the IncrementalTaskInputs.removed(org.gradle.api.Action) action:

Example: Running the incremental task with an input property changed

Output of `gradle -q -PtaskInputProperty=changed incrementalReverse`

```
> gradle -q -PtaskInputProperty=changed incrementalReverse
ALL inputs considered out of date
out of date: 1.txt
out of date: 2.txt
out of date: 3.txt
```

Storing incremental state for cached tasks

Using Gradle's `IncrementalTaskInputs` is not the only way to create tasks that only works 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 to confuse the tool during the next execution. Gradle can ensure such stale files are removed if they are declared via `task.localState.register()` or a property is marked with the `@LocalState` annotation.

Declaring and Using Command Line Options

NOTE	The API for exposing command line options is an incubating feature.

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.

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 247. Using a command line option

build.gradle

```
task verifyUrl(type: UrlVerify)
```

build.gradle.kts

```
task<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`
: Describes an option with the value `true` or `false`. Passing the option on the command line does not require assigning a value. For example `--enabled` equates to `true`. The absence of the option uses the default values assign to the property; that is `false` for `boolean` and `null` for the complex data type.

`String`
: Describes an option with an arbitrary String value. Passing the option on the command line requires a key-value pair of option and value separated by an equals sign e.g. `--containerId=2x94held`.

`enum`
: Describes an option as enum. The enum has to be passed on the command line as key-value pair similar to the String type e.g. `--log-level=DEBUG`. The provided 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 distinct declarations e.g. `--imageId=123 --imageId=456`. Other notations like 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 reponse 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.

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.

The Worker API

NOTE	The Worker API is an incubating feature.

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

In order to submit work to the Worker API, two things must be provided: an implementation of the unit of work, and a configuration for the unit of work. The implementation is simply a class that extends `java.lang.Runnable`. This class should have a constructor that is annotated with `javax.inject.Inject` and accepts parameters that configure the class for a single unit of work. When a unit of work is submitted to the WorkerExecutor, an instance of this class will be created and the parameters configured for the unit of work will be passed to the constructor.

Example 248. Creating a unit of work implementation

build.gradle

```
import org.gradle.workers.WorkerExecutor

import javax.inject.Inject

// The implementation of a single unit of work
class ReverseFile implements Runnable {
    File fileToReverse
    File destinationFile

    @Inject
    public ReverseFile(File fileToReverse, File destinationFile) {
        this.fileToReverse = fileToReverse
        this.destinationFile = destinationFile
    }

    @Override
    public void run() {
        destinationFile.text = fileToReverse.text.reverse()
    }
}
```

build.gradle.kts

```
import org.gradle.workers.WorkerExecutor

import javax.inject.Inject

// The implementation of a single unit of work
open class ReverseFile @Inject constructor(val fileToReverse: File, val
destinationFile: File) : Runnable {

    override fun run() {
        destinationFile.writeText(fileToReverse.readText().reversed())
    }
}
```

The configuration of the worker is represented by a WorkerConfiguration and is set by configuring an instance of this object at the time of submission. However, in order to submit the unit of work, it is necessary to first acquire the WorkerExecutor. To do this, a constructor should be provided that is annotated with `javax.inject.Inject` and accepts a WorkerExecutor parameter. Gradle will inject the instance of WorkerExecutor at runtime when the task is created.

Example 249. Submitting a unit of work for execution

build.gradle

```
class ReverseFiles extends SourceTask {
    final WorkerExecutor workerExecutor

    @OutputDirectory
    File outputDir

    // The WorkerExecutor will be injected by Gradle at runtime
    @Inject
    public ReverseFiles(WorkerExecutor workerExecutor) {
        this.workerExecutor = workerExecutor
    }

    @TaskAction
    void reverseFiles() {
        // Create and submit a unit of work for each file
        source.each { file ->
            workerExecutor.submit(ReverseFile.class) { WorkerConfiguration
config ->
                // Use the minimum level of isolation
                config.isolationMode = IsolationMode.NONE

                // Constructor parameters for the unit of work implementation
                config.params file, project.file("$outputDir/${file.name}")
            }
        }
    }
}
```

build.gradle.kts

```
// The WorkerExecutor will be injected by Gradle at runtime
open class ReverseFiles @Inject constructor(val workerExecutor:
WorkerExecutor) : SourceTask() {
    @OutputDirectory
    lateinit var outputDir: File

    @TaskAction
    fun reverseFiles() {
        // Create and submit a unit of work for each file
        source.forEach { file ->
            workerExecutor.submit(ReverseFile::class) {
                // Use the minimum level of isolation
                isolationMode = IsolationMode.NONE

                // Constructor parameters for the unit of work implementation
                params(file, project.file("$outputDir/${file.name}"))
            }
        }
    }
}
```

Note that one element of the WorkerConfiguration is the `params` property. These are the parameters passed to the constructor of the unit of work implementation for each item of work submitted. Any parameters provided to the unit of work *must* be `java.io.Serializable`.

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 WorkerExecutor.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 WorkerExecutor.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 WorkerExecutor.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 250. Waiting for asynchronous work to complete

build.gradle

```
        // Create and submit a unit of work for each file
        source.each { file ->
            workerExecutor.submit(ReverseFile.class) { config ->
                config.isolationMode = IsolationMode.NONE
                // Constructor parameters for the unit of work implementation
                config.params file, project.file("${outputDir}/${file.name}")
            }
        }

        // Wait for all asynchronous work to complete before continuing
        workerExecutor.await()
        logger.lifecycle("Created ${outputDir.listFiles().size()} reversed
files in ${project.relativePath(outputDir)}")
```

build.gradle.kts

```
        // Create and submit a unit of work for each file
        source.forEach { file ->
            workerExecutor.submit(ReverseFile::class) {
                isolationMode = IsolationMode.NONE
                // Constructor parameters for the unit of work implementation
                params(file, project.file("$outputDir/${file.name}"))
            }
        }

        // Wait for all asynchronous work to complete before continuing
        workerExecutor.await()
        logger.lifecycle("Created ${outputDir.listFiles().size} reversed
files in ${project.relativePath(outputDir)}")
```

Isolation Modes

Gradle provides three isolation modes that can be configured on a unit of work and are specified using the IsolationMode enum:

IsolationMode.NONE

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.

IsolationMode.CLASSLOADER

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 WorkerConfiguration.classpath(java.lang.Iterable).

IsolationMode.PROCESS

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 WorkerConfiguration.classpath(java.lang.Iterable). 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 WorkerConfiguration.forkOptions(org.gradle.api.Action).

Worker Daemons

When using `IsolationMode.PROCESS`, gradle will start a long-lived *Worker Daemon* process that can be reused for future work items.

Example 251. Submitting an item of work to run in a worker daemon

build.gradle

```
            workerExecutor.submit(ReverseFile.class) { WorkerConfiguration
config ->
                // Run this work in an isolated process
                config.isolationMode = IsolationMode.PROCESS

                // Configure the options for the forked process
                config.forkOptions { JavaForkOptions options ->
                    options.maxHeapSize = "512m"
                    options.systemProperty "org.gradle.sample.showFileSize",
"true"
                }

                // Constructor parameters for the unit of work implementation
                config.params file, project.file("${outputDir}/${file.name}")
            }
```

build.gradle.kts

```
            workerExecutor.submit(ReverseFile::class) {
                // Run this work in an isolated process
                isolationMode = IsolationMode.PROCESS

                // Configure the options for the forked process
                forkOptions {
                    maxHeapSize = "512m"
                    systemProperty("org.gradle.sample.showFileSize", "true")
                }

                // Constructor parameters for the unit of work implementation
                params(file, project.file("$outputDir/${file.name}"))
            }
```

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 WorkerConfiguration.forkOptions(org.gradle.api.Action).

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 if it has more classpath entries in addition to those requested.

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.

Re-using logic between task classes

There are different ways to re-use logic between task classes. The easiest case is when you can extract the logic you want to share in a separate method or class and then use the extracted piece of code in your tasks. For example, the Copy task re-uses the logic of the Project.copy(org.gradle.api.Action) method. Another option is to add a task dependency on the task which outputs you want to re-use. Other options include using task rules or the worker API.

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 252. 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.

`uploadConfiguration` — *task rule*

Does the same as `buildConfiguration`, but also uploads all the artifacts attached to the given 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 for artifacts:

`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. This results in an `uploadArchives` task for publishing artifacts attached to the `archives` configuration.

Note that the `assemble` task generates all artifacts that are attached to the `archives` configuration.

Conventions

The Base Plugin only adds conventions related to the creation of archives, such as ZIPs, TARs and JARs. Specifically, it provides the following project properties that you can set:

`archivesBaseName` — *default:* `$project.name`

Provides the default AbstractArchiveTask.getBaseName() for archive tasks.

`distsDirName` — *default: distributions*

Default name of the directory in which distribution archives, i.e. non-JARs, are created.

`libsDirName` — *default: 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:

`destinationDir`

Defaults to `$buildDir/$distsDirName` for non-JAR archives and `$buildDir/$libsDirName` for JARs and derivatives of JAR, such as WARs.

`version`

Defaults to `$project.version` or 'unspecified' if the project has no version.

`baseName`

Defaults to `$archivesBaseName`.

Dependency Management

Introduction to Dependency Management

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 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.

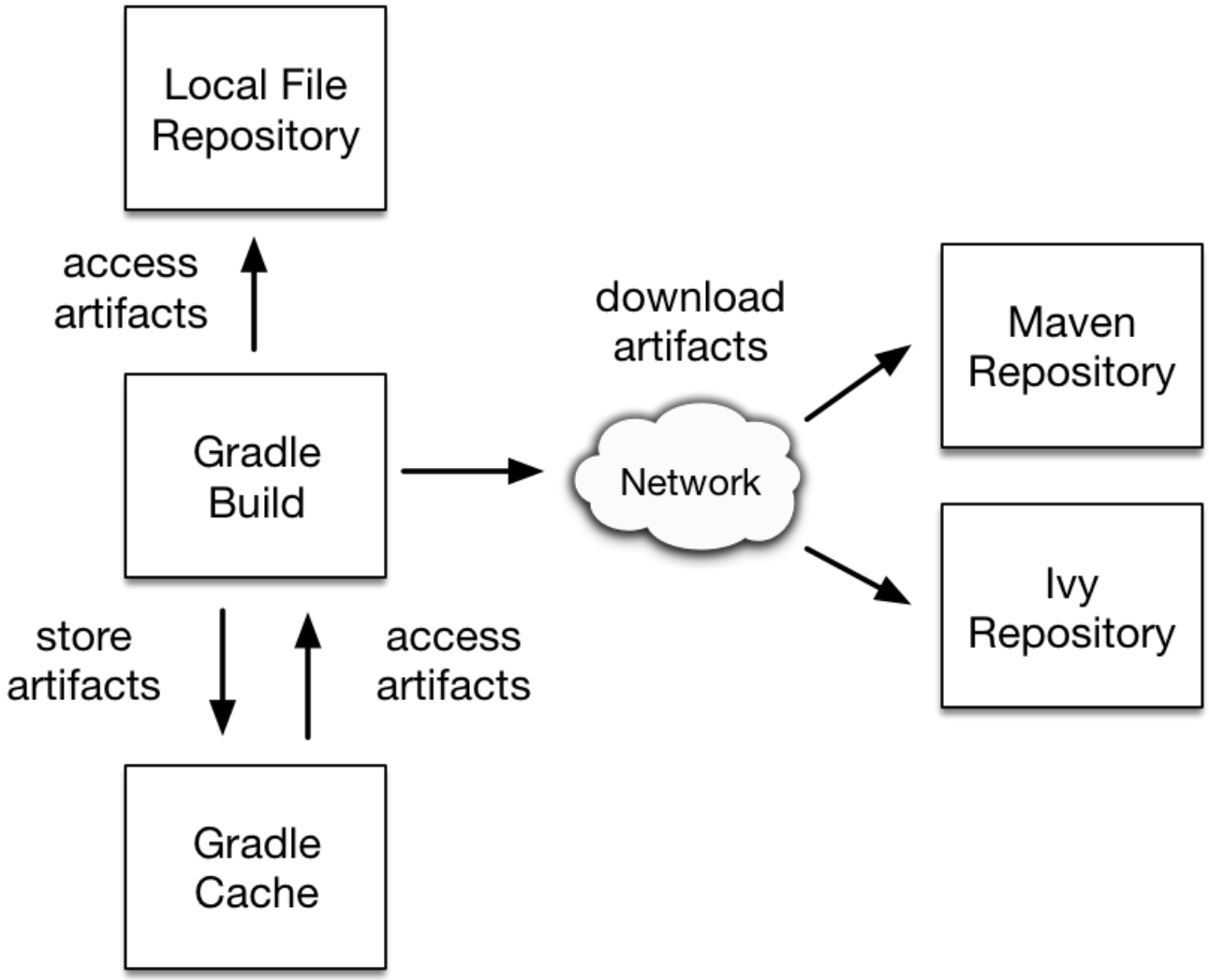


Figure 13. Dependency management big picture

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 dependency resolution works.

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 Inspecting Dependencies.

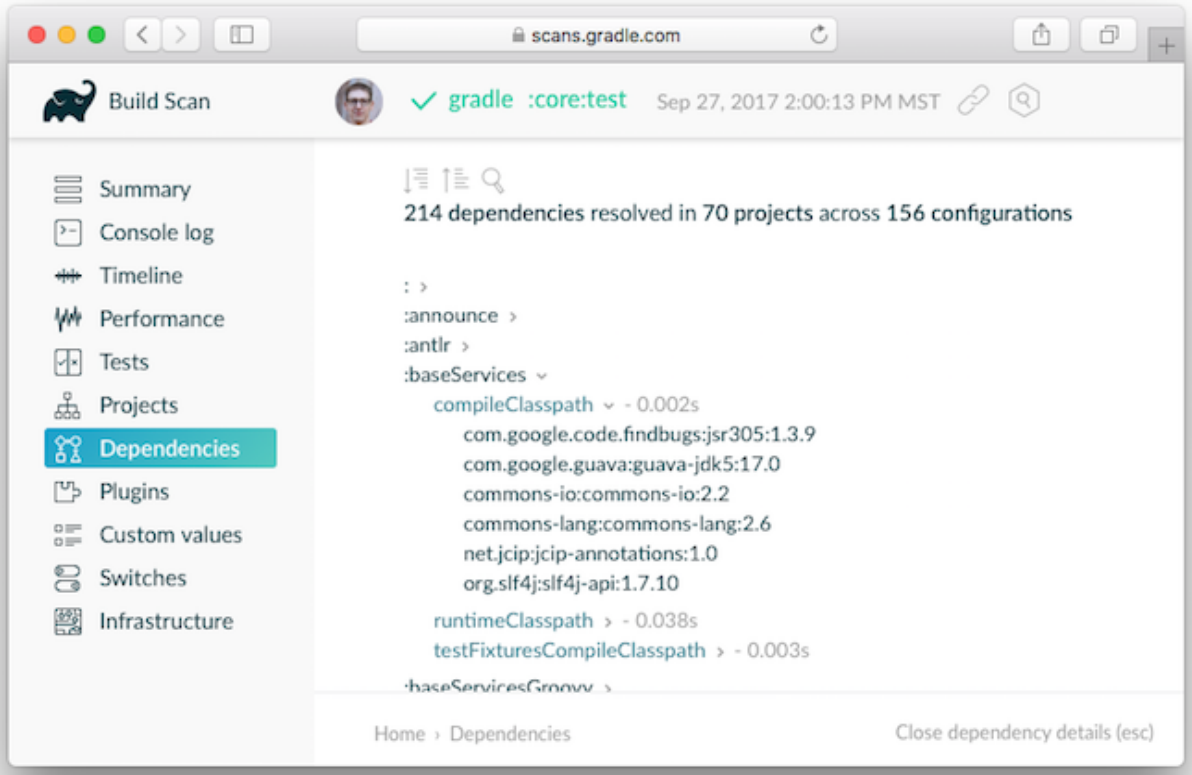


Figure 14. Build scan dependencies report

How dependency resolution works

Gradle takes your dependency declarations and repository definitions and attempts to download all of your dependencies by a process called *dependency resolution*. Below is a brief outline of how this process works.

- Given a required dependency, 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.

    - If the dependency is declared as a dynamic version (like `1.+`), Gradle will resolve this to the highest available concrete version (like `1.2`) in the repository. For Maven repositories, this is done using the `maven-metadata.xml` file, while for Ivy repositories this is done by directory listing.
    - 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.
- Once each repository has been inspected for the module, Gradle will choose the 'best' one to use. This is done using the following criteria:
    - For a dynamic version, a 'higher' concrete version is preferred over a 'lower' version.
    - Modules declared by a module metadata file (`.module`, `.pom` or `ivy.xml` file) are preferred over modules that have an artifact file only.
    - Modules from earlier repositories are preferred over modules in later repositories.
    - When the dependency is declared by a concrete version and a module metadata file is found in a repository, there is no need to continue searching later repositories and the remainder of the process is short-circuited.
- All of the artifacts for the module are then requested from the *same repository* that was chosen in the process above.

The dependency resolution process is highly customizable to meet enterprise requirements. For more information, see the chapter on customizing dependency resolution.

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.

Configuration

A configuration is a named set of dependencies grouped together for a specific goal: For example the `implementation` configuration represents the set of dependencies required to compile a project. Configurations provide access to the underlying, resolved modules and their artifacts. For more information, see Managing Dependency 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 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 Dependency Constraints.

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 can provide metadata. Metadata is the data that describes the module in more detail e.g. the coordinates for locating it in a repository, information about the project or required transitive dependencies. In Maven the metadata file is called `.pom`, in Ivy it is called `ivy.xml`.

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.

Repository

A repository hosts a set of modules, each of which may provide one or many releases 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. Resolution rules are defined as part of the build logic. For more information, see Customizing Dependency Resolution Behavior.

Transitive dependency

A module 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. However, the behavior is highly customizable. For more information, see Managing Transitive Dependencies.

Dependency Types

Module dependencies

Module dependencies are the most common dependencies. They refer to a module in a repository.

Example 253. Module dependencies

build.gradle

```
dependencies {
    runtime group: 'org.springframework', name: 'spring-core', version: '2.5'
    runtime 'org.springframework:spring-core:2.5',
            'org.springframework:spring-aop:2.5'
    runtime(
        [group: 'org.springframework', name: 'spring-core', version: '2.5'],
        [group: 'org.springframework', name: 'spring-aop', version: '2.5']
    )
    runtime('org.hibernate:hibernate:3.0.5') {
        transitive = true
    }
    runtime group: 'org.hibernate', name: 'hibernate', version: '3.0.5',
transitive: true
    runtime(group: 'org.hibernate', name: 'hibernate', version: '3.0.5') {
        transitive = true
    }
}
```

build.gradle.kts

```
dependencies {
    runtime(group = "org.springframework", name = "spring-core", version =
"2.5")
    runtime("org.springframework:spring-aop:2.5")
    runtime("org.hibernate:hibernate:3.0.5") {
        isTransitive = true
    }
    runtime(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, Gradle may look, depending on the metadata sources definitions, for an artifact file called `hibernate-3.0.5.jar` directly. 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

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 254. File dependencies

build.gradle

```
dependencies {
    runtime files('libs/a.jar', 'libs/b.jar')
    runtime fileTree('libs') { include '*.jar' }
}
```

build.gradle.kts

```
dependencies {
    runtime(files("libs/a.jar", "libs/b.jar"))
    runtime(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 with the same build.

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 255. Generated file dependencies

build.gradle

```
dependencies {
    compile files("$buildDir/classes") {
        builtBy 'compile'
    }
}

task compile {
    doLast {
        println 'compiling classes'
    }
}

task list(dependsOn: configurations.compile) {
    doLast {
        println "classpath = ${configurations.compile.collect { File file ->
file.name }}"
    }
}
```

build.gradle.kts

```
dependencies {
    compile(files("$buildDir/classes") {
        builtBy("compile")
    })
}

task("compile") {
    doLast {
        println("compiling classes")
    }
}

task("list") {
    dependsOn(configurations["compile"])
    doLast {
        println("classpath = ${configurations["compile"].map { file: File ->
file.name }}")
    }
}
```

```
$ gradle -q list
compiling classes
classpath = [classes]
```

Project dependencies

Gradle distinguishes between external dependencies and dependencies on projects which are part of the same multi-project build. For the latter you can declare *project dependencies*.

Example 256. Project dependencies

build.gradle

```
dependencies {
    implementation project(':shared')
}
```

build.gradle.kts

```
dependencies {
    implementation(project(":shared"))
}
```

For more information see the API documentation for ProjectDependency.

Multi-project builds are discussed in this chapter.

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 257. 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 258. 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 259. Gradle's Groovy dependencies

build.gradle

```
dependencies {
    implementation localGroovy()
}
```

build.gradle.kts

```
dependencies {
    implementation(localGroovy())
}
```

Repository Types

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 260. 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. Note that 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. However, 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. 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.

Maven Central repository

Maven Central is a popular repository hosting open source libraries for consumption by Java projects.

To declare the central Maven repository for your build add this to your script:

Example 261. Adding central Maven repository

build.gradle

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

build.gradle.kts

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

JCenter Maven repository

Bintray's JCenter is an up-to-date collection of all popular Maven OSS artifacts, including artifacts published directly to Bintray.

To declare the JCenter Maven repository add this to your build script:

Example 262. Adding Bintray's JCenter Maven repository

build.gradle

```
repositories {
    jcenter()
}
```

build.gradle.kts

```
repositories {
    jcenter()
}
```

Google Maven repository

The Google repository hosts Android-specific artifacts including the Android SDK. For usage examples, see the relevant documentation.

To declare the Google Maven repository add this to your build script:

Example 263. Adding Google Maven repository

build.gradle

```
repositories {
    google()
}
```

build.gradle.kts

```
repositories {
    google()
}
```

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.

To declare the local Maven cache as a repository add this to your build script:

Example 264. 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`.

Custom 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 265. 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")
    }
}
```

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 266. 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 first URL for the POM and the JAR. If the JAR can't be found there, the artifact URLs are used to look for JARs.

See Configuring HTTP authentication schemes for authentication options.

Custom 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 267. 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 268. 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'`, `'ivy'` and `'pattern'`. See IvyArtifactRepository.layout(java.lang.String, groovy.lang.Closure) 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 269. 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 270. 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 271. 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 password-protected Ivy repositories

You can specify credentials for Ivy repositories secured by basic authentication.

Example 272. 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"
        }
    }
}
```

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 273. 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 7. Repository transport protocols

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 5.0. Before 5.0, `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 { mavenPom(); artifact() }`. In a similar way, you can opt into the new behavior in older Gradle verisions using `mavenCentral { mavenPom() }`

Supported repository transport protocols

Maven and Ivy repositories support the use of various transport protocols. At the moment the following protocols are supported:

Table 8. Repository transport protocols

Type	Credential types
`file`	none
`http`	username/password
`https`	username/password
`sftp`	username/password
`s3`	access key/secret key/session token or Environment variables
`gcs`	default application credentials sourced from well known files, Environment variables etc.

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 a HTTP-based Maven and Ivy repository:

Example 274. 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 275. 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"
        }
    }
}
```

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 276. Declaring a 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 277. Declaring a 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")
        }
    }
}
```

When using a Google Cloud Storage backed repository default application credentials will be used with no further configuration required:

Example 278. 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")
    }
}
```

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`

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 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 `/`.

Google Cloud Storage 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)

Configuring HTTP authentication schemes

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 api:org.gradle.api.credentials.PasswordCredentials[].

Example 279. 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"
        }
    }
}
```

NOTE The code for this example can be found at `samples/userguide/artifacts/defineRepository` in the '-all' distribution of Gradle.

The following example show how to configure a repository to use only api:org.gradle.authentication.http.DigestAuthentication[]:

Example 280. 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 a 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 281. 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 api:org.gradle.api.credentials.HttpHeaderCredentials[]

with api:org.gradle.authentication.http.HttpHeaderAuthentication[].

Example 282. 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")
        }
    }
}
```

NOTE The code for this example can be found at `samples/userguide/artifacts/defineRepository` in the '-all' distribution of Gradle.

Declaring Dependencies

Gradle builds can declare dependencies on modules hosted in repositories, files and other Gradle projects. You can find examples for common scenarios in this section. For more information, see the full reference on all types of dependencies.

Every dependency needs to be assigned to a configuration when declared in a build script. For

more information on the purpose and syntax of configurations, see Managing Dependency Configurations.

Declaring a dependency to a module

Modern software projects rarely build code in isolation. Projects reference modules for the purpose of reusing existing and proven functionality. Upon resolution, selected versions of modules are downloaded from dedicated repositories and stored in the dependency cache to avoid unnecessary network traffic.

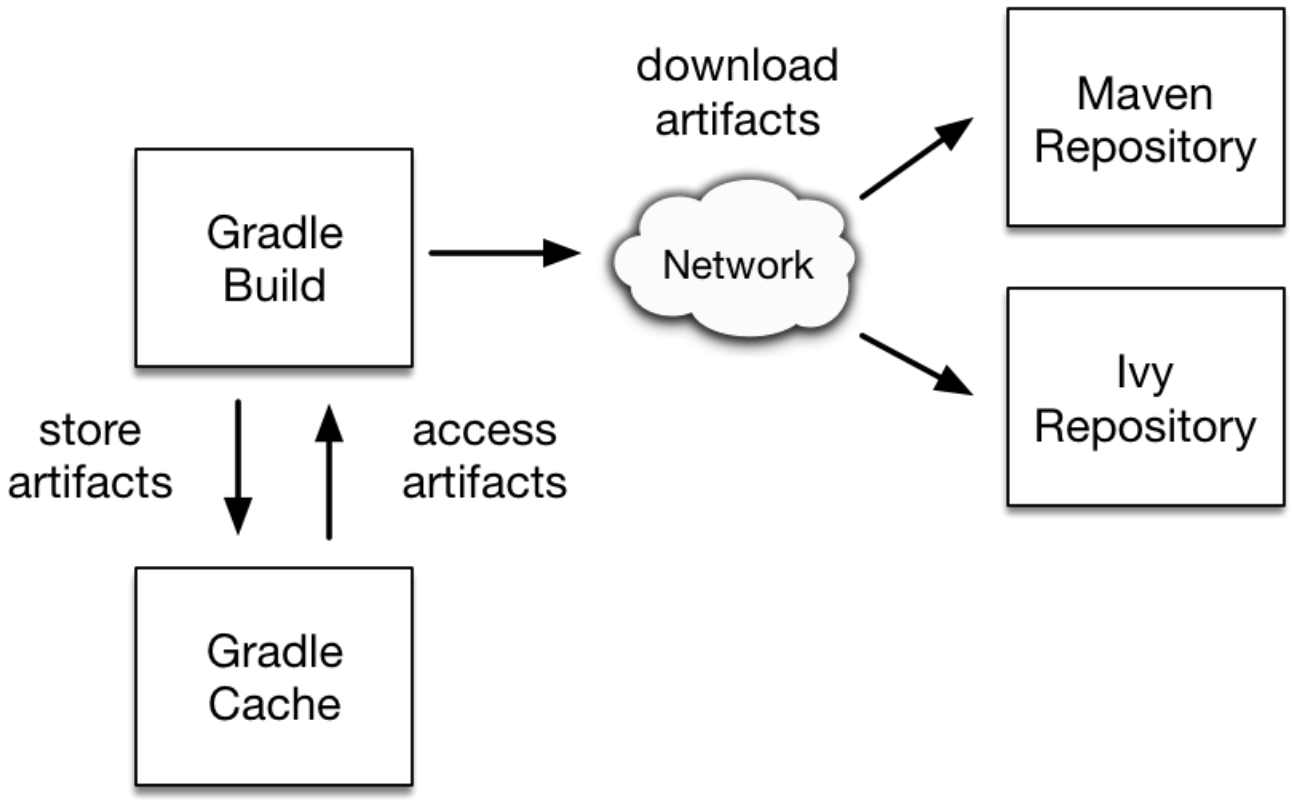


Figure 15. Resolving dependencies from remote repositories

Declaring a concrete version of a dependency

A typical example for such a library in a Java project is the Spring framework. The following code snippet declares a compile-time dependency on the Spring web module by its coordinates: `org.springframework:spring-web:5.0.2.RELEASE`. Gradle resolves the module including its transitive dependencies from the Maven Central repository and uses it to compile Java source code. The version attribute of the dependency coordinates points to a *concrete version* indicating that the underlying artifacts do not change over time. The use of concrete versions ensure reproducibility for the aspect of dependency resolution.

Example 283. Declaring a dependency with a concrete version

build.gradle

```
plugins {
    id 'java-library'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework:spring-web:5.0.2.RELEASE'
}
```

build.gradle.kts

```
plugins {
    `java-library`
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.springframework:spring-web:5.0.2.RELEASE")
}
```

A Gradle project can define other types of repositories hosting modules. You can learn more about the syntax and API in the section on declaring repositories. Refer to the chapter on the Java Plugin for a deep dive on declaring dependencies for a Java project. The resolution behavior for dependencies is highly customizable.

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 284. 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 {
        add("implementation", "org.springframework:spring-web:5.0.2.RELEASE")
    }
}
```

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 285. 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.

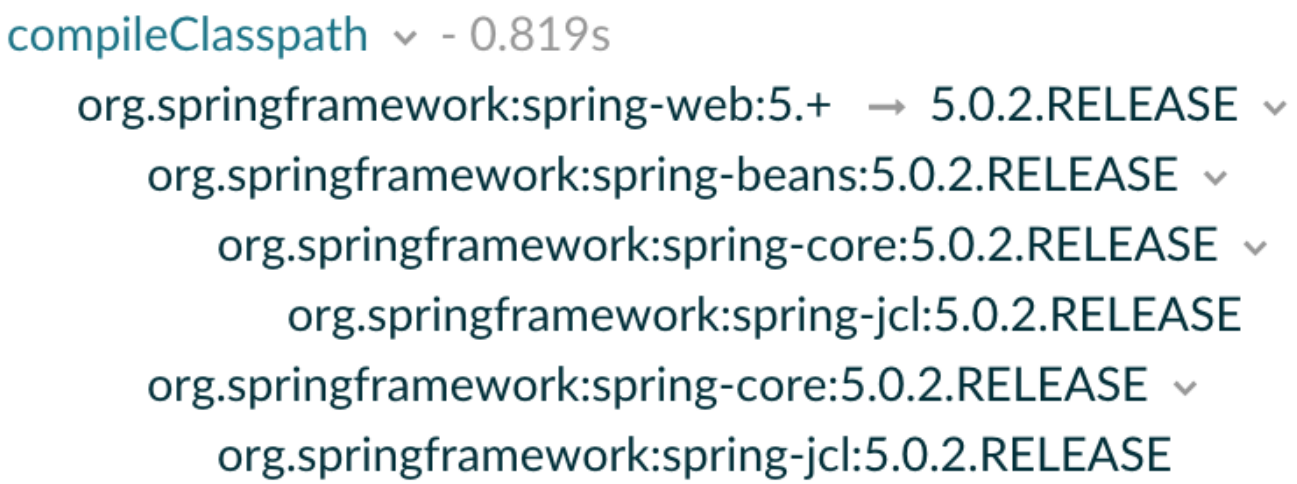


Figure 16. Dynamic dependencies in build scan

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 286. 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`.

Declaring a file dependency

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.

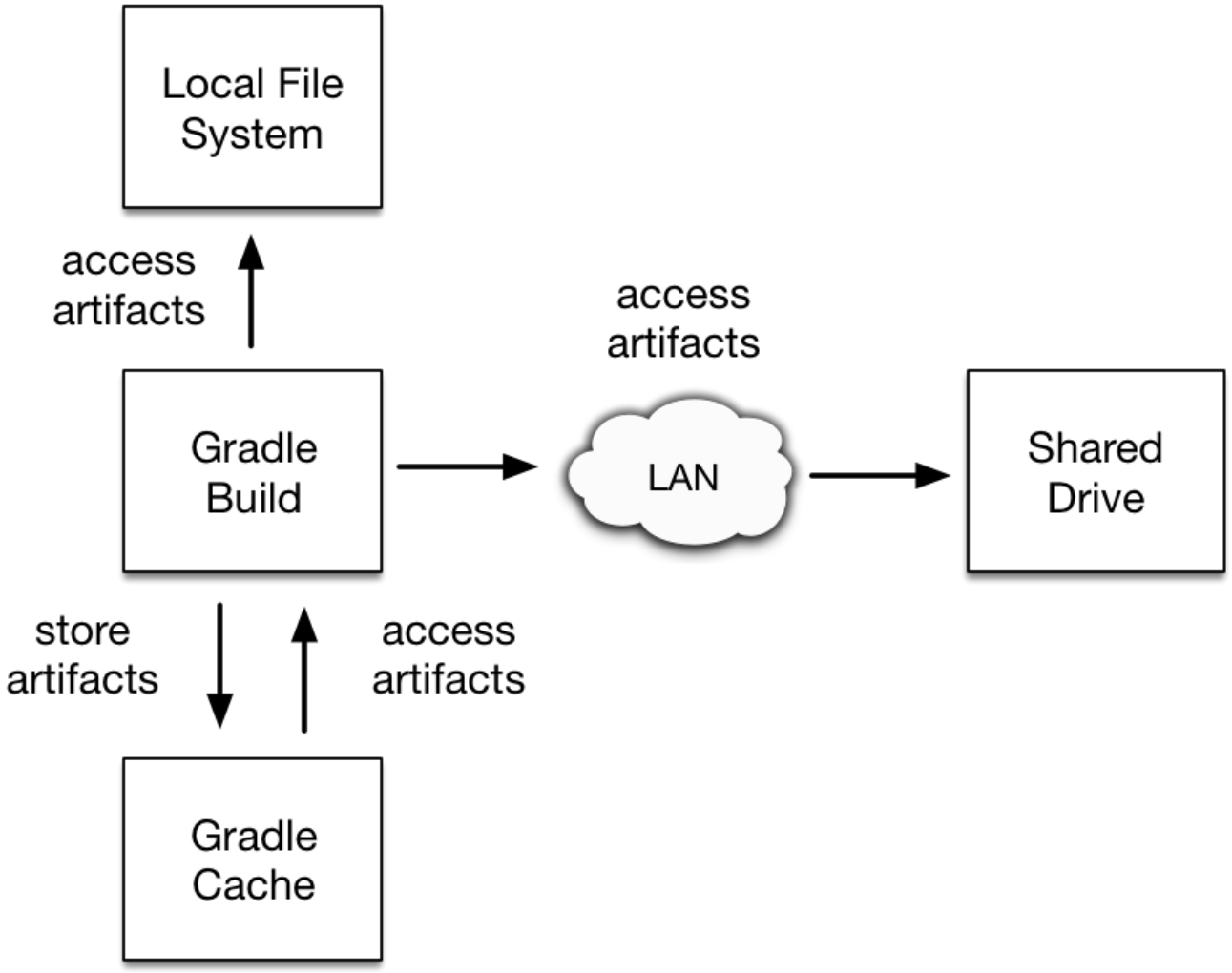


Figure 17. Resolving file dependencies from the local file system and a shared drive

The following example resolves file dependencies from the directories `ant`, `libs` and `tools`.

Example 287. 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...), ProjectLayout.configurableFiles(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.

Declaring a project dependency

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.

Gradle can model dependencies between modules. Those dependencies are called *project dependencies* because each module is represented by a Gradle project. 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.

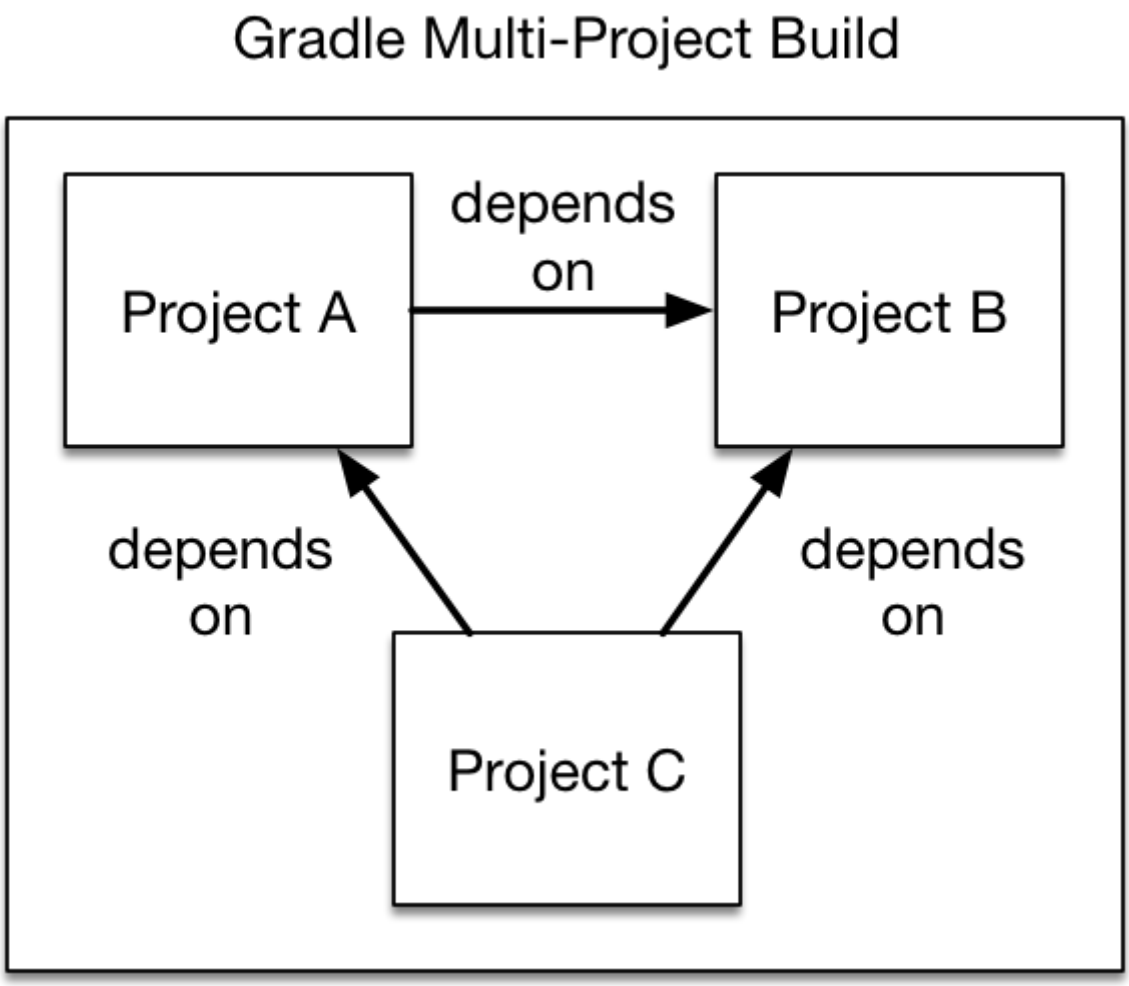


Figure 18. Dependencies between projects

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 288. Declaring project dependencies

build.gradle

```
project(':web-service') {
    dependencies {
        implementation project(':utils')
        implementation project(':api')
    }
}
```

build.gradle.kts

```
project(":web-service") {
    dependencies {
        "implementation"(project(":utils"))
        "implementation"(project(":api"))
    }
}
```

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 289. 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]'
        }
    }
}

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]")
        }
    }
}

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 290. 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]'
        }
    }
}

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]")
        }
    }
}

configurations {
    create("js")
}

dependencies {
    "js"("jquery:jquery:3.2.1:min@js")
}
```

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, Bintray JCenter and the Google Android repository. Gradle provides built-in shortcut methods for the most widely-used repositories.

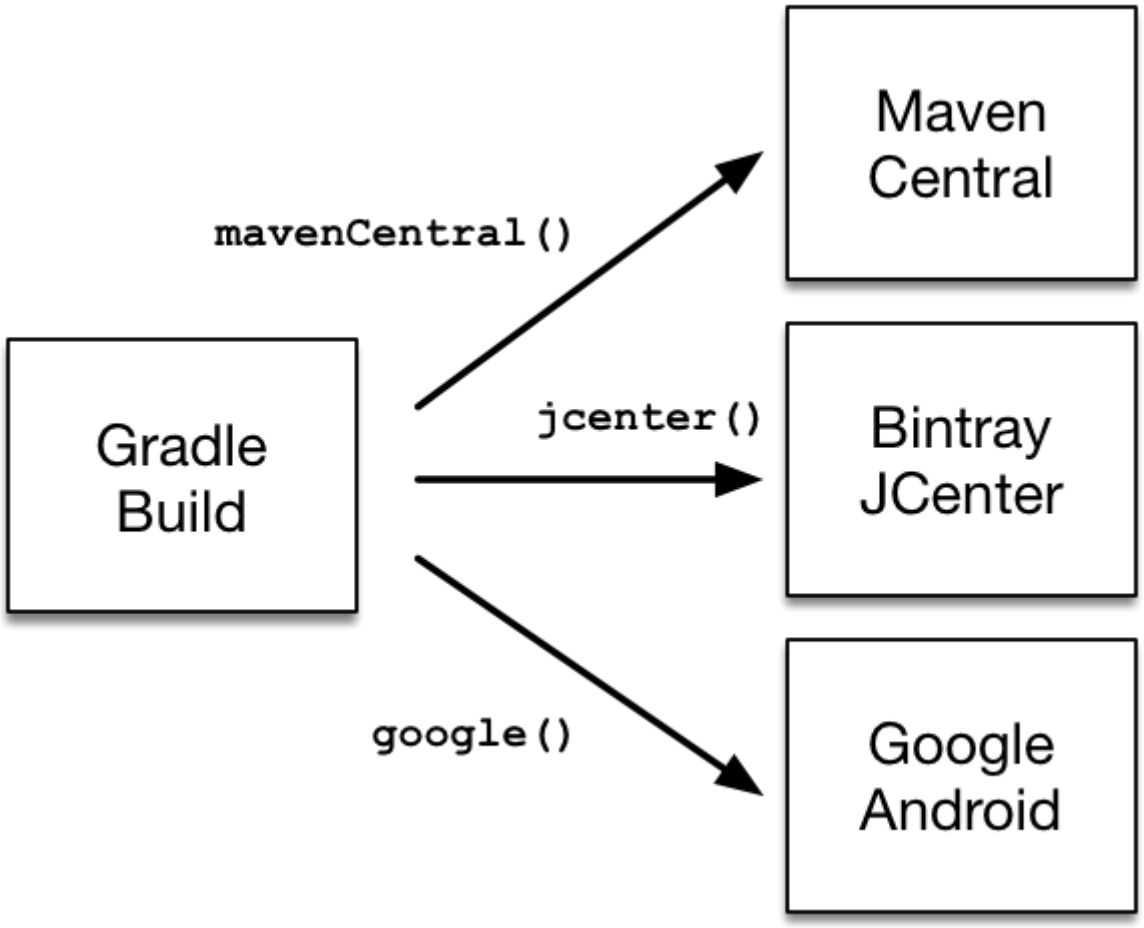


Figure 19. Declaring a repository with the help of shortcut methods

To declare JCenter as repository, add this code to your build script:

Example 291. Declaring JCenter repository as source for resolving dependencies

build.gradle

```
repositories {
    jcenter()
}
```

build.gradle.kts

```
repositories {
    jcenter()
}
```

Under the covers Gradle resolves dependencies from the respective URL of the public repository defined by the shortcut method. All shortcut methods are available via the RepositoryHandler API. Alternatively, you can spell out the URL of the repository for more fine-grained control.

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.

Add the following code to declare an in-house repository for your build reachable through a custom URL.

Example 292. Declaring a custom repository by URL

build.gradle

```
repositories {
    maven {
      url 'http://repo.mycompany.com/maven2'
    }
}
```

build.gradle.kts

```
repositories {
    maven {
      url = uri("http://repo.mycompany.com/maven2")
    }
}
```

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 reference manual on supported transport protocols.

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 shortcut and custom URL repositories for a project:

Example 293. Declaring multiple repositories

build.gradle

```
repositories {
    jcenter()
    maven {
        url "https://maven.springframework.org/release"
    }
    maven {
        url "https://maven.restlet.com"
    }
}
```

build.gradle.kts

```
repositories {
    jcenter()
    maven {
        url = uri("https://maven.springframework.org/release")
    }
    maven {
        url = uri("https://maven.restlet.com")
    }
}
```

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 Gradle's resolution mechanism.

Inspecting 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.

Listing dependencies in a project

A project can declare one or more dependencies. 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.

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 doesn't pollute other contexts like the compilation classpath for your production source code.

Example 294. Declaring the JGit dependency with a custom configuration

build.gradle

```
repositories {
    jcenter()
}

configurations {
    scm
}

dependencies {
    scm 'org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r'
}
```

build.gradle.kts

```
repositories {
    jcenter()
}

configurations {
    create("scm")
}

dependencies {
    "scm"("org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r")
}
```

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.

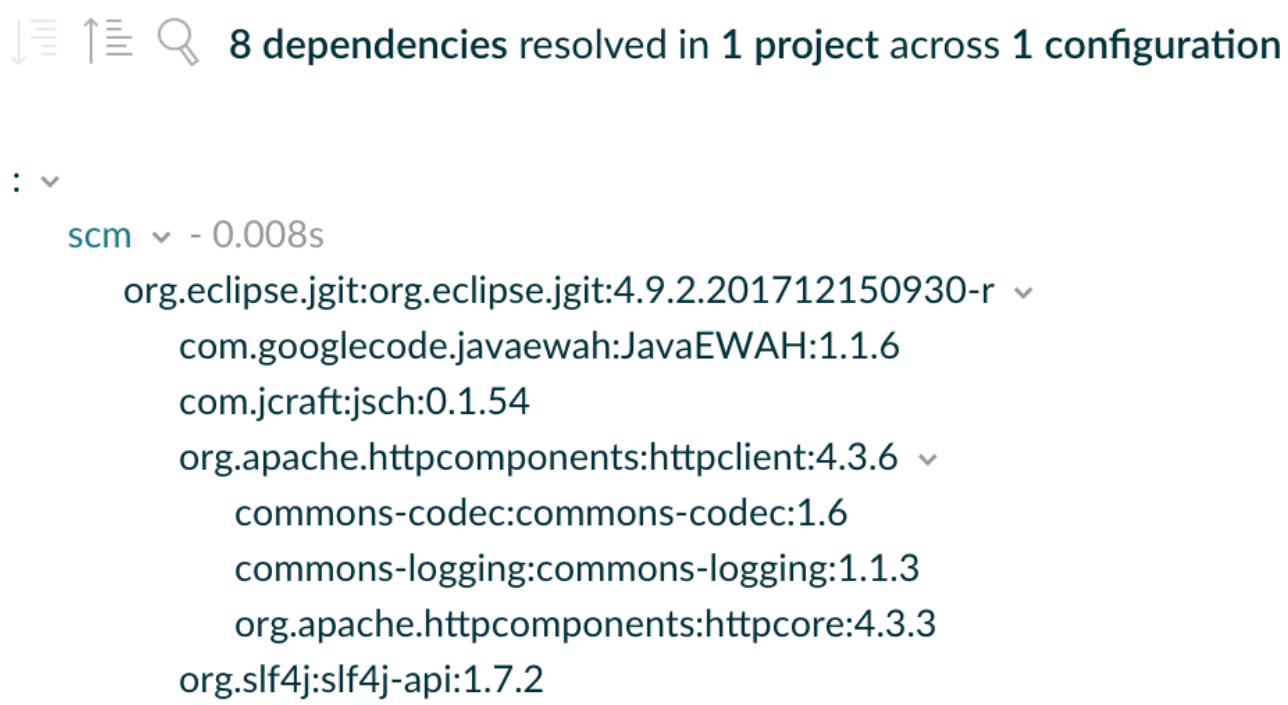


Figure 20. Dependency tree in a build scan

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 pair down on the information provide the optional parameter `--configuration`.

Example: Rendering the dependency report for a custom configuration

Output of `gradle -q dependencies --configuration scm`

```
> gradle -q dependencies --configuration scm

------------------------------------------------------------
Root project
------------------------------------------------------------

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 295. Declaring the JGit dependency and a conflicting dependency

build.gradle

```
repositories {
    jcenter()
}

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 {
    jcenter()
}

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 21. 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. When executing the task you have to provide the mandatory parameter `--dependency` to specify the coordinates of the dependency under inspection. The parameters `--configuration` and `--singlepath` are optional but help with filtering the output.

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" [
      org.gradle.status = release (not requested)
   ]
   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.
```

Justifying dependency declarations with custom reasons

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 296. Giving a reason for choosing a certain module version in a dependency declaration

build.gradle

```
plugins {
    id 'java-library'
}

repositories {
    jcenter()
}

dependencies {
    implementation('org.ow2.asm:asm:6.0') {
        because 'we require a JDK 9 compatible bytecode generator'
    }
}
```

build.gradle.kts

```
plugins {
    `java-library`
}

repositories {
    jcenter()
}

dependencies {
    implementation("org.ow2.asm:asm:6.0") {
        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:6.0
   variant "compile" [
      org.gradle.status             = release (not requested)
      org.gradle.usage              = java-api
      org.gradle.component.category = library (not requested)
   ]
   Selection reasons:
      - Was requested : we require a JDK 9 compatible bytecode generator

org.ow2.asm:asm:6.0
\--- compileClasspath

A web-based, searchable dependency report is available by adding the --scan option.
```

Managing Dependency Configurations

What is a configuration?

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. The sections above demonstrate how to declare dependencies for different use cases.

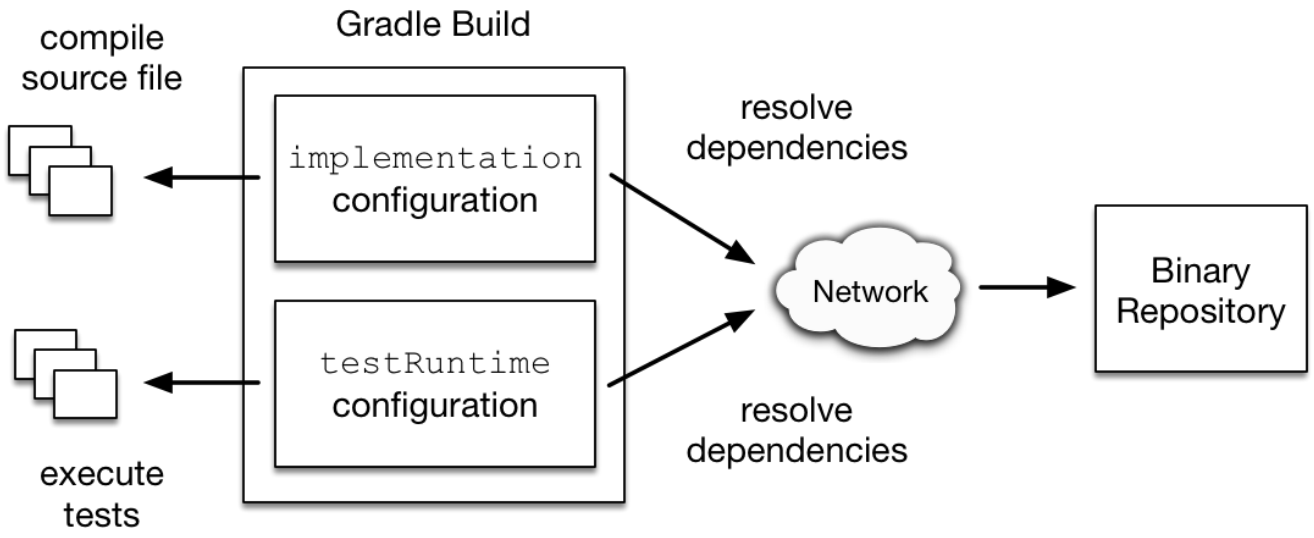


Figure 22. Configurations use declared dependencies for specific purposes

For more examples on the usage of configurations to navigate, inspect and post-process metadata

and artifacts of assigned dependencies, see Working with Dependencies.

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 297. Declaring and using a custom configuration

build.gradle

```
configurations {
    jasper
}

repositories {
    mavenCentral()
}

dependencies {
    jasper 'org.apache.tomcat.embed:tomcat-embed-jasper:9.0.2'
}

task 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")
}

task("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"))
        }
    }
}
```

A project's configurations are managed by a `configurations` object. Configurations have a name and can extend each other. To learn more about this API have a look at ConfigurationContainer.

Inheriting dependencies from other configurations

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.

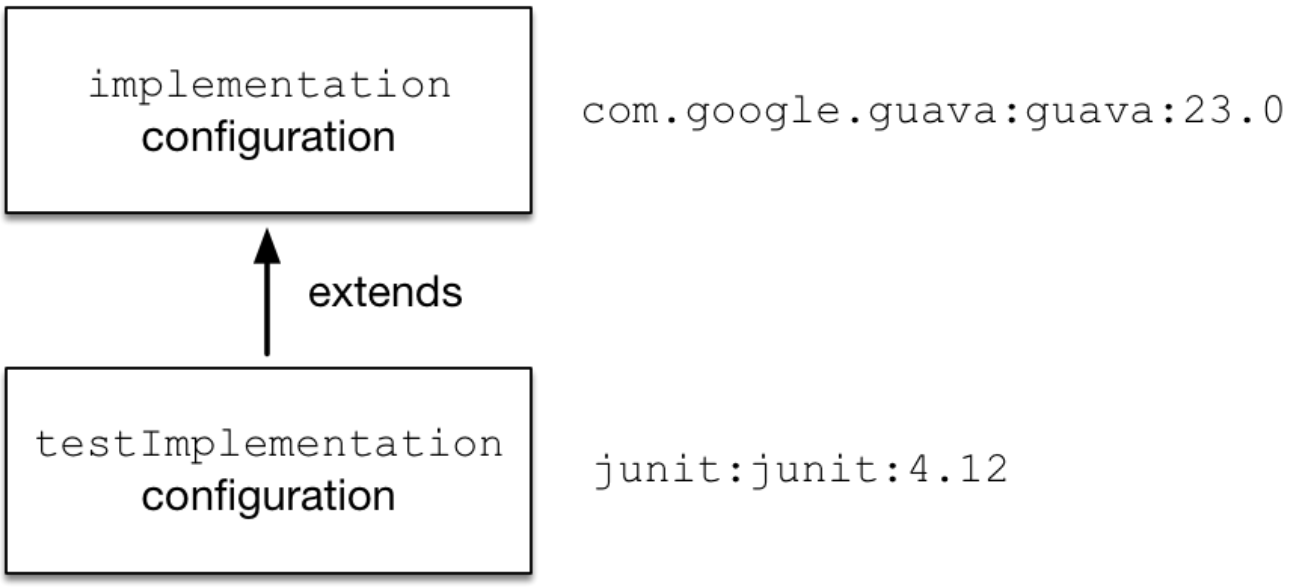


Figure 23. Configuration inheritance provided by the Java plugin

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 298. Extending a configuration from another configuration

build.gradle

```
configurations {
    smokeTest.extendsFrom testImplementation
}

dependencies {
    testImplementation 'junit:junit:4.12'
    smokeTest 'org.apache.httpcomponents:httpclient:4.5.5'
}
```

build.gradle.kts

```
val smokeTest by configurations.creating {
    extendsFrom(configurations.testImplementation.get())
}

dependencies {
    testImplementation("junit:junit:4.12")
    smokeTest("org.apache.httpcomponents:httpclient:4.5.5")
}
```

Managing Transitive Dependencies

Resolution behavior for transitive dependencies can be customized to a high degree to meet enterprise requirements.

Managing versions of transitive dependencies with dependency constraints

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 299. 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 {
        add("implementation", "org.apache.httpcomponents:httpclient:4.5.3") {
            because("previous versions have a bug impacting this
application")
        }
        add("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.

NOTE Dependency constraints are not yet published, but that will be added in a future release. This means that their use currently only targets builds that do not publish artifacts to maven or ivy repositories.

Dependency constraints themselves can also be added transitively.

Excluding transitive module dependencies

Declared dependencies in a build script can pull in a lot of transitive dependencies. You might decide that you do not want a particular transitive dependency as part of the dependency graph for a good reason.

- The dependency is undesired due to licensing constraints.
- The dependency is not available in any of the declared repositories.
- The metadata for the dependency exists but the artifact does not.
- The metadata provides incorrect coordinates for a transitive dependency.

Transitive dependencies can be excluded on the level of a declared dependency or a configuration. Let's demonstrate both use cases. In the following two examples the build script declares a dependency on Log4J, a popular logging framework in the Java world. The metadata of the particular version of Log4J also defines transitive dependencies.

Example 300. Unresolved artifacts for transitive dependencies

build.gradle

```
plugins {
    id 'java'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'log4j:log4j:1.2.15'
}
```

build.gradle.kts

```
plugins {
    java
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("log4j:log4j:1.2.15")
}
```

If resolved from Maven Central some of the transitive dependencies provide metadata but not the corresponding binary artifact. As a result any task requiring the binary files will fail e.g. a compilation task.

```
> gradle -q compileJava

* What went wrong:
Could not resolve all files for configuration ':compileClasspath'.
> Could not find jms.jar (javax.jms:jms:1.1).
  Searched in the following locations:
      https://repo.maven.apache.org/maven2/javax/jms/jms/1.1/jms-1.1.jar
> Could not find jmxtools.jar (com.sun.jdmk:jmxtools:1.2.1).
  Searched in the following locations:
      https://repo.maven.apache.org/maven2/com/sun/jdmk/jmxtools/1.2.1/jmxtools-
1.2.1.jar
> Could not find jmxri.jar (com.sun.jmx:jmxri:1.2.1).
  Searched in the following locations:
      https://repo.maven.apache.org/maven2/com/sun/jmx/jmxri/1.2.1/jmxri-1.2.1.jar
```

The situation can be fixed by adding a repository containing those dependencies. In the given example project, the source code does not actually use any of Log4J's functionality that require the JMS (e.g. `JMSAppender`) or JMX libraries. It's safe to exclude them from the dependency declaration.

Exclusions need to spelled out as a key/value pair via the attributes `group` and/or `module`. For more information, refer to ModuleDependency.exclude(java.util.Map).

Example 301. Excluding transitive dependency for a particular dependency declaration

build.gradle

```
dependencies {
    implementation('log4j:log4j:1.2.15') {
        exclude group: 'javax.jms', module: 'jms'
        exclude group: 'com.sun.jdmk', module: 'jmxtools'
        exclude group: 'com.sun.jmx', module: 'jmxri'
    }
}
```

build.gradle.kts

```
dependencies {
    implementation("log4j:log4j:1.2.15") {
        exclude(group = "javax.jms", module = "jms")
        exclude(group = "com.sun.jdmk", module = "jmxtools")
        exclude(group = "com.sun.jmx", module = "jmxri")
    }
}
```

You may find that other dependencies will want to pull in the same transitive dependency that misses the artifacts. Alternatively, you can exclude the transitive dependencies for a particular configuration by calling the method Configuration.exclude(java.util.Map).

Example 302. Excluding transitive dependency for a particular configuration

build.gradle

```
configurations {
    implementation {
        exclude group: 'javax.jms', module: 'jms'
        exclude group: 'com.sun.jdmk', module: 'jmxtools'
        exclude group: 'com.sun.jmx', module: 'jmxri'
    }
}

dependencies {
    implementation 'log4j:log4j:1.2.15'
}
```

build.gradle.kts

```
configurations {
    "implementation" {
        exclude(group = "javax.jms", module = "jms")
        exclude(group = "com.sun.jdmk", module = "jmxtools")
        exclude(group = "com.sun.jmx", module = "jmxri")
    }
}

dependencies {
    implementation("log4j:log4j:1.2.15")
}
```

NOTE As a build script author you often times know that you want to exclude a dependency for all configurations available in the project. You can use the method DomainObjectCollection.all(org.gradle.api.Action) to define a global rule.

You might encounter other use cases that don't quite fit the bill of an exclude rule. For example you want to automatically select a version for a dependency with a specific requested version or you want to select a different group for a requested dependency to react to a relocation. Those use cases are better solved by the ResolutionStrategy API. Some of these use cases are covered in Customizing Dependency Resolution Behavior.

Enforcing a particular dependency version

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.

NOTE	Enforcing 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.

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 in the build script and setting ExternalDependency.setForce(boolean) to `true`.

Example 303. 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 304. 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")
}
```

Disabling resolution of transitive dependencies

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 305. 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 306. 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")
}
```

Importing version recommendations from a Maven BOM

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 307. 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(dependencies.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 over version found in the graph. You can enable this behavior by using the `enforcedPlatform` keyword, instead of `platform`, when importing the BOM:

Example 308. 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 overriden 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(dependencies.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 overriden by the one found in the BOM
    implementation("org.codehaus.groovy:groovy:1.8.6")
}
```

Dependency version alignment

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 309. 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 platform. 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 modules and indicates that they work well together (typically, the Spring Platform).

Declaring participation in a platform

We can fix the example above by declaring that all Jackson modules "belong to" the same platform.

Example 310. A dependency version alignment rule

build.gradle

```
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-platform:${id
.version}")
            }
        }
    }
}
```

build.gradle.kts

```
open 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-
platform:${id.version}")
            }
        }
    }
}
```

By using the `belongsTo` keyword, we declare that all modules belong to the same *virtual platform*, which is treated specially by the engine, in particular with regards to alignment. We can use the rule we just created by registering it:

Example 311. Making use of a dependency version alignment rule

build.gradle

```
dependencies {
    components.all(JacksonAlignmentRule)
}
```

build.gradle.kts

```
dependencies {
    components.all(JacksonAlignmentRule::class.java)
}
```

Then all versions in the example above would align to `2.9.5`. However, Gradle would let you override that choice by specifying a dependency on the Jackson platform:

Example 312. Forceful platform downgrade

build.gradle

```
dependencies {
    // Forcefully downgrade the Jackson platform to 2.8.9
    implementation enforcedPlatform('com.fasterxml.jackson:jackson-
platform:2.8.9')
}
```

build.gradle.kts

```
dependencies {
    // Forcefully downgrade the Jackson platform to 2.8.9
    implementation(enforcedPlatform("com.fasterxml.jackson:jackson-
platform:2.8.9"))
}
```

Virtual vs published platforms

A platform defined by a component metadata rule for which the `belongsTo` target module isn't published on a repository is called a virtual platform. A virtual platform is considered specially by the engine and participates in dependency resolution like a published module, but triggers

dependency version alignment. On the other hand, we can find "real" platforms published on public repositories. Typical examples include BOMs, like the Spring BOM. They differ in the sense that a published platform may refer to modules which are effectively different things. For example the Spring BOM declares dependencies on Spring as well as Apache Groovy. Obviously those things are versioned differently, so it doesn't make sense to align in this case. In other words, if a platform is *published*, Gradle trusts its metadata, and will not try to align dependency versions of this platform.

NOTE	Gradle doesn't yet support publishing platforms. It can consume BOMs, or declare participation to a virtual platform, but it's not yet possible to declare and publish an adhoc platform module.

Component capabilities

Introduction to 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

NOTE	Capabilities are a core feature of the experimental Gradle metadata file format. This means that components published with the experimental Gradle metadata file format can declare capabilities, but also that this feature is only natively understood by Gradle. However, it's possible to declare capabilities on components which were *not* built by Gradle, as explained in this section.

If your build file contains the following dependencies:

Example 313. 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 pre-emptively detect the conflict by adding a rule which will declare that both logging frameworks provide the same capability:

Example 314. 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("org.slf4j", "slf4j-capability",
"1.0")
                    }
                }
            }
        }
    }
}
```

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("org.slf4j", "slf4j-capability", "1.0")
                }
            }
        }
    }
}
```

By adding this rule, we will make sure that Gradle *will* detect conflicts and properly fail:

```
Cannot choose between log4j:log4j:1.2.16 and org.slf4j:log4j-over-slf4j:1.7.10 because
they provide the same capability: org.slf4j:slf4j-capability:1.0
```

It does not, however, choose what component to use for you: detecting a conflict is the first step, then you have to fix it.

Solving capability conflicts

By default, Gradle will automatically choose the component with the highest version of a capability. This can be useful 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 315. Automatic conflict resolution by capability versioning

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)
                }
            }
        }
    }
}
```

However, fixing by automatic capability version conflict resolution is not always suitable. In our logging example, it doesn't matter what version of the logging frameworks we use, we should always select the slf4j bridge.

In this case, we can fix it by using dependency substitution:

Example 316. Substitute log4j with slf4j

build.gradle

```
    configurations.compileClasspath.resolutionStrategy.dependencySubstitution
{
        substitute(module("log4j:log4j"))
            .because("Prefer SLF4J for logging")
            .with(module("org.slf4j:log4j-over-slf4j:1.7.10"))

    }
```

build.gradle.kts

```
    configurations.compileClasspath.resolutionStrategy.dependencySubstitution
{
        substitute(module("log4j:log4j"))
            .because("Prefer SLF4J for logging")
            .with(module("org.slf4j:log4j-over-slf4j:1.7.10"))

    }
```

Dependency Locking

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.

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 317. Locking a specific configuration

build.gradle

```
configurations {
    compileClasspath {
        resolutionStrategy.activateDependencyLocking()
    }
}
```

build.gradle.kts

```
configurations.compileClasspath {
    resolutionStrategy.activateDependencyLocking()
}
```

Or the following, as a way to lock all configurations:

Example 318. Locking all configurations

build.gradle

```
dependencyLocking {
    lockAllConfigurations()
}
```

build.gradle.kts

```
dependencyLocking {
    lockAllConfigurations()
}
```

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.

NOTE The above will lock all *project* configurations, but not the *buildscript* ones.

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 319. 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.

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 will resolve all configurations

Example 320. Resolving all configurations

build.gradle

```
task 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

```
task("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 choosing 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 in the folder `gradle/dependency-locks` inside the project or subproject directory. Each file is named by the configuration it locks and has the `lockfile` extension. The one exception to this rule is for configurations for the buildscript itself. In that case the configuration name will be prefixed with `buildscript-`.

The content of the file is a module notation per line, with a header giving some context. Module notations are ordered alphabetically, to ease diffs.

gradle/dependency-locks/compileClasspath.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
org.springframework:spring-core:5.0.5.RELEASE
org.springframework:spring-jcl:5.0.5.RELEASE
```

which matches the following dependency declaration:

Example 321. Dynamic dependency declaration

build.gradle

```
dependencies {
    implementation 'org.springframework:spring-beans:[5.0,6.0)'
}
```

build.gradle.kts

```
dependencies {
    implementation("org.springframework:spring-beans:[5.0,6.0)")
}
```

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

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. Remove the file matching the configurations where you no longer want locking.

If you only perform the second step above, then locking will effectively no longer be applied. However, if that configuration happens to be resolved in the future at a time where lock state is persisted, it will once again be locked.

Locking limitations

- Locking can not yet be applied to source dependencies.

Nebula locking plugin

This feature is inspired by the Nebula Gradle dependency lock plugin.

Troubleshooting Dependency Resolution

Managing dependencies in a project can be challenging. This chapter describes techniques for troubleshooting issues you might encounter in your project as well as best practices for avoiding common problems.

Resolving version conflicts

Gradle resolves version conflicts by picking the highest version of a module. Build scans and the dependency insight report are immensely helpful in identifying why a specific version was

selected. If the resolution result is not satisfying (e.g. the selected version of a module is too high) or it fails (because you configured ResolutionStrategy.failOnVersionConflict()) you have the following possibilities to fix it.

- Configuring any dependency (transitive or not) as *forced*. This approach is useful if the dependency in conflict is a transitive dependency. See Enforcing a particular dependency version for examples.
- Configuring dependency resolution to *prefer modules that are part of your build* (transitive or not). This approach is useful if your build contains custom forks of modules (as part of multi-project builds or as include in composite builds). See ResolutionStrategy.preferProjectModules() for more information.
- Using dependency resolve rules for fine-grained control over the version selected for a particular dependency.

Using dynamic versions and changing modules

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!

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 322. 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 323. 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

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.

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.

Forcing all dependencies to be re-resolved

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.

Locking dependency versions

The use of dynamic dependencies in a build is convenient. The user does not need to know the latest version of a dependency and Gradle automatically uses new versions once they are published. However, dynamic dependencies make builds non-reproducible, as they can resolve to a different version at a later point in time. This makes it hard to reproduce old builds when debugging a problem. It can also disrupt development if a new, but incompatible version is selected. In the best case the CI build catches the problem and someone needs to investigate. In the worst case, the problem makes it to production unnoticed.

Gradle offers dependency locking to solve this problem. The user can run a build asking to persist the resolved versions for every module dependency. This file is then checked in and the versions in it are used on all subsequent runs until the lock is updated or removed again.

Versioning of file dependencies

Legacy projects sometimes prefer to consume file dependencies instead of module dependencies. File dependencies can point to any file in the filesystem and do not need to adhere a specific naming convention. 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's much easier to uncover potential API incompatibilities by the assigned version.

Constraints on configuration resolution

Configurations need to be resolved safely when crossing project boundaries because resolving configurations can have side effects on Gradle's project model. Gradle can usually manage this safe access, but the configuration needs to be accessed in a way that enables Gradle to do so. There are a number of ways a configuration might be resolved unsafely and Gradle will produce a deprecation warning for each unsafe access.

For example:

- A task from one project directly resolves a configuration in another project.
- A task specifies a configuration from another project as an input file collection.
- A build script for a project resolves a configuration in another project during evaluation.
- A configuration is resolved in a user-managed thread (i.e., a thread not managed by Gradle).

If your build has an unsafe access deprecation warning, it needs to be fixed. It's a symptom of these bad practices and cause strange and indeterminate errors.

More importantly, resolving a configuration from a user-managed thread is not supported. To ensure that the configuration is resolved safely, it must be resolved in a Gradle-managed thread. Afterwards, the resolution result can be used in user-managed threads.

In most cases, the deprecation warning can be fixed by defining a configuration in the project where the resolution is occurring and setting it to extend from the configuration in the other project.

Customizing Dependency Resolution Behavior

There are a number of ways that you can influence how Gradle resolves dependencies. All of these mechanisms offer an API to define a reason for why they are used. Providing reasons makes dependency resolution results more understandable. If any customization influenced the resolution result, the provided reason will show up in dependency insight report.

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.

Modelling releasable units

Often an organisation publishes a set of libraries with a single version; where the libraries are built,

tested and published together. These libraries form a "releasable unit", designed and intended to be used as a whole. It does not make sense to use libraries from different releasable units together.

But it is easy for transitive dependency resolution to violate this contract. For example:

- `module-a` depends on `releasable-unit:part-one:1.0`
- `module-b` depends on `releasable-unit:part-two:1.1`

A build depending on both `module-a` and `module-b` will obtain different versions of libraries within the releasable unit.

Dependency resolve rules give you the power to enforce releasable units in your build. Imagine a releasable unit defined by all libraries that have `org.gradle` group. We can force all of these libraries to use a consistent version:

Example 324. Forcing a consistent version for a group of libraries

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.group == 'org.gradle') {
            details.useVersion '1.4'
            details.because 'API breakage in higher versions'
        }
    }
}
```

build.gradle.kts

```
configurations.all {
    resolutionStrategy.eachDependency {
        if (requested.group == "org.gradle") {
            useVersion("1.4")
            because("API breakage in higher versions")
        }
    }
}
```

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 325. 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")
}
```

Blacklisting a particular version with a replacement

Dependency resolve rules provide a mechanism for blacklisting 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 326. 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")
        }
    }
}
```

Substituting a dependency module with a compatible replacement

At times a completely different module can serve as a replacement for a requested module dependency. Examples include using `groovy` in place of `groovy-all`, or using `log4j-over-slf4j`

instead of `log4j`. You can perform these substitutions using dependency resolve rules:

Example 327. Changing dependency group and/or name during resolution

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.name == 'groovy-all') {
            details.useTarget group: details.requested.group, name: 'groovy',
version: details.requested.version
            details.because "prefer 'groovy' over 'groovy-all'"
        }
        if (details.requested.name == 'log4j') {
            details.useTarget "org.slf4j:log4j-over-slf4j:1.7.10"
            details.because "prefer 'log4j-over-slf4j' 1.7.10 over any
version of 'log4j'"
        }
    }
}
```

build.gradle.kts

```
configurations.all {
    resolutionStrategy.eachDependency {
        if (requested.name == "groovy-all") {
            useTarget(mapOf("group" to requested.group, "name" to "groovy",
"version" to requested.version))
            because("""prefer "groovy" over "groovy-all"""")
        }
        if (requested.name == "log4j") {
            useTarget("org.slf4j:log4j-over-slf4j:1.7.10")
            because("""prefer "log4j-over-slf4j" 1.7.10 over any version of
"log4j"""")
        }
    }
}
```

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 328. Substituting a module with a project

build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute module("org.utils:api") because "we work with the
unreleased development version" with project(":api")
        substitute module("org.utils:util:2.5") with project(":util")
    }
}
```

build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(module("org.utils:api")).apply {
            with(project(":api"))
            because("we work with the unreleased development version")
        }
        substitute(module("org.utils:util:2.5")).with(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 329. Substituting a project with a module

build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute project(":api") because "we use a stable version of
org.utils:api" with module("org.utils:api:1.3")
    }
}
```

build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(project(":api")).apply {
            with(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 330. 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.

Using component metadata rules

Each module has metadata associated with it, such as its group, name, version, dependencies, and so on. This metadata typically originates in the module's descriptor. Metadata rules allow certain parts of a module's metadata to be manipulated from within the build script. They take effect after a module's descriptor has been downloaded, but before it has been selected among all candidate versions. This makes metadata rules another instrument for customizing dependency resolution.

One piece of module metadata that Gradle understands is a module's *status scheme*. This concept, also known from Ivy, models the different levels of maturity that a module transitions through over time. The default status scheme, ordered from least to most mature status, is `integration`, `milestone`, `release`. Apart from a status scheme, a module also has a (current) *status*, which must be one of the values in its status scheme. If not specified in the (Ivy) descriptor, the status defaults to `integration` for Ivy modules and Maven snapshot modules, and `release` for Maven modules that aren't snapshots.

A module's status and status scheme are 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, with the default status scheme in place, `latest.integration` will select the highest module version regardless of its status (because `integration` is the least mature status), whereas `latest.release` will select the highest module version with status `release`. Here is what this looks like in code:

Example 331. 'Latest' version selector

build.gradle

```
dependencies {
    config1 "org.sample:client:latest.integration"
    config2 "org.sample:client:latest.release"
}

task listConfigs {
    doLast {
        configurations.config1.each { println it.name }
        println()
        configurations.config2.each { println it.name }
    }
}
```

build.gradle.kts

```
dependencies {
    "config1"("org.sample:client:latest.integration")
    "config2"("org.sample:client:latest.release")
}

task("listConfigs") {
    doLast {
        configurations["config1"].forEach { println(it.name) }
        println()
        configurations["config2"].forEach { println(it.name) }
    }
}
```

Output of `gradle -q listConfigs`

```
> gradle -q listConfigs
client-1.5.jar

client-1.4.jar
```

The next example demonstrates `latest` selectors based on a custom status scheme declared in a component metadata rule that applies to all modules:

Example 332. Custom status scheme

build.gradle

```
class CustomStatusRule implements ComponentMetadataRule {
    @Override
    void execute(ComponentMetadataContext context) {
        def details = context.details
        if (details.id.group == "org.sample" && details.id.name == "api") {
            details.statusScheme = ["bronze", "silver", "gold", "platinum"]
        }
    }
}

dependencies {
    config3 "org.sample:api:latest.silver"
    components {
        all(CustomStatusRule)
    }
}
```

build.gradle.kts

```
class CustomStatusRule : ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        val details = context.details
        if (details.id.group == "org.sample" && details.id.name == "api") {
            details.statusScheme = listOf("bronze", "silver", "gold",
"platinum")
        }
    }
}

dependencies {
    "config3"("org.sample:api:latest.silver")
    components {
        all(CustomStatusRule::class.java)
    }
}
```

Component metadata rules can be applied to a specified module. Modules must be specified in the form of `group:module`.

Example 333. Custom status scheme by module

build.gradle

```
class ModuleStatusRule implements ComponentMetadataRule {
    @Override
    void execute(ComponentMetadataContext context) {
        context.details.statusScheme = ["int", "rc", "prod"]
    }
}

dependencies {
    config4 "org.sample:lib:latest.prod"
    components {
        withModule('org.sample:lib', ModuleStatusRule)
    }
}
```

build.gradle.kts

```
class ModuleStatusRule : ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        context.details.statusScheme = listOf("int", "rc", "prod")
    }
}

dependencies {
    "config4"("org.sample:lib:latest.prod")
    components {
        withModule("org.sample:lib", ModuleStatusRule::class.java)
    }
}
```

Gradle can also provide to component metadata rules the Ivy-specific metadata for modules resolved from an Ivy repository. Values from the Ivy descriptor are made available via the IvyModuleDescriptor interface.

Example 334. Ivy component metadata rule

build.gradle

```
class IvyComponentRule implements ComponentMetadataRule {
    @Override
    void execute(ComponentMetadataContext context) {
        def descriptor = context.getDescriptor(IvyModuleDescriptor)
        if (descriptor != null && descriptor.branch == 'testing') {
            context.details.status = "rc"
        }
    }
}
dependencies {
    config5 "org.sample:lib:latest.rc"
    components {
        withModule("org.sample:lib", IvyComponentRule)
    }
}
```

build.gradle.kts

```
class IvyComponentRule : ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        val descriptor = context.getDescriptor(IvyModuleDescriptor::class)
        if (descriptor != null && descriptor.branch == "testing") {
            context.details.status = "rc"
        }
    }
}
dependencies {
    "config5"("org.sample:lib:latest.rc")
    components {
        withModule("org.sample:lib", IvyComponentRule::class.java)
    }
}
```

Note that while any rule can request the IvyModuleDescriptor, only components sourced from an Ivy repository will have a non-null value for it.

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.

The next example shows how you can configure the `ComponentMetadataRule` through an

ActionConfiguration.

Example 335. Configuration of ComponentMetadataRule

build.gradle

```
class ConfiguredRule implements ComponentMetadataRule {
    String param

    @javax.inject.Inject
    ConfiguredRule(String param) {
        this.param = param
    }

    @Override
    void execute(ComponentMetadataContext context) {
        if (param == 'sampleValue') {
            context.details.statusScheme = ["bronze", "silver", "gold",
"platinum"]
        }
    }
}
dependencies {
    config6 "org.sample:api:latest.gold"
    components {
        withModule('org.sample:api', ConfiguredRule, {
            params('sampleValue')
        })
    }
}
```

build.gradle.kts

```
class ConfiguredRule @javax.inject.Inject constructor(val param: String) :
ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        if (param == "sampleValue") {
            context.details.statusScheme = listOf("bronze", "silver", "gold",
"platinum")
        }
    }
}
dependencies {
    "config6"("org.sample:api:latest.gold")
    components {
        withModule("org.sample:api", ConfiguredRule::class.java) {
            params("sampleValue")
        }
    }
}
```

This happens 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 passed in parameters must be `Serializable` or known Gradle types that can be isolated.

In addition, Gradle services can be injected into your `ComponentMetadataRule`. This is for the moment limited to the RepositoryResourceAccessor. Because of this, the moment you have a constructor, it must be annotated with `@javax.inject.Inject`.

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 336. 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 337. 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 338. 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 have testing
branch or 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.

Lastly, component selection rules can also be defined using a *rule source* object. A rule source object is any object that contains exactly one method that defines the rule action and is annotated with `@Mutate`.

This method:

- must return void.
- must have ComponentSelection as its argument.

Example 339. Component selection rule using a rule source object

build.gradle

```
class RejectTestBranch {
    @Mutate
    void evaluateRule(ComponentSelection selection) {
        if (selection.getDescriptor(IvyModuleDescriptor)?.branch == "test") {
            selection.reject("reject test branch")
        }
    }
}

configurations {
    ruleSourceConfig {
        resolutionStrategy {
            componentSelection {
                all new RejectTestBranch()
            }
        }
    }
}
```

build.gradle.kts

```
class RejectTestBranch {
    @Mutate
    fun evaluateRule(selection: ComponentSelection) {
        if (selection.getDescriptor(IvyModuleDescriptor::class)?.branch ==
"test") {
            selection.reject("reject test branch")
        }
    }
}

configurations {
    create("ruleSourceConfig") {
        resolutionStrategy {
            componentSelection {
                all(RejectTestBranch())
            }
        }
    }
}
```

NOTE	Declaring additional arguments on component selection rules is deprecated and scheduled for removal in Gradle 6.0. Use instead the added methods on ComponentSelection.

Using module replacement rules

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. Starting from Gradle 2.2 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 340. 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.

Modifying dependencies for a configuration

At times, a plugin needs to modify or enhance the dependencies declared by a user. The following methods on `Configuration` provide a mechanism to achieve this.

Specifying default dependencies for a configuration

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 341. 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"))
        }
    }
}
```

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 342. 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")
            }
        }
    }
}
```

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 343. 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
}
```

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 meta-data, 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 meta-data store is being read or written, but is released for slow operations such as downloading remote artifacts.

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.

Working with Dependencies

Gradle provides an extensive API for navigating, inspecting and post-processing metadata and artifacts of resolved dependencies.

The main entry point for this functionality is the Configuration API. To learn more about the fundamentals of configurations, see Managing Dependency Configurations.

Iterating over dependencies assigned to a configuration

Sometimes you'll want to implement logic based on the dependencies declared in the build script of a project e.g. to inspect them in a Gradle plugin. You can iterate over the set of dependencies assigned to a configuration with the help of the method Configuration.getDependencies(). Alternatively, you can also use Configuration.getAllDependencies() to include the dependencies declared in superconfigurations. These APIs only return the declared dependencies and do not trigger dependency resolution. Therefore, the dependency sets do not include transitive dependencies. Calling the APIs during the configuration phase of the build lifecycle does not result in a significant performance impact.

Example 344. Iterating over the dependencies assigned to a configuration

build.gradle

```
task iterateDeclaredDependencies {
    doLast {
        DependencySet dependencySet = configurations.scm.dependencies

        dependencySet.each {
            logger.quiet "$it.group:$it.name:$it.version"
        }
    }
}
```

build.gradle.kts

```
task("iterateDeclaredDependencies") {
    doLast {
        val dependencySet = configurations["scm"].dependencies

        dependencySet.forEach {
            logger.quiet("${it.group}:${it.name}:${it.version}")
        }
    }
}
```

Iterating over artifacts resolved for a module

None of the dependency reporting helps you with inspecting or further processing the underlying, resolved artifacts of a module. A typical use case for accessing the artifacts is to copy them into a specific directory or filter out files of interest based on a specific file extension.

You can iterate over the complete set of artifacts resolved for a module with the help of the method FileCollection.getFiles(). Every file instance returned from the method points to its location in the dependency cache. Using this method on a `Configuration` instance is possible as the interface extends `FileCollection`.

Example 345. Iterating over the artifacts resolved for a module

build.gradle

```
task iterateResolvedArtifacts {
    dependsOn configurations.scm

    doLast {
        configurations.scm.each {
            logger.quiet it.absolutePath
        }
    }
}
```

build.gradle.kts

```
task("iterateResolvedArtifacts") {
    val scm = configurations["scm"]
    dependsOn(scm)

    doLast {
        scm.forEach {
            logger.quiet(it.absolutePath)
        }
    }
}
```

NOTE Iterating over the artifacts of a module automatically resolves the configuration. A resolved configuration becomes immutable and cannot add or remove dependencies. If needed you can copy a configuration for further modification via Configuration.copy().

Navigating the dependency graph

As a plugin developer, you may want to navigate the full graph of dependencies assigned to a configuration e.g. for turning the dependency graph into a visualization. You can access the full graph of dependencies for a configuration with the help of the ResolutionResult.

The resolution result provides various methods for accessing the resolved and unresolved dependencies. For demonstration purposes the sample code uses ResolutionResult.getRoot() to access the root node the resolved dependency graph. Each dependency of this component returns an instance of ResolvedDependencyResult or UnresolvedDependencyResult providing detailed information about the node.

Example 346. Walking the resolved and unresolved dependencies of a configuration

build.gradle

```
task walkDependencyGraph(type: DependencyGraphWalk) {
    dependsOn configurations.scm
}

class DependencyGraphWalk extends DefaultTask {
    @TaskAction
    void walk() {
        Configuration configuration = project.configurations.scm
        ResolutionResult resolutionResult = configuration.incoming
.resolutionResult
        ResolvedComponentResult root = resolutionResult.root
        logger.quiet configuration.name
        traverseDependencies(0, root.dependencies)
    }

    private void traverseDependencies(int level, Set<? extends
DependencyResult> results) {
        for (DependencyResult result : results) {
            if (result instanceof ResolvedDependencyResult) {
                ResolvedComponentResult componentResult = result.selected
                ComponentIdentifier componentIdentifier = componentResult.id
                String node = calculateIndentation(level) + "-
$componentIdentifier.displayName ($componentResult.selectionReason)"
                logger.quiet node
                traverseDependencies(level + 1, componentResult.dependencies)
            } else if (result instanceof UnresolvedDependencyResult) {
                ComponentSelector componentSelector = result.attempted
                String node = calculateIndentation(level) + "-
$componentSelector.displayName (failed)"
                logger.quiet node
            }
        }
    }

    private String calculateIndentation(int level) {
        '    ' * level
    }
}
```

build.gradle.kts

```
task<DependencyGraphWalk>("walkDependencyGraph") {
    dependsOn(configurations["scm"])
}

open class DependencyGraphWalk: DefaultTask() {
    @TaskAction
    fun walk() {
        val configuration: Configuration =
project.configurations.getByName("scm")
        val resolutionResult: ResolutionResult =
configuration.incoming.resolutionResult
        val root: ResolvedComponentResult = resolutionResult.root
        logger.quiet(configuration.name)
        traverseDependencies(0, root.dependencies)
    }

    private fun traverseDependencies(level: Int, results:
Set<DependencyResult>) {
        results.forEach { result ->
            if (result is ResolvedDependencyResult) {
                val componentResult: ResolvedComponentResult =
result.selected
                val componentIdentifier: ComponentIdentifier =
componentResult.id
                val node: String = "${calculateIndentation(level)}-
${componentIdentifier.displayName} (${componentResult.selectionReason})"
                logger.quiet(node)
                traverseDependencies(level + 1, componentResult.dependencies)
            } else if (result is UnresolvedDependencyResult) {
                val componentSelector: ComponentSelector = result.attempted
                val node: String = "${calculateIndentation(level)}-
${componentSelector.displayName} (failed)"
                logger.quiet(node)
            }
        }
    }

    private fun calculateIndentation(level: Int) = "     ".repeat(level)
}
```

Accessing a module's metadata file

As part of the dependency resolution process, Gradle downloads the metadata file of a module and stores it in the dependency cache. Some organizations enforce strong restrictions on accessing repositories outside of internal network. Instead of downloading artifacts, those organizations

prefer to provide an "installable" Gradle cache with all artifacts contained in it to fulfill the build's dependency requirements.

The artifact query API provides access to the raw files of a module. Currently, it allows getting a handle to the metadata file and some selected, additional artifacts (e.g. a JVM-based module's source and Javadoc files). The main API entry point is ArtifactResolutionQuery.

Let's say you wanted to post-process the metadata file of a Maven module. The group, name and version of the module component serve as input to the artifact resolution query. After executing the query, you get a handle to all components that match the criteria and their underlying files. Additionally, it's very easy to post-process the metadata file. The example code uses Groovy's XmlSlurper to ask for POM element values.

Example 347. Accessing a Maven module's metadata artifact

build.gradle

```
plugins {
    id 'java-library'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'com.google.guava:guava:18.0'
}

task printGuavaMetadata {
    dependsOn configurations.compileClasspath

    doLast {
        ArtifactResolutionQuery query = dependencies
.createArtifactResolutionQuery()
            .forModule('com.google.guava', 'guava', '18.0')
            .withArtifacts(MavenModule, MavenPomArtifact)
        ArtifactResolutionResult result = query.execute()

        for(component in result.resolvedComponents) {
            Set<ArtifactResult> mavenPomArtifacts = component.getArtifacts
(MavenPomArtifact)
            ArtifactResult guavaPomArtifact = mavenPomArtifacts.find { it
.file.name == 'guava-18.0.pom' }
            def xml = new XmlSlurper().parse(guavaPomArtifact.file)
            println guavaPomArtifact.file.name
            println xml.name
            println xml.description
        }
    }
}
```

build.gradle.kts

```
import groovy.util.XmlSlurper

plugins {
    `java-library`
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("com.google.guava:guava:18.0")
}

task("printGuavaMetadata") {
    dependsOn(configurations.compileClasspath)

    doLast {
        val query: ArtifactResolutionQuery =
dependencies.createArtifactResolutionQuery()
            .forModule("com.google.guava", "guava", "18.0")
            .withArtifacts(MavenModule::class, MavenPomArtifact::class)
        val result: ArtifactResolutionResult = query.execute()

        result.resolvedComponents.forEach { component ->
            val mavenPomArtifacts: Set<ArtifactResult> =
component.getArtifacts(MavenPomArtifact::class)
            val guavaPomArtifact =
                mavenPomArtifacts.find { it is ResolvedArtifactResult &&
it.file.name == "guava-18.0.pom" } as ResolvedArtifactResult
            val xml = XmlSlurper().parse(guavaPomArtifact.file)
            println(guavaPomArtifact.file.name)
            println(xml.getProperty("name"))
            println(xml.getProperty("description"))
        }
    }
}
```

Attribute based matching in dependency resolution

This chapter is primarily aimed towards plugin authors who want to understand better how to leverage the capabilities of the dependency resolution engine to support variant-aware dependency management. Users who simply want to understand what configuration attributes are will also find support here.

Different kinds of configurations

Historically, configurations have been at the root of dependency resolution in Gradle. In the end, what we want to make a diffence is between a *consumer* and a *producer*. For this purpose, configurations are used for at least 3 different aspects:

1. to declare dependencies
2. as a *consumer*, to resolve a set of dependencies to files
3. as a *producer*, to expose artifacts for consumption by other projects

For example, if I want to express that my application `app` *depends on* library `lib`, we need *at least* one configuration:

Example 348. 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 extend other configuration, in order to inherit their dependencies. However, the code above doesn't tell anything about the *consumer*. In particular, it doesn't tell what is the *use* of the configuration. Let's say that `lib` is a Java library: it can expose different things, such as its API, implementation or test fixtures. If we want to resolve the dependencies of `app`, we need to know what kind of task we're performing (compiling against the API of `lib`, executing the application, compiling tests, ...). For this purpose, you'll often find companion configurations, which are meant to unambiguously declare the usage:

Example 349. 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 stage, we have 3 different configurations, which already have different goals:

- `someConfiguration` declares the dependencies of my application. It's just a bucket where we declare a list of dependencies.
- `compileClasspath` and `runtimeClasspath` are configurations *meant to be resolved*: when resolved they should contain respectively the compile classpath, and the runtime classpath of the application.

This is actually represented on the `Configuration` type by the `canBeResolved` flag. 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 extend, 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 for consumers looking for its API. Such a configuration is going to be consumable, but is not meant to be resolved. This is expressed via the *canBeConsumed* flag of a `Configuration`:

Example 350. 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 role is determined by the `canBeResolved` and `canBeConsumed` flag

combinations:

Table 9. 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, those flags have both `true` as the default value, but as a plugin author, you should always determine the right values for those flags, or you might accidentally introduce resolution errors.

Configuration attributes

We have explained that we have 3 configuration roles, and explained that we may want to resolve the compile and runtime classpath differently, but there's nothing in what we've written which allows explaining the difference. This is where *attributes* come into play. The role of attributes is to perform the selection of the right *variant* of a component. In our example, the `lib` library exposes 2 variants: its API (via `exposedApi`) and its runtime (via `exposedRuntime`). There's no restriction on the number of variants a component can expose. We may, for example, want to expose the test fixtures of a component too. But then, the consumer needs to explain *what* configuration to consume, and this is done by setting *attributes* on both the *consumer* and *producer* ends.

Attributes consist of a *name* and a *value* pair. Gradle comes with a standard attribute named `org.gradle.usage` specifically to deal with the concept of selecting the right variant of a component based on the usage of the consumer (compile, runtime …). It is however possible to define an arbitrary number of attributes. As a producer, I can express that a consumable configuration represents the API of a component by attaching the `(org.gradle.usage,JAVA_API)` attribute to the configuration. As a consumer, I can express that I need the API of the dependencies of a resolvable configuration by attaching the `(org.gradle.usage,JAVA_API)` attribute to it. Now Gradle has a way to *automatically select the appropriate variant* by looking at the configuration attributes: - the consumer wants `org.gradle.usage=JAVA_API` - the dependent project exposes 2 different variants. One with `org.gradle.usage=JAVA_API`, the other with `org.gradle.usage=JAVA_RUNTIME`. - Gradle selects the `org.gradle.usage=JAVA_API` variant

In other words: attributes are used to perform the selection based on the values of the attributes. It doesn't matter what the names of the configurations are: only the attributes matter.

Declaring attributes

Attributes are *typed*. An attribute can be created via the `Attribute<T>.of` method:

Example 351. 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)
```

Currently, only attribute types of `String`, or anything extending `Named` is supported. Attributes must be declared in the *attribute schema* found on the `dependencies` handler:

Example 352. 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 353. 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 354. 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 compatibility rules

Attributes let the engine select *compatible variants*. However, there are cases where a provider may not have *exactly* what the consumer wants, but still something that it can use. For example, if the consumer is asking for the API of a library, there's a possibility that the producer doesn't have such a variant, but only a *runtime* variant. This is typical of libraries published on 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). To deal with this, Gradle provides attribute compatibilty rules. The role of a compatibility rule is to explain what variants are *compatible* with 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

Because multiple values for an attribute can be *compatible* with the requested attribute, Gradle needs to choose between the candidates. 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.

Publishing Artifacts

Publishing

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.

NOTE Looking for information on upload tasks and the `archives` configuration? See the Legacy Publishing chapter.

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.

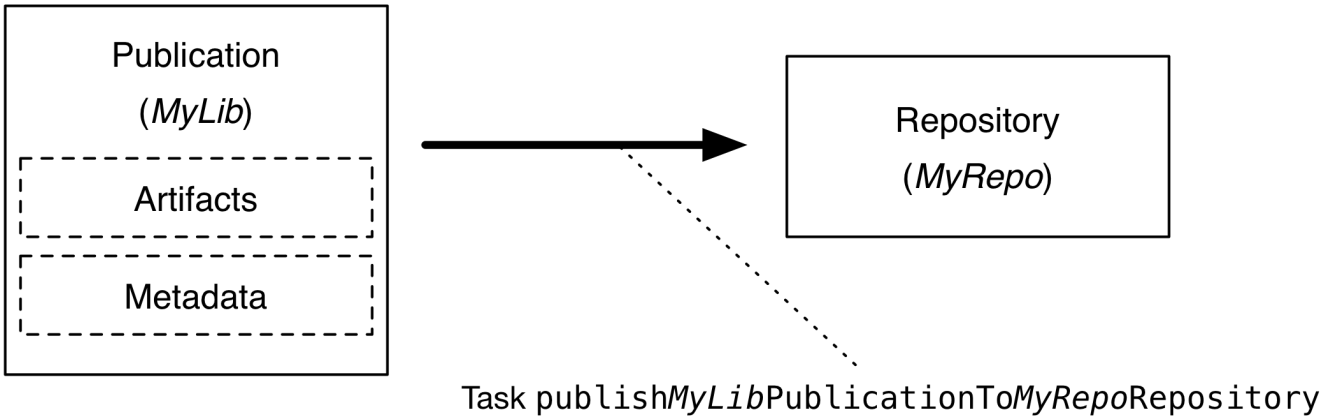


Figure 24. The publishing process

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 — plus a POM file describing 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.

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 355. 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 356. 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 = "file://${buildDir}/repo"
        }
    }
}
```

build.gradle.kts

```
group = "org.example"
version = "1.0"

publishing {
    publications {
        create<MavenPublication>("myLibrary") {
            from(components["java"])
        }
    }

    repositories {
        maven {
            name = "myRepo"
            url = uri("file://${buildDir}/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.

NOTE 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.

Adding custom artifacts to a publication

Users often need to include additional artifacts with a publication, one of the most common examples being that of "-sources" and "-javadoc" JARs for JVM libraries. This is easy to do for both Maven- and Ivy-compatible repositories via the `artifact` configuration.

The following sample configures "-sources" and "-javadoc" JARs for a Java project and attaches them to the main (Maven) publication, i.e. the production JAR:

Example 357. Adding an additional archive artifact to a MavenPublication

build.gradle

```
task sourcesJar(type: Jar) {
    classifier = 'sources'
    from sourceSets.main.allJava
}

task javadocJar(type: Jar) {
    classifier = 'javadoc'
    from javadoc.destinationDir
}

publishing {
    publications {
        mavenJava(MavenPublication) {
            from components.java

            artifact sourcesJar
            artifact javadocJar
        }
    }
}
```

build.gradle.kts

```
task<Jar>("sourcesJar") {
    classifier = "sources"
    from(sourceSets.main.get().allJava)
}

task<Jar>("javadocJar") {
    classifier = "javadoc"
    from(tasks.javadoc.get().destinationDir)
}

publishing {
    publications {
        create<MavenPublication>("mavenJava") {
            from(components["java"])

            artifact(tasks["sourcesJar"])
            artifact(tasks["javadocJar"])
        }
    }
}
```

There are several important things to note about the sample:

- The `artifact()` method accepts archive tasks as an argument — like `sourcesJar` in the sample — as well as any type of argument accepted by Project.file(java.lang.Object), such as a `File` instance or string file path.
- 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. That's why the above sample does not specify those properties in the artifact configurations.

When you're attaching extra artifacts to a publication, remember that they are *secondary* artifacts that support a *primary* artifact. The metadata that a publication defines — such as dependency information — is associated with that primary artifact only. Thinking about publications in this way should help you determine whether you should be adding custom artifacts to an existing publication, or defining a new publication.

Publishing a custom primary artifact (no component)

If your build produces a primary artifact that isn't supported by a predefined component, then you will need to configure a custom artifact. This isn't much different to adding a custom artifact to an existing publication. There are just a couple of extra considerations:

- You may want to make the artifact available to other projects in the build
- You will need to manually construct the necessary metadata for publishing

Inter-project dependencies have nothing to do with publishing, but both features typically apply to the same set of artifacts in a Gradle project. So how do you tie them together?

You 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 358. Defining a custom artifact for a configuration

build.gradle

```
def rpmFile = file("$buildDir/rpms/my-package.rpm")
def rpmArtifact = artifacts.add('archives', rpmFile) {
    type 'rpm'
    builtBy 'rpm'
}
```

build.gradle.kts

```
val rpmFile = file("$buildDir/rpms/my-package.rpm")
val rpmArtifact = artifacts.add("archives", rpmFile) {
    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 359. 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)
        }
    }
}
```

Now you can publish the RPM as well as depend on it from another project using the `project(path: ':my-project', configuration: 'archives')` syntax.

NOTE There is currently no easy way to define dependency information for a custom artifact.

The `groupId` and `artifactId` properties are specific to Maven publications. See IvyPublication for the relevant Ivy properties.

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 desciptors. 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 360. 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 `publish PubNamePublicationToRepoNameRepository` 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 :generatePomFileForMavenJavaPublication
> Task :compileJava
> Task :processResources
> Task :classes
> Task :jar
> Task :javadoc
> Task :javadocJar
> Task :sourcesJar
> Task :signMavenJavaPublication
> Task :publishMavenJavaPublicationToMavenRepository
> Task :publish

BUILD SUCCESSFUL in 0s
9 actionable tasks: 9 executed
```

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 361. 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 = "$buildDir/repos/external"
        }
        maven {
            name = 'internal'
            url = "$buildDir/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("$buildDir/repos/external")
        }
        maven {
            name = "internal"
            url = uri("$buildDir/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(org.gradle.api.specs.Spec) method. The following sample demonstrates how to implement the constraints we just mentioned:

Example 362. Configuring which artifacts should be published to which repositories

build.gradle

```
tasks.withType(PublishToMavenRepository) {
    onlyIf {
        (repository == publishing.repositories.external &&
            publication == publishing.publications.binary) ||
        (repository == publishing.repositories.internal &&
            publication == publishing.publications.binaryAndSources)
    }
}
tasks.withType(PublishToMavenLocal) {
    onlyIf {
        publication == publishing.publications.binaryAndSources
    }
}
```

build.gradle.kts

```
tasks.withType<PublishToMavenRepository> {
    onlyIf {
        (repository == publishing.repositories["external"] &&
            publication == publishing.publications["binary"]) ||
        (repository == publishing.repositories["internal"] &&
            publication == publishing.publications["binaryAndSources"])
    }
}
tasks.withType<PublishToMavenLocal>() {
    onlyIf {
        publication == publishing.publications["binaryAndSources"]
    }
}
```

Output of `gradle publish`

```
> gradle publish
> Task :generatePomFileForBinaryAndSourcesPublication
> Task :compileJava
> Task :processResources
> Task :classes
> Task :jar
> Task :sourcesJar
> Task :publishBinaryAndSourcesPublicationToExternalRepository SKIPPED
> Task :publishBinaryAndSourcesPublicationToInternalRepository
> Task :generatePomFileForBinaryPublication
> Task :publishBinaryPublicationToExternalRepository
> Task :publishBinaryPublicationToInternalRepository SKIPPED
> Task :publish

BUILD SUCCESSFUL in 0s
8 actionable tasks: 8 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 363. Defining your own shorthand tasks for publishing

build.gradle

```
task 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

```
task("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 364. 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 = "$buildDir/poms/${publicationName}-pom.xml"
}
```

build.gradle.kts

```
tasks.withType<GenerateMavenPom>().all {
    val matcher =
Regex("""generatePomFileFor(\w+)Publication""").matchEntire(name)
    val publicationName = matcher?.let { it.groupValues[1] }
    destination = file("$buildDir/poms/$publicationName-pom.xml")
}
```

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.

Terminology

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.

Component

Any single version of a module.

Components are defined by plugins and provide a simple way to define a publication for publishing. They comprise one or more artifacts as well as the appropriate metadata. For example, the `java` component consists of the production JAR — produced by the `jar` task — and its dependency information.

Configuration

A named collection of dependencies or artifacts.

Gradle's configurations can be somewhat confusing because they apply to both dependencies and artifacts. The main difference is that dependencies are consumed by the project, while artifacts are produced by it. Even then, the artifacts produced by a project are often consumed as dependencies by other projects.

Configurations allow different aspects of the build to work with known subsets of a project's dependencies or artifacts, e.g. the dependencies required for compilation, or the artifacts related to a project's API.

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 nature of that information depends on what type of repository you publish the publication to. In the case of Maven, the information takes the form of a POM.

One thing to bear in mind is that Maven repositories only allow a single *primary* artifact, i.e. one with metadata, but they do allow *secondary* artifacts such as packages of the associated source files and documentation ("-sources" and "-javadoc" JARs in the Java world).

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 365. 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

`generatePomFileForPubNamePublication` — *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*.

`publishPubNamePublicationToRepoNameRepository` — *PublishToMavenRepository*

Publishes the *PubName* publication to the repository named *RepoName*. If you have a repository definition without an explicit name, *RepoName* will be "Maven".

`publishPubNamePublicationToMavenLocal` — *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 366. Customizing the publication identity

build.gradle

```
publishing {
    publications {
        maven(MavenPublication) {
            groupId = 'org.gradle.sample'
            artifactId = 'project1-sample'
            version = '1.1'

            from components.java
        }
    }
}
```

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("maven") {
            groupId = "org.gradle.sample"
            artifactId = "project1-sample"
            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 367. 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'
                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")
                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

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 368. Declaring repositories to publish to

build.gradle

```
publishing {
    repositories {
        maven {
            // change to point to your repo, e.g. http://my.org/repo
            url = "$buildDir/repo"
        }
    }
}
```

build.gradle.kts

```
publishing {
    repositories {
        maven {
            // change to point to your repo, e.g. http://my.org/repo
            url = uri("$buildDir/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 369. Configuring repository URL based on project version

build.gradle

```
publishing {
    repositories {
        maven {
            def releasesRepoUrl = "$buildDir/repos/releases"
            def snapshotsRepoUrl = "$buildDir/repos/snapshots"
            url = version.endsWith('SNAPSHOT') ? snapshotsRepoUrl :
releasesRepoUrl
        }
    }
}
```

build.gradle.kts

```
publishing {
    repositories {
        maven {
            val releasesRepoUrl = "$buildDir/repos/releases"
            val snapshotsRepoUrl = "$buildDir/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 370. Configuring repository URL based on project property

build.gradle

```
publishing {
    repositories {
        maven {
            def releasesRepoUrl = "$buildDir/repos/releases"
            def snapshotsRepoUrl = "$buildDir/repos/snapshots"
            url = project.hasProperty('release') ? releasesRepoUrl :
snapshotsRepoUrl
        }
    }
}
```

build.gradle.kts

```
publishing {
    repositories {
        maven {
            val releasesRepoUrl = "$buildDir/repos/releases"
            val snapshotsRepoUrl = "$buildDir/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.

Complete example

The following example demonstrates how to sign and publish a Java library including sources, Javadoc, and a customized POM:

Example 371. Publishing a Java library

build.gradle

```
plugins {
    id 'java-library'
    id 'maven-publish'
    id 'signing'
}

group = 'com.example'
version = '1.0'

task sourcesJar(type: Jar) {
    from sourceSets.main.allJava
    classifier = 'sources'
}

task javadocJar(type: Jar) {
    from javadoc
    classifier = 'javadoc'
}

publishing {
    publications {
        mavenJava(MavenPublication) {
            artifactId = 'my-library'
            from components.java
            artifact sourcesJar
            artifact javadocJar
            pom {
                name = 'My Library'
                description = 'A concise description of my library'
                url = 'http://www.example.com/library'
                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 = "$buildDir/repos/releases"
            def snapshotsRepoUrl = "$buildDir/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"

task<Jar>("sourcesJar") {
    from(sourceSets.main.get().allJava)
    classifier = "sources"
}

task<Jar>("javadocJar") {
    from(tasks.javadoc)
    classifier = "javadoc"
}
```

```
publishing {
    publications {
        create<MavenPublication>("mavenJava") {
            artifactId = "my-library"
            from(components["java"])
            artifact(tasks["sourcesJar"])
            artifact(tasks["javadocJar"])
            pom {
                name.set("My Library")
                description.set("A concise description of my library")
                url.set("http://www.example.com/library")
                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("$buildDir/repos/releases")
            val snapshotsRepoUrl = uri("$buildDir/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.baseName
            }
        }
    }
}
```

build.gradle.kts

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

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.baseName
                }
            }
        }
    }
}
```

build.gradle.kts

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

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 372. 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 `publish`PubName`PublicationTo`RepoName`Repository` 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 373. 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 374. 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'
            }
        }
    }
}
```

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")
            }
        }
    }
}
```

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 can not 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.

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 375. Declaring repositories to publish to

build.gradle

```
publishing {
    repositories {
        ivy {
            // change to point to your repo, e.g. http://my.org/repo
            url = "$buildDir/repo"
        }
    }
}
```

build.gradle.kts

```
publishing {
    repositories {
        ivy {
            // change to point to your repo, e.g. http://my.org/repo
            url = uri("$buildDir/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 and a configured additional source artifact. The descriptor file is customized to include the project description for each project.

Example 376. Publishing a Java module

build.gradle

```
subprojects {
    apply plugin: 'java'
    apply plugin: 'ivy-publish'

    version = '1.0'
    group = 'org.gradle.sample'

    repositories {
        mavenCentral()
    }
    task sourcesJar(type: Jar) {
        from sourceSets.main.java
        classifier = 'sources'
    }
}

project(':project1') {
    description = 'The first project'

    dependencies {
        implementation 'junit:junit:4.12'
        implementation project(':project2')
    }
}

project(':project2') {
    description = 'The second project'

    dependencies {
        implementation 'commons-collections:commons-collections:3.2.2'
    }
```

```
}

subprojects {
    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
                artifact(sourcesJar) {
                    type = 'sources'
                    conf = 'compile'
                }
                descriptor.description {
                    text = description
                }
            }
        }
    }
}
```

build.gradle.kts

```
subprojects {
    apply(plugin = "java")
    apply(plugin = "ivy-publish")

    version = "1.0"
    group = "org.gradle.sample"

    repositories {
        mavenCentral()
    }
    task<Jar>("sourcesJar") {
        from(project.the<SourceSetContainer>()["main"].java)
        classifier = "sources"
    }
}

project(":project1") {
    description = "The first project"

    dependencies {
        "implementation"("junit:junit:4.12")
        "implementation"(project(":project2"))
```

```
    }
}

project(":project2") {
    description = "The second project"

    dependencies {
        "implementation"("commons-collections:commons-collections:3.2.2")
    }
}

subprojects {
    configure<PublishingExtension>() {
        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"])
                artifact(tasks["sourcesJar"]) {
                    type = "sources"
                    conf = "compile"
                }
                descriptor.description {
                    text.set(description)
                }
            }
        }
    }
}
```

The result is that the following artifacts will be published for each project:

- The Ivy module descriptor file: `ivy-1.0.xml`.
- The primary JAR artifact for the Java component: `project1-1.0.jar`.
- The source JAR artifact that has been explicitly configured: `project1-1.0-source.jar`.

When `project1` is published, the module descriptor (i.e. the `ivy.xml` file) that is produced will look like:

Example: Generated ivy.xml

output-ivy.xml

```
<!-- This file is an example of the Ivy module descriptor that this build will produce
-->
<?xml version="1.0" encoding="UTF-8"?>
<ivy-module version="2.0" xmlns:m="http://ant.apache.org/ivy/maven">
  <info organisation="org.gradle.sample" module="project1" revision="1.0" status=
"integration" publication="«PUBLICATION-TIME-STAMP»">
    <description>The first project</description>
  </info>
  <configurations>
    <conf name="compile" visibility="public"/>
    <conf name="default" visibility="public" extends="compile,runtime"/>
    <conf name="runtime" visibility="public"/>
  </configurations>
  <publications>
    <artifact name="project1" type="sources" ext="jar" conf="compile" m:classifier=
"sources"/>
    <artifact name="project1" type="jar" ext="jar" conf="compile"/>
  </publications>
  <dependencies>
    <dependency org="junit" name="junit" rev="4.12" conf="runtime-&amp;gt;default"/>
    <dependency org="org.gradle.sample" name="project2" rev="1.0" conf="runtime-
&amp;gt;default"/>
  </dependencies>
</ivy-module>
```

TIP Note that «PUBLICATION-TIME-STAMP» in this example Ivy module descriptor will be the timestamp of when the descriptor was generated.

Legacy publishing

NOTE This chapter describes the *original* publishing mechanism available in Gradle 1.0, which has since been superseded by an alternative model. The approach detailed in this chapter — based on Upload tasks — should not be used in new builds. We cover it in order to help users work with and update existing builds that use it.

Introduction

This chapter is about how you declare the outgoing artifacts of your project, and how to work with them (e.g. upload them). We define the artifacts of the projects as the files the project provides to the outside world. This might be a library or a ZIP distribution or any other file. A project can publish as many artifacts as it wants.

Artifacts and configurations

Like dependencies, artifacts are grouped by configurations. In fact, a configuration can contain both artifacts and dependencies at the same time.

For each configuration in your project, Gradle provides the tasks `upload`*`ConfigurationName`* and `build`*`ConfigurationName`* when the base plugin is applied. Execution of these tasks will build or upload the artifacts belonging to the respective configuration.

This listing shows the configurations added by the Java plugin. Two of the configurations are relevant for the usage with artifacts. The `archives` configuration is the standard configuration to assign your artifacts to. The Java plugin automatically assigns the default jar to this configuration. We will talk more about the `runtime` configuration further on. As with dependencies, you can declare as many custom configurations as you like and assign artifacts to them.

Declaring artifacts

Archive task artifacts

You can use an archive task to define an artifact:

Example 377. Defining an artifact using an archive task

build.gradle

```
task myJar(type: Jar)

artifacts {
    archives myJar
}
```

build.gradle.kts

```
val myJar = task<Jar>("myJar")

artifacts {
    add("archives", myJar)
}
```

It is important to note that the custom archives you are creating as part of your build are not automatically assigned to any configuration. You have to explicitly do this assignment.

File artifacts

You can also use a file to define an artifact:

Example 378. Defining an artifact using a file

build.gradle

```
def someFile = file("$buildDir/somefile.txt")

artifacts {
    archives someFile
}
```

build.gradle.kts

```
val someFile = file("$buildDir/somefile.txt")

artifacts {
    add("archives", someFile)
}
```

Gradle will figure out the properties of the artifact based on the name of the file. You can customize these properties:

Example 379. Customizing an artifact

build.gradle

```
task myTask(type:  MyTaskType) {
    destFile = file("$buildDir/somefile.txt")
}

artifacts {
    archives(myTask.destFile) {
        name 'my-artifact'
        type 'text'
        builtBy myTask
    }
}
```

build.gradle.kts

```
val myTask = task<MyTaskType>("myTask") {
    destFile = file("$buildDir/somefile.txt")
}

artifacts {
    add("archives", myTask.destFile!!) {
        name = "my-artifact"
        type = "text"
        builtBy(myTask)
    }
}
```

There is a map-based syntax for defining an artifact using a file. The map must include a `file` entry that defines the file. The map may include other artifact properties:

Example 380. Map syntax for defining an artifact using a file

build.gradle

```
task generate(type:  MyTaskType) {
    destFile = file("$buildDir/somefile.txt")
}

artifacts {
    archives file: generate.destFile, name: 'my-artifact', type: 'text',
builtBy: generate
}
```

build.gradle.kts

```
val generate = task<MyTaskType>("generate") {
    destFile = file("$buildDir/somefile.txt")
}

artifacts {
    add("archives",
        mapOf("file" to generate.destFile, "name" to "my-artifact", "type" to
"text", "builtBy" to generate))
}
```

Publishing artifacts

We have said that there is a specific upload task for each configuration. Before you can do an upload, you have to configure the upload task and define where to publish the artifacts to. The repositories you have defined (as described in Declaring Repositories) are not automatically used for uploading. In fact, some of those repositories only allow downloading artifacts, not uploading. Here is an example of how you can configure the upload task of a configuration:

Example 381. Configuration of the upload task

build.gradle

```
repositories {
    flatDir {
        name "fileRepo"
        dirs "repo"
    }
}

uploadArchives {
    repositories {
        add project.repositories.fileRepo
        ivy {
            credentials {
                username "username"
                password "pw"
            }
            url "http://repo.mycompany.com"
        }
    }
}
```

build.gradle.kts

```
repositories {
    flatDir {
        name = "fileRepo"
        dirs("repo")
    }
}

tasks.getByName<Upload>("uploadArchives") {
    repositories {
        add(project.repositories["fileRepo"])
        ivy {
            credentials {
                username = "username"
                password = "pw"
            }
            url = uri("http://repo.mycompany.com")
        }
    }
}
```

As you can see, you can either use a reference to an existing repository or create a new repository.

If an upload repository is defined with multiple patterns, Gradle must choose a pattern to use for uploading each file. By default, Gradle will upload to the pattern defined by the `url` parameter, combined with the optional `layout` parameter. If no `url` parameter is supplied, then Gradle will use the first defined `artifactPattern` for uploading, or the first defined `ivyPattern` for uploading Ivy files, if this is set.

Uploading to a Maven repository is described in this section.

More about project libraries

If your project is supposed to be used as a library, you need to define what are the artifacts of this library and what are the dependencies of these artifacts. The Java plugin adds a `runtime` configuration for this purpose, with the implicit assumption that the `runtime` dependencies are the dependencies of the artifact you want to publish. Of course this is fully customizable. You can add your own custom configuration or let the existing configurations extend from other configurations. You might have a different group of artifacts which have a different set of dependencies. This mechanism is very powerful and flexible.

If someone wants to use your project as a library, she simply needs to declare which configuration of the dependency to depend on. A Gradle dependency offers the `configuration` property to declare this. If this is not specified, the `default` configuration is used (see Managing Dependency Configurations). Using your project as a library can either happen from within a multi-project build or by retrieving your project from a repository. In the latter case, an `ivy.xml` descriptor in the repository is supposed to contain all the necessary information. If you work with Maven repositories you don't have the flexibility as described above. For how to publish to a Maven repository, see the section Uploading to Maven repositories.

Maven Plugin

NOTE

This chapter describes deploying artifacts to Maven repositories using the *original* publishing mechanism available in Gradle 1.0: in Gradle 1.3 a new mechanism for publishing was introduced. This new mechanism introduces some new concepts and features that make Gradle publishing even more powerful and is now the preferred option for publishing artifacts.

You can read about the new publishing plugins in Publishing Ivy and Publishing Maven.

The Maven plugin adds support for deploying artifacts to Maven repositories.

Usage

To use the Maven plugin, include the following in your build script:

Example 382. Using the Maven plugin

build.gradle

```
plugins {
    id 'maven'
}
```

build.gradle.kts

```
plugins {
    maven
}
```

Tasks

The Maven plugin defines the following tasks:

`install` — *Upload*

Depends on: All tasks that build the associated archives.

Installs the associated artifacts to the local Maven cache, including Maven metadata generation. By default the install task is associated with the `archives` configuration. This configuration has by default only the default jar as an element. To learn more about installing to the local repository, see Installing to the local repository

Dependency management

The Maven plugin does not define any dependency configurations.

Convention properties

The Maven plugin defines the following convention properties:

`mavenPomDir` — `File`

The directory where the generated POMs are written to. *Default value*: `${project.buildDir}/poms`

`conf2ScopeMappings` — `Conf2ScopeMappingContainer`

Instructions for mapping Gradle configurations to Maven scopes. See Dependency mapping.

These properties are provided by a MavenPluginConvention convention object.

Convention methods

The maven plugin provides a factory method for creating a POM. This is useful if you need a POM without the context of uploading to a Maven repo.

Example 383. Creating a standalone pom.

build.gradle

```
task writeNewPom {
    doLast {
        pom {
            project {
                inceptionYear '2008'
                licenses {
                    license {
                        name 'The Apache Software License, Version 2.0'
                        url 'http://www.apache.org/licenses/LICENSE-2.0.txt'
                        distribution 'repo'
                    }
                }
            }
        }.writeTo("$buildDir/newpom.xml")
    }
}
```

build.gradle.kts

```
task("writeNewPom") {
    doLast {
        maven.pom {
            withGroovyBuilder {
                "project" {
                    setProperty("inceptionYear", "2008")
                    "licenses" {
                        "license" {
                            setProperty("name", "The Apache Software License,
Version 2.0")
                            setProperty("url",
"http://www.apache.org/licenses/LICENSE-2.0.txt")
                            setProperty("distribution", "repo")
                        }
                    }
                }
            }
        }.writeTo("$buildDir/newpom.xml")
    }
}
```

Amongst other things, Gradle supports the same builder syntax as polyglot Maven. To learn more

about the Gradle Maven POM object, see MavenPom. See also: MavenPluginConvention

Interacting with Maven repositories

Introduction

With Gradle you can deploy to remote Maven repositories or install to your local Maven repository. This includes all Maven metadata manipulation and works also for Maven snapshots. In fact, Gradle's deployment is 100 percent Maven compatible as we use the native Maven Ant tasks under the hood.

Deploying to a Maven repository is only half the fun if you don't have a POM. Fortunately Gradle can generate this POM for you using the dependency information it has.

Deploying to a Maven repository

Let's assume your project produces just the default jar file. Now you want to deploy this jar file to a remote Maven repository.

Example 384. Upload of file to remote Maven repository

build.gradle

```
plugins {
    id 'maven'
}

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
        }
    }
}
```

build.gradle.kts

```
plugins {
    maven
}

tasks.getByName<Upload>("uploadArchives") {
    repositories.withGroovyBuilder {
        "mavenDeployer" {
            "repository"("url" to "file://localhost/tmp/myRepo/")
        }
    }
}
```

That is all. Calling the `uploadArchives` task will generate the POM and deploys the artifact and the POM to the specified repository.

There is more work to do if you need support for protocols other than `file`. In this case the native Maven code we delegate to needs additional libraries. Which libraries are needed depends on what protocol you plan to use. The available protocols and the corresponding libraries are listed in Protocol JARs for Maven deployment (those libraries have transitive dependencies which have transitive dependencies). [9: It is planned for a future release to provide out-of-the-box support for this] For example, to use the ssh protocol you can do:

Example 385. Upload of file via SSH

build.gradle

```
configurations {
    deployerJars
}

repositories {
    mavenCentral()
}

dependencies {
    deployerJars "org.apache.maven.wagon:wagon-ssh:2.2"
}

uploadArchives {
    repositories.mavenDeployer {
        configuration = configurations.deployerJars
        repository(url: "scp://repos.mycompany.com/releases") {
            authentication(userName: "me", password: "myPassword")
        }
    }
}
```

build.gradle.kts

```
val deployerJars by configurations.creating

repositories {
    mavenCentral()
}

dependencies {
    deployerJars("org.apache.maven.wagon:wagon-ssh:2.2")
}

tasks.getByName<Upload>("uploadArchives") {
    repositories.withGroovyBuilder {
        "mavenDeployer" {
            setProperty("configuration", deployerJars)
            "repository"("url" to "scp://repos.mycompany.com/releases") {
                "authentication"("userName" to "me", "password" to
"myPassword")
            }
        }
    }
}
```

There are many configuration options for the Maven deployer. The configuration is done via a Groovy builder. All the elements of this tree are Java beans. To configure the simple attributes you pass a map to the bean elements. To add bean elements to its parent, you use a closure. In the example above *repository* and *authentication* are such bean elements. Configuration elements of Maven deployer lists the available bean elements and a link to the Javadoc of the corresponding class. In the Javadoc you can see the possible attributes you can set for a particular element.

In Maven you can define repositories and optionally snapshot repositories. If no snapshot repository is defined, releases and snapshots are both deployed to the `repository` element. Otherwise snapshots are deployed to the `snapshotRepository` element.

Table 10. Protocol jars for Maven deployment

Protocol	Library
http	org.apache.maven.wagon:wagon-http:2.2
ssh	org.apache.maven.wagon:wagon-ssh:2.2
ssh-external	org.apache.maven.wagon:wagon-ssh-external:2.2
ftp	org.apache.maven.wagon:wagon-ftp:2.2
webdav	org.apache.maven.wagon:wagon-webdav:1.0-beta-2
file	-

Table 11. Configuration elements of the MavenDeployer

Element	Javadoc
root	MavenDeployer
repository	org.apache.maven.artifact.ant.RemoteRepository
authentication	org.apache.maven.artifact.ant.Authentication
releases	org.apache.maven.artifact.ant.RepositoryPolicy
snapshots	org.apache.maven.artifact.ant.RepositoryPolicy
proxy	org.apache.maven.artifact.ant.Proxy
snapshotRepository	org.apache.maven.artifact.ant.RemoteRepository

Installing to the local repository

The Maven plugin adds an `install` task to your project. This task depends on all the archives task of the `archives` configuration. It installs those archives to your local Maven repository. If the default location for the local repository is redefined in a Maven `settings.xml`, this is considered by this task.

Maven POM generation

When deploying an artifact to a Maven repository, Gradle automatically generates a POM for it. The `groupId`, `artifactId`, `version` and `packaging` elements used for the POM default to the values shown in the table below. The `dependency` elements are created from the project's dependency declarations.

Table 12. Default Values for Maven POM generation

Maven Element	Default Value
groupId	project.group
artifactId	uploadTask.repositories.mavenDeployer.pom.artifactId (if set) or archiveTask.baseName.
version	project.version
packaging	archiveTask.extension

Here, `uploadTask` and `archiveTask` refer to the tasks used for uploading and generating the archive, respectively (for example `uploadArchives` and `jar`). `archiveTask.baseName` defaults to `project.archivesBaseName` which in turn defaults to `project.name`.

NOTE When you set the "`archiveTask.baseName`" property to a value other than the default, you'll also have to set `uploadTask.repositories.mavenDeployer.pom.artifactId` to the same value. Otherwise, the project at hand may be referenced with the wrong artifact ID from generated POMs for other projects in the same build.

Generated POMs can be found in `<buildDir>/poms`. They can be further customized via the MavenPom API. For example, you might want the artifact deployed to the Maven repository to have a different version or name than the artifact generated by Gradle. To customize these you can do:

Example 386. Customization of pom

build.gradle

```
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
            pom.version = '1.0Maven'
            pom.artifactId = 'myMavenName'
        }
    }
}
```

build.gradle.kts

```
tasks.getByName<Upload>("uploadArchives") {
    repositories.withGroovyBuilder {
        "mavenDeployer" {
            "repository"("url" to "file://localhost/tmp/myRepo/")
            "pom" {
                setProperty("version", "1.0Maven")
                setProperty("artifactId", "myMavenName")
            }
        }
    }
}
```

To add additional content to the POM, the `pom.project` builder can be used. With this builder, any element listed in the Maven POM reference can be added.

Example 387. Builder style customization of pom

build.gradle

```
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
            pom.project {
                licenses {
                    license {
                        name 'The Apache Software License, Version 2.0'
                        url 'http://www.apache.org/licenses/LICENSE-2.0.txt'
                        distribution 'repo'
                    }
                }
            }
        }
    }
}
```

build.gradle.kts

```
tasks.getByName<Upload>("uploadArchives") {
    repositories.withGroovyBuilder {
        "mavenDeployer" {
            "repository"("url" to "file://localhost/tmp/myRepo/")
            "pom" {
                "project" {
                    "licenses" {
                        "license" {
                            setProperty("name", "The Apache Software License,
Version 2.0")
                            setProperty("url",
"http://www.apache.org/licenses/LICENSE-2.0.txt")
                            setProperty("distribution", "repo")
                        }
                    }
                }
            }
        }
    }
}
```

Note: `groupId`, `artifactId`, `version`, and `packaging` should always be set directly on the `pom` object.

Example 388. Modifying auto-generated content

build.gradle

```
def installer = install.repositories.mavenInstaller
def deployer = uploadArchives.repositories.mavenDeployer

[installer, deployer]*.pom*.whenConfigured {pom ->
    pom.dependencies.find {dep -> dep.groupId == 'group3' && dep.artifactId
== 'runtime' }.optional = true
}
```

build.gradle.kts

```
val uploadArchives by tasks.getting(Upload::class)
val installer = tasks.install.get().repositories.withGroovyBuilder {
getProperty("mavenInstaller") as MavenResolver }
val deployer = uploadArchives.repositories.withGroovyBuilder {
getProperty("mavenDeployer") as MavenResolver }

listOf(installer, deployer).forEach {
    it.pom.whenConfigured {
        dependencies.firstOrNull { dep ->
            dep!!.withGroovyBuilder {
                getProperty("groupId") == "group3" &&
getProperty("artifactId") == "runtime"
            }
        }?.withGroovyBuilder {
            setProperty("optional", true)
        }
    }
}
```

If you have more than one artifact to publish, things work a little bit differently. See Multiple artifacts per project.

To customize the settings for the Maven installer (see Installing to the local repository), you can do:

Example 389. Customization of Maven installer

build.gradle

```
install {
    repositories.mavenInstaller {
        pom.version = '1.0Maven'
        pom.artifactId = 'myName'
    }
}
```

build.gradle.kts

```
tasks.install {
    repositories.withGroovyBuilder {
        "mavenInstaller" {
            "pom" {
                setProperty("version", "1.0Maven")
                setProperty("artifactId", "myName")
            }
        }
    }
}
```

Multiple artifacts per project

Maven can only deal with one artifact per project. This is reflected in the structure of the Maven POM. We think there are many situations where it makes sense to have more than one artifact per project. In such a case you need to generate multiple POMs. In such a case you have to explicitly declare each artifact you want to publish to a Maven repository. The MavenDeployer and the MavenInstaller both provide an API for this:

Example 390. Generation of multiple poms

build.gradle

```
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
            addFilter('api') {artifact, file ->
                artifact.name == 'api'
            }
            addFilter('service') {artifact, file ->
                artifact.name == 'service'
            }
            pom('api').version = 'mySpecialMavenVersion'
        }
    }
}
```

build.gradle.kts

```
tasks.getByName<Upload>("uploadArchives") {
    repositories.withGroovyBuilder {
        "mavenDeployer" {
            "repository"("url" to "file://localhost/tmp/myRepo/")
            "addFilter"("api") {
                getProperty("artifact").withGroovyBuilder {
setProperty("name", "api") }
            }
            "addFilter"("service") {
                getProperty("artifact").withGroovyBuilder {
setProperty("name", "service") }
            }
            "pom"("api")?.withGroovyBuilder { setProperty("version",
"mySpecialMavenVersion") }
        }
    }
}
```

You need to declare a filter for each artifact you want to publish. This filter defines a boolean expression for which Gradle artifact it accepts. Each filter has a POM associated with it which you can configure. To learn more about this have a look at PomFilterContainer and its associated classes.

Dependency mapping

The Maven plugin configures the default mapping between the Gradle configurations added by the Java and War plugin and the Maven scopes. Most of the time you don't need to touch this and you can safely skip this section. The mapping works like the following. You can map a configuration to one and only one scope. Different configurations can be mapped to one or different scopes. You can also assign a priority to a particular configuration-to-scope mapping. Have a look at Conf2ScopeMappingContainer to learn more. To access the mapping configuration you can say:

Example 391. Accessing a mapping configuration

build.gradle

```
task mappings {
    doLast {
        println conf2ScopeMappings.mappings
    }
}
```

build.gradle.kts

```
task("mappings") {
    doLast {
        println(maven.conf2ScopeMappings.mappings)
    }
}
```

Gradle exclude rules are converted to Maven excludes if possible. Such a conversion is possible if in the Gradle exclude rule the group as well as the module name is specified (as Maven needs both in contrast to Ivy). Per-configuration excludes are also included in the Maven POM, if they are convertible.

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 392. 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 source the information however you need to and set the project properties manually.

build.gradle

```
gradle.taskGraph.whenReady { taskGraph ->
    if (taskGraph.allTasks.any { it instanceof Sign }) {
        // Use Java's console to read from the console (no good for
        // a CI environment)
        def console = System.console()
        console.printf "\n\nWe have to sign some things in this build." +
                       "\n\nPlease enter your signing details.\n\n"

        def id = console.readLine("PGP Key Id: ")
        def file = console.readLine("PGP Secret Key Ring File (absolute path): ")
        def password = console.readPassword("PGP Private Key Password: ")

        allprojects {
            ext."signing.keyId" = id
            ext."signing.secretKeyRingFile" = file
            ext."signing.password" = password
        }

        console.printf "\nThanks.\n\n"
    }
}
```

build.gradle.kts

```
gradle.taskGraph.whenReady {
    if (allTasks.any { it is Sign }) {
        // Use Java's console to read from the console (no good for
        // a CI environment)
        val console = System.console()
        console.printf("\n\nWe have to sign some things in this build." +
                       "\n\nPlease enter your signing details.\n\n")

        val id = console.readLine("PGP Key Id: ")
        val file = console.readLine("PGP Secret Key Ring File (absolute path): ")
        val password = console.readPassword("PGP Private Key Password: ")

        allprojects {
            extra["signing.keyId"] = id
            extra["signing.secretKeyRingFile"] = file
            extra["signing.password"] = password
        }

        console.printf("\nThanks.\n\n")
    }
}
```

Note that the presence of a null value for any these three properties will cause an exception.

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 393. 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 `gpg2` (on Windows: `gpg2.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 394. 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 :generatePomFileForMavenJavaPublication
> Task :compileJava
> Task :processResources
> Task :classes
> Task :jar
> Task :javadoc
> Task :javadocJar
> Task :sourcesJar
> Task :signMavenJavaPublication

BUILD SUCCESSFUL in 0s
8 actionable tasks: 8 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 395. 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 Tasks

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 396. Signing a task

build.gradle

```
task stuffZip(type: Zip) {
    baseName = 'stuff'
    from 'src/stuff'
}

signing {
    sign stuffZip
}
```

build.gradle.kts

```
task<Zip>("stuffZip") {
    baseName = "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 397. Specifying when signing is required

build.gradle

```
version = '1.0-SNAPSHOT'
ext.isReleaseVersion = !version.endsWith("SNAPSHOT")

signing {
    required { isReleaseVersion && gradle.taskGraph.hasTask("uploadArchives")
}
    sign configurations.archives
}
```

build.gradle.kts

```
import java.util.concurrent.Callable

version = "1.0-SNAPSHOT"
extra["isReleaseVersion"] = !version.toString().endsWith("SNAPSHOT")

signing {
    setRequired(Callable {
        (project.extra["isReleaseVersion"] as Boolean) &&
gradle.taskGraph.hasTask("uploadArchives")
    })
    sign(configurations.archives.get())
}
```

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 398. Specifying when signing is skipped

build.gradle

```
tasks.withType(Sign) {
    onlyIf { isReleaseVersion }
}
```

build.gradle.kts

```
tasks.withType<Sign> {
    onlyIf { 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. This means that if you want to upload your signatures to your distribution repository along with the artifacts you simply execute the `uploadArchives` task.

Signing POM files

NOTE This section covers signing POM files for the *original* publishing mechanism available in Gradle 1.0. The POM file generated by the *new* Maven publishing support provided by the Maven Publishing plugin is automatically signed if the corresponding publication is specified to be signed.

When deploying signatures for your artifacts to a Maven repository, you will also want to sign the published POM file. The Signing Plugin adds a `signing.signPom()` (see SigningExtension.signPom(org.gradle.api.artifacts.maven.MavenDeployment, groovy.lang.Closure)) method that can be used in the `beforeDeployment()` block in your upload task configuration.

Example 399. Signing a POM for deployment

build.gradle

```
uploadArchives {
    repositories {
        mavenDeployer {
            beforeDeployment { MavenDeployment deployment -> signing.signPom
(deployment) }
        }
    }
}
```

build.gradle.kts

```
tasks.getByName<Upload>("uploadArchives") {
    repositories {
        withConvention(MavenRepositoryHandlerConvention::class) {
            mavenDeployer {
                beforeDeployment { signing.signPom(this) }
            }
        }
    }
}
```

When signing is not required and the POM cannot be signed due to insufficient configuration (i.e. no credentials for signing) then the `signPom()` method will silently do nothing.

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 400. 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 401. 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 402. Configuring the main distribution

build.gradle

```
distributions {
    main {
        baseName = 'someName'
        contents {
            from 'src/readme'
        }
    }
}
```

build.gradle.kts

```
distributions {
    main {
        baseName = "someName"
        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 `baseName` property has 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, or via the *original* publishing mechanism using the `uploadArchives` task.

Using the Ivy/Maven Publish Plugins

To publish a distribution to an Ivy repository with the Ivy Publish Plugin, simply 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 403. 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 404. 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.get())
            artifact(tasks["customDistTar"])
        }
    }
}
```

Using the `uploadArchives` task

The Distribution Plugin adds the distribution archives as default publishing artifact candidates. With the Maven Plugin applied, the distribution ZIP file will be published when running `uploadArchives` if no other default artifact is configured.

Example 405. Publishing the distribution ZIP with the Maven Plugin

build.gradle

```
plugins {
    id 'maven'
}

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://some/repo")
        }
    }
}
```

build.gradle.kts

```
plugins {
    maven
}

tasks.getByName<Upload>("uploadArchives") {
    repositories.withGroovyBuilder {
        "mavenDeployer" {
            "repository"("url" to "file://some/repo")
        }
    }
}
```

Native Projects

Building native software

NOTE	Support for building native software is currently incubating. Please be aware that the DSL, APIs and other configuration may change in later Gradle versions.

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.
Windows	GCC with Cygwin 32	Windows XP and later.
Windows	GCC with MinGW	Windows XP and later. Mingw-w64 is currently not supported.

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	
Windows	GCC with Cygwin 64	Windows XP and later.
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.

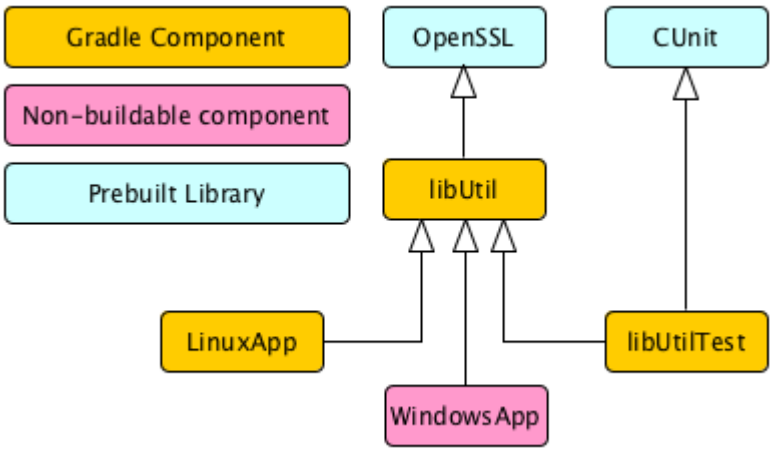


Figure 25. Dependent Components Example

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

```
apply plugin: "c"
apply plugin: 'cunit-test-suite'

model {
    flavors {
        passing
        failing
    }
    platforms {
        x86 {
            architecture "x86"
        }
    }
    components {
        operators(NativeLibrarySpec) {
            targetPlatform "x86"
        }
    }
    testSuites {
        operatorsTest(CUnitTestSuiteSpec) {
            testing $.components.operators
        }
    }
}
```

NOTE	The code for this example can be found at `samples/native-binaries/cunit` in the '-all' distribution of Gradle.

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
------------------------------------------------------------

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
------------------------------------------------------------

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
------------------------------------------------------------

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
------------------------------------------------------------

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

```
apply plugin: "cpp"
// You don't need to apply the plugin below if you're already using CUnit or
GoogleTest support
apply plugin: TestingModelBasePlugin

task 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
            }
        }
    }
}
```

NOTE	The code for this example can be found at `samples/native-binaries/custom-check` in the '-all' distribution of Gradle.

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
------------------------------------------------------------

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 (eg '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

```
apply plugin: '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

```
apply plugin: '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

```
apply plugin: '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

```
apply plugin: '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

```
apply plugin: '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: eg 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

```
apply plugin: '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

build.gradle

```
project(":lib") {
    apply plugin: "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"
                }
            }
        }
    }
}

project(":exe") {
    apply plugin: "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.args '/Zi'
                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
Clang	x86, x86_64
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

```
apply plugin: "c"
apply plugin: 'cunit-test-suite'

model {
    flavors {
        passing
        failing
    }
    platforms {
        x86 {
            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 code for this example can be found at `samples/native-binaries/cunit` in the '-all' distribution of Gradle.

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 code for this example can be found at `samples/native-binaries/google-test` in the '-all' distribution of Gradle.

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.

Software model concepts

CAUTION Rule based configuration will be deprecated. New plugins should not use this concept.

The software model describes how a piece of software is built and how the components of the software relate to each other. The software model is organized around some key concepts:

- A *component* is a general concept that represents some logical piece of software. Examples of components are a command-line application, a web application or a library. A component is often composed of other components. Most Gradle builds will produce at least one component.
- A *library* is a reusable component that is linked into or combined into some other component. In the Java ecosystem, a library is often built as a Jar file, and then later bundled into an application of some kind. In the native ecosystem, a library may be built as a shared library or static library, or both.
- A *source set* represents a logical group of source files. Most components are built from source sets of various languages. Some source sets contain source that is written by hand, and some source sets may contain source that is generated from something else.
- A *binary* represents some output that is built for a component. A component may produce multiple different output binaries. For example, for a C++ library, both a shared library and a static library binary may be produced. Each binary is initially configured to be built from the component sources, but additional source sets can be added to specific binary variants.
- A *variant* represents some mutually exclusive binary of a component. A library, for example, might target Java 7 and Java 8, effectively producing two distinct binaries: a Java 7 Jar and a Java 8 Jar. These are different variants of the library.
- The *API* of a library represents the artifacts and dependencies that are required to compile against that library. The API typically consists of a binary together with a set of dependencies.

Rule based model configuration

CAUTION Rule based configuration will be deprecated. New plugins should not use this concept.

Rule based model configuration enables *configuration logic to itself have dependencies* on other elements of configuration, and to make use of the resolved states of those other elements of configuration while performing its own configuration.

Background

In a nutshell, the Software Model is a very declarative way to describe how a piece of software is built and the other components it needs as dependencies in the process. It also provides a new, rule-based engine for configuring a Gradle build. When we started to implement the software model we set ourselves the following goals:

- Improve configuration and execution time performance.
- Make customizations of builds with complex tool chains easier.
- Provide a richer, more standardized way to model different software ecosystems.

Gradle drastically improved configuration performance through other measures. There is no longer any need for a drastic, incompatible change in how Gradle builds are configured. Gradle's support for building native software and Play Framework applications still use the configuration model.

Basic Concepts

The "model space"

The term "model space" is used to refer to the formal model, which can be read and modified by rules.

A counterpart to the model space is the "project space", which should be familiar to readers. The "project space" is a graph of objects (e.g `project.repositories`, `project.tasks` etc.) having a Project as its root. A build script is effectively adding and configuring objects of this graph. For the most part, the "project space" is opaque to Gradle. It is an arbitrary graph of objects that Gradle only partially understands.

Each project also has its own model space, which is distinct from the project space. A key characteristic of the "model space" is that Gradle knows much more about it (which is knowledge that can be put to good use). The objects in the model space are "managed", to a greater extent than objects in the project space. The origin, structure, state, collaborators and relationships of objects in the model space are first class constructs. This is effectively the characteristic that functionally distinguishes the model space from the project space: the objects of the model space are defined in ways that Gradle can understand them intimately, as opposed to an object that is the result of running relatively opaque code. A "rule" is effectively a building block of this definition.

The model space will eventually replace the project space, becoming the only "space".

Rules

The model space is defined by "rules". A rule is just a function (in the abstract sense) that either produces a model element, or acts upon a model element. Every rule has a single subject and zero or more inputs. Only the subject can be changed by a rule, while the inputs are effectively immutable.

Gradle guarantees that all inputs are fully "realized" before the rule executes. The process of "realizing" a model element is effectively executing all the rules for which it is the subject, transitioning it to its final state. There is a strong analogy here to Gradle's task graph and task

execution model. Just as tasks depend on each other and Gradle ensures that dependencies are satisfied before executing a task, rules effectively depend on each other (i.e. a rule depends on all rules whose subject is one of the inputs) and Gradle ensures that all dependencies are satisfied before executing the rule.

Model elements are very often defined in terms of other model elements. For example, a compile task's configuration can be defined in terms of the configuration of the source set that it is compiling. In this scenario, the compile task would be the subject of a rule and the source set an input. Such a rule could configure the task subject based on the source set input without concern for how it was configured, who it was configured by or when the configuration was specified.

There are several ways to declare rules, and in several forms.

Rule sources

One way to define rules is via a RuleSource subclass. If an object extends RuleSource and contains any methods annotated by '@Mutate', then each such method defines a rule. For each such method, the first argument is the subject, and zero or more subsequent arguments may follow and are inputs of the rule.

Example: applying a rule source plugin

build.gradle

```
@Managed
interface Person {
  void setFirstName(String name)
  String getFirstName()

  void setLastName(String name)
  String getLastName()
}

class PersonRules extends RuleSource {
  @Model void person(Person p) {}

  //Create a rule that modifies a Person and takes no other inputs
  @Mutate void setFirstName(Person p) {
    p.firstName = "John"
  }

  //Create a rule that modifies a ModelMap<Task> and takes as input a Person
  @Mutate void createHelloTask(ModelMap<Task> tasks, Person p) {
    tasks.create("hello") {
      doLast {
        println "Hello $p.firstName $p.lastName!"
      }
    }
  }
}

apply plugin: PersonRules
```

Output of `gradle hello`

```
> gradle hello

> Task :hello
Hello John Smith!

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Each of the different methods of the rule source are discrete, independent rules. Their order, or the fact that they belong to the same class, do not affect their behavior.

Example: a model creation rule

build.gradle

```
@Model void person(Person p) {}
```

This rule declares that there is a model element at path `"person"` (defined by the method name), of type `Person`. This is the form of the Model type rule for Managed types. Here, the person object is the rule subject. The method could potentially have a body, that mutated the person instance. It could also potentially have more parameters, which would be the rule inputs.

Example: a model mutation rule

build.gradle

```
//Create a rule that modifies a Person and takes no other inputs
@Mutate void setFirstName(Person p) {
  p.firstName = "John"
}
```

This Mutate rule mutates the person object. The first parameter to the method is the subject. Here, a by-type reference is used as no Path annotation is present on the parameter. It could also potentially have more parameters, that would be the rule inputs.

Example: creating a task

build.gradle

```
//Create a rule that modifies a ModelMap<Task> and takes as input a Person
@Mutate void createHelloTask(ModelMap<Task> tasks, Person p) {
  tasks.create("hello") {
    doLast {
      println "Hello $p.firstName $p.lastName!"
    }
  }
}
```

This Mutate rule effectively adds a task, by mutating the tasks collection. The subject here is the `"tasks"` node, which is available as a ModelMap of Task. The only input is our person element. As the person is being used as an input here, it will have been realised before executing this rule. That is, the task container effectively *depends on* the person element. If there are other configuration rules for the person element, potentially specified in a build script or other plugin, they will also be guaranteed to have been executed.

As `Person` is a Managed type in this example, any attempt to modify the person parameter in this method would result in an exception being thrown. Managed objects enforce immutability at the appropriate point in their lifecycle.

Rule source plugins can be packaged and distributed in the same manner as other types of plugins (see Custom Plugins). They also may be applied in the same manner (to project objects) as Plugin implementations (i.e. via Project.apply(java.util.Map)).

Please see the documentation for RuleSource for more information on constraints on how rule sources must be implemented and for more types of rules.

Advanced Concepts

Model paths

A model path identifies the location of an element relative to the root of its model space. A common representation is a period-delimited set of names. For example, the model path `"tasks"` is the path to the element that is the task container. Assuming a task whose name is `hello`, the path `"tasks.hello"` is the path to this task.

Managed model elements

Currently, any kind of Java object can be part of the model space. However, there is a difference between “managed” and “unmanaged” objects.

A “managed” object is transparent and enforces immutability once realized. Being transparent means that its structure is understood by the rule infrastructure and as such each of its properties are also individual elements in the model space.

An “unmanaged” object is opaque to the model space and does not enforce immutability. Over time, more mechanisms will be available for defining managed model elements culminating in all model elements being managed in some way.

Managed models can be defined by attaching the `@Managed` annotation to an interface:

Example: a managed type

build.gradle

```
@Managed
interface Person {
  void setFirstName(String name)
  String getFirstName()

  void setLastName(String name)
  String getLastName()
}
```

By defining a getter/setter pair, you are effectively declaring a managed property. A managed property is a property for which Gradle will enforce semantics such as immutability when a node of the model is not the subject of a rule. Therefore, this example declares properties named *firstName* and *lastName* on the managed type *Person*. These properties will only be writable when the view is mutable, that is to say when the *Person* is the subject of a `Rule` (see below the explanation for rules).

Managed properties can be of any scalar type. In addition, properties can also be of any type which is itself managed:

Property type	Nullable	Example
`String`	Yes	[source,groovy,indent=0] ---- void setFirstName(String name) String getFirstName() ----
`File`	Yes	[source,groovy,indent=0] ---- void setHomeDirectory(File homeDir) File getHomeDirectory() ----
`Integer`, `Boolean`, `Byte`, `Short`, `Float`, `Long`, `Double`	Yes	[source,groovy,indent=0] ---- void setId(Long id) Long getId() ----
`int`, `boolean`, `byte`, `short`, `float`, `long`, `double`	No	[source,groovy,indent=0] ---- void setEmployed(boolean isEmployed) boolean isEmployed() ---- ''' [source,groovy,indent=0] ---- void setAge(int age) int getAge() ----
Another _managed_ type.	Only if read/write	[source,groovy,indent=0] ---- void setMother(Person mother) Person getMother() ----
An _enumeration_ type.	Yes	[source,groovy,indent=0] ---- void setMaritalStatus(MaritalStatus status) MaritalStatus getMaritalStatus() ----
A `ManagedSet`. A managed set supports the creation of new named model elements, but not their removal.	Only if read/write	[source,groovy,indent=0] ---- ModelSet<Person> getChildren() ----
A `Set` or `List` of scalar types. All classic operations on collections are supported: add, remove, clear...	Only if read/write	[source,groovy,indent=0] ---- void setUserGroups(List<String> groups) List<String> getUserGroups() ----

If the type of a property is itself a managed type, it is possible to declare only a getter, in which case you are declaring a read-only property. A read-only property will be instantiated by Gradle, and cannot be replaced with another object of the same type (for example calling a setter). However, the properties of that property can potentially be changed, if, and only if, the property is the subject of a rule. If it's not the case, the property is immutable, like any classic read/write managed property, and properties of the property cannot be changed at all.

Managed types can be defined out of interfaces or abstract classes and are usually defined in plugins, which are written either in Java or Groovy. Please see the Managed annotation for more information on creating managed model objects.

Model element types

There are particular types (language types) supported by the model space and can be generalised as follows:

Table 13. Type definitions

Type	Definition
Scalar	A scalar type is one of the following: * a primitive type (e.g. `int`) or its boxed type (e.g `Integer`) * a `BigInteger` or `BigDecimal` * a `String` * a `File` * an enumeration type
Scalar Collection	A java.util.List or java.util.Set containing one of the scalar types
Managed type	Any class which is a valid managed model (i.e.annotated with @link:{javadocPath}/org/gradle/model/Managed.html[Managed])
Managed collection	A link:{javadocPath}/org/gradle/model/ModelMap.html[ModelMap] or link:{javadocPath}/org/gradle/model/ModelSet.html[ModelSet]

There are various contexts in which these types can be used:

Table 14. Model type support

Context	Supported types
Creating top level model elements	* Any managed type * link:{javadocPath}/org/gradle/language/base/FunctionalSourceSet.html[FunctionalSourceSet] (when the link:{javadocPath}/org/gradle/language/base/plugins/LanguageBasePlugin.html[LanguageBasePlugin] plugin has been applied) * Subtypes of link:{javadocPath}/org/gradle/language/base/LanguageSourceSet.html[LanguageSourceSet] which have been registered via link:{javadocPath}/org/gradle/platform/base/ComponentType.html[ComponentType]
Properties of managed model elements	The properties (attributes) of a managed model elements may be one or more of the following: * A managed type * A type which is annotated with @link:{javadocPath}/org/gradle/model/Unmanaged.html[Unmanaged] * A Scalar Collection * A Managed collection containing managed types * A Managed collection containing link:{javadocPath}/org/gradle/language/base/FunctionalSourceSet.html[FunctionalSourceSet]'s (when the link:{javadocPath}/org/gradle/language/base/plugins/LanguageBasePlugin.html[LanguageBasePlugin] plugin has been applied) * Subtypes of link:{javadocPath}/org/gradle/language/base/LanguageSourceSet.html[LanguageSourceSet] which have been registered via link:{javadocPath}/org/gradle/platform/base/ComponentType.html[ComponentType]

Language source sets

FunctionalSourceSets and subtypes of LanguageSourceSet (which have been registered via ComponentType) can be added to the model space via rules or via the model DSL.

Example: Strongly modelling sources sets

build.gradle

```
apply plugin: 'java-lang'

//Creating LanguageSourceSets via rules
class LanguageSourceSetRules extends RuleSource {
    @Model
    void mySourceSet(JavaSourceSet javaSource) {
        javaSource.source.srcDir("src/main/my")
    }
}
apply plugin: LanguageSourceSetRules

//Creating LanguageSourceSets via the model DSL
model {
    another(JavaSourceSet) {
        source {
            srcDir "src/main/another"
        }
    }
}

//Using FunctionalSourceSets
@Managed
interface SourceBundle {
    FunctionalSourceSet getFreeSources()
    FunctionalSourceSet getPaidSources()
}
model {
    sourceBundle(SourceBundle) {
        freeSources.create("main", JavaSourceSet)
        freeSources.create("resources", JvmResourceSet)
        paidSources.create("main", JavaSourceSet)
        paidSources.create("resources", JvmResourceSet)
    }
}
```

Output of `gradle help`

```
> gradle help

> Task :help
```

NOTE	The code for this example can be found at `samples/modelRules/language-support` in the '-all' distribution of Gradle.

References, binding and scopes

As previously mentioned, a rule has a subject and zero or more inputs. The rule's subject and inputs are declared as "references" and are "bound" to model elements before execution by Gradle. Each rule must effectively forward declare the subject and inputs as references. Precisely how this is done depends on the form of the rule. For example, the rules provided by a RuleSource declare references as method parameters.

A reference is either "by-path" or "by-type".

A "by-type" reference identifies a particular model element by its type. For example, a reference to the TaskContainer effectively identifies the `"tasks"` element in the project model space. The model space is not exhaustively searched for candidates for by-type binding; rather, a rule is given a scope (discussed later) that determines the search space for a by-type binding.

A "by-path" reference identifies a particular model element by its path in model space. By-path references are always relative to the rule scope; there is currently no way to path "out" of the scope. All by-path references also have an associated type, but this does not influence what the reference binds to. The element identified by the path must however by type compatible with the reference, or a fatal "binding failure" will occur.

Binding scope

Rules are bound within a "scope", which determines how references bind. Most rules are bound at the project scope (i.e. the root of the model graph for the project). However, rules can be scoped to a node within the graph. The ModelMap.named(java.lang.String, java.lang.Class) method is an example of a mechanism for applying scoped rules. Rules declared in the build script using the `model {}` block, or via a `RuleSource` applied as a plugin use the root of the model space as the scope. This can be considered the default scope.

By-path references are always relative to the rule scope. When the scope is the root, this effectively allows binding to any element in the graph. When it is not, then only the children of the scope can be referenced using "by-path" notation.

When binding by-type references, the following elements are considered:

- The scope element itself.
- The immediate children of the scope element.
- The immediate children of the model space (i.e. project space) root.

For the common case, where the rule is effectively scoped to the root, only the immediate children of the root need to be considered.

Binding to all elements in a scope matching type

Mutating or validating all elements of a given type in some scope is a common use-case. To

accommodate this, rules can be applied via the `@Each` annotation.

In the example below, a `@Defaults` rule is applied to each `FileItem` in the model setting a default file size of "1024". Another rule applies a RuleSource to every `DirectoryItem` that makes sure all file sizes are positive and divisible by "16".

Example: a DSL example applying a rule to every element in a scope

build.gradle

```
@Managed interface Item extends Named {}
@Managed interface FileItem extends Item {
    void setSize(int size)
    int getSize()
}
@Managed interface DirectoryItem extends Item {
    ModelMap<Item> getChildren()
}

class PluginRules extends RuleSource {
    @Defaults void setDefaultFileSize(@Each FileItem file) {
        file.size = 1024
    }

    @Rules void applyValidateRules(ValidateRules rules, @Each DirectoryItem directory)
{}
}
apply plugin: PluginRules

abstract class ValidateRules extends RuleSource {
    @Validate
    void validateSizeIsPositive(ModelMap<FileItem> files) {
        files.each { file ->
            assert file.size > 0
        }
    }

    @Validate
    void validateSizeDivisibleBySixteen(ModelMap<FileItem> files) {
        files.each { file ->
            assert file.size % 16 == 0
        }
    }
}

model {
    root(DirectoryItem) {
        children {
            dir(DirectoryItem) {
                children {
                    file1(FileItem)
                    file2(FileItem) { size = 2048 }
                }
            }
            file3(FileItem)
        }
    }
}
```

NOTE	The code for this example can be found at `samples/modelRules/ruleSourcePluginEach` in the '-all' distribution of Gradle.

The model DSL

In addition to using a RuleSource, it is also possible to declare a model and rules directly in a build script using the "model DSL".

TIP	The model DSL makes heavy use of various Groovy DSL features. Please have a read of Groovy DSL basics for an introduction to these Groovy features.

The general form of the model DSL is:

```
model {
    «rule-definitions»
}
```

All rules are nested inside a `model` block. There may be any number of rule definitions inside each `model` block, and there may be any number of `model` blocks in a build script. You can also use a `model` block in build scripts that are applied using `apply from: $uri`.

There are currently 2 kinds of rule that you can define using the model DSL: configuration rules, and creation rules.

Configuration rules

You can define a rule that configures a particular model element. A configuration rule has the following form:

```
model {
    «model-path-to-subject» {
        «configuration code»
    }
}
```

Continuing with the example so far of the model element `"person"` of type `Person` being present, the following DSL snippet adds a configuration rule for the person that sets its `lastName` property.

Example: DSL configuration rule

build.gradle

```
model {
    person {
        lastName = "Smith"
    }
}
```

A configuration rule specifies a path to the subject that should be configured and a closure containing the code to run when the subject is configured. The closure is executed with the subject passed as the closure delegate. Exactly what code you can provide in the closure depends on the type of the subject. This is discussed below.

You should note that the configuration code is not executed immediately but is instead executed only when the subject is required. This is an important behaviour of model rules and allows Gradle to configure only those elements that are required for the build, which helps reduce build time. For example, let's run a task that uses the "person" object:

Example: Configuration run when required

build.gradle

```
model {
    person {
        println "configuring person"
        lastName = "Smith"
    }
}
```

Output of `gradle showPerson`

```
> gradle showPerson
configuring person

> Task :showPerson
Hello John Smith!

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

You can see that before the task is run, the "person" element is configured by running the rule closure. Now let's run a task that does not require the "person" element:

Example: Configuration not run when not required

Output of `gradle somethingElse`

```
> gradle somethingElse

> Task :somethingElse
Not using person

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

In this instance, you can see that the "person" element is not configured at all.

Creation rules

It is also possible to create model elements at the root level. The general form of a creation rule is:

```
model {
    «element-name»(«element-type») {
        «initialization code»
    }
}
```

The following model rule creates the `"person"` element:

Example: DSL creation rule

build.gradle

```
model {
    person(Person) {
        firstName = "John"
    }
}
```

A creation rule definition specifies the path of the element to create, plus its public type, represented as a Java interface or class. Only certain types of model elements can be created.

A creation rule may also provide a closure containing the initialization code to run when the element is created. The closure is executed with the element passed as the closure delegate. Exactly what code you can provide in the closure depends on the type of the subject. This is discussed below.

The initialization closure is optional and can be omitted, for example:

Example: DSL creation rule without initialization

build.gradle

```
model {
    barry(Person)
}
```

You should note that the initialization code is not executed immediately but is instead executed only when the element is required. The initialization code is executed before any configuration rules are run. For example:

Example: Initialization before configuration

build.gradle

```
model {
    person {
        println "configuring person"
        println "last name is $lastName, should be Smythe"
        lastName = "Smythe"
    }
    person(Person) {
        println "creating person"
        firstName = "John"
        lastName = "Smith"
    }
}
```

Output of `gradle showPerson`

```
> gradle showPerson
creating person
configuring person
last name is Smith, should be Smythe

> Task :showPerson
Hello John Smythe!

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Notice that the creation rule appears in the build script *after* the configuration rule, but its code runs before the code of the configuration rule. Gradle collects up all the rules for a particular subject before running any of them, then runs the rules in the appropriate order.

Model rule closures

Most DSL rules take a closure containing some code to run to configure the subject. The code you can use in this closure depends on the type of the subject of the rule.

TIP You can use the model report to determine the type of a particular model element.

In general, a rule closure may contain arbitrary code, mixed with some type specific DSL syntax.

`ModelMap<T>` **subject**

A ModelMap is basically a map of model elements, indexed by some name. When a `ModelMap` is used as the subject of a DSL rule, the rule closure can use any of the methods defined on the ModelMap interface.

A rule closure with `ModelMap` as a subject can also include nested creation or configuration rules. These behave in a similar way to the creation and configuration rules that appear directly under the `model` block.

Here is an example of a nested creation rule:

Example: Nested DSL creation rule

build.gradle

```
model {
    people {
        john(Person) {
            firstName = "John"
        }
    }
}
```

As before, a nested creation rule defines a name and public type for the element, and optionally, a closure containing code to use to initialize the element. The code is run only when the element is required in the build.

Here is an example of a nested configuration rule:

Example: Nested DSL configuration rule

build.gradle

```
model {
    people {
        john {
            lastName = "Smith"
        }
    }
}
```

As before, a nested configuration rule defines the name of the element to configure and a closure containing code to use to configure the element. The code is run only when the element is required in the build.

`ModelMap` introduces several other kinds of rules. For example, you can define a rule that targets each of the elements in the map. The code in the rule closure is executed once for each element in the map, when that element is required. Let's run a task that requires all of the children of the "people" element:

Example: DSL configuration rule for each element in a map

build.gradle

```
model {
    people {
        john(Person) {
            println "creating $it"
            firstName = "John"
            lastName = "Smith"
        }
        all {
            println "configuring $it"
        }
        barry(Person) {
            println "creating $it"
            firstName = "Barry"
            lastName = "Barry"
        }
    }
}
```

Output of `gradle listPeople`

```
> gradle listPeople
creating Person 'people.barry'
configuring Person 'people.barry'
creating Person 'people.john'
configuring Person 'people.john'

> Task :listPeople
Hello Barry Barry!
Hello John Smith!

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Any method on ModelMap that accepts an Action as its last parameter can also be used to define a nested rule.

`@Managed` type subject

When a managed type is used as the subject of a DSL rule, the rule closure can use any of the methods defined on the managed type interface.

A rule closure can also configure the properties of the element using nested closures. For example:

Example: Nested DSL property configuration

build.gradle

```
model {
    person {
        address {
            city = "Melbourne"
        }
    }
}
```

NOTE	Currently, the nested closures do not define rules and are executed immediately. Please be aware that this behaviour will change in a future Gradle release.

All other subjects

For all other types, the rule closure can use any of the methods defined by the type. There is no special DSL defined for these elements.

Automatic type coercion

Scalar properties in managed types can be assigned `CharSequence` values (e.g. `String`, `GString`, etc.) and they will be converted to the actual property type for you. This works for all scalar types including `File`s, which will be resolved relative to the current project.

Example: a DSL example showing type conversions

build.gradle

```
enum Temperature {
   TOO_HOT,
   TOO_COLD,
   JUST_RIGHT
}

@Managed
interface Item {
   void setName(String n); String getName()

   void setQuantity(int q); int getQuantity()

   void setPrice(float p); float getPrice()

   void setTemperature(Temperature t)
   Temperature getTemperature()

   void setDataFile(File f); File getDataFile()
```

```
}

class ItemRules extends RuleSource {
   @Model
   void item(Item item) {
      def data = item.dataFile.text.trim()
      def (name, quantity, price, temp) = data.split(',')
      item.name = name
      item.quantity = quantity
      item.price = price
      item.temperature = temp
   }

   @Defaults
   void setDefaults(Item item) {
      item.dataFile = 'data.csv'
   }

   @Mutate
   void createDataTask(ModelMap<Task> tasks, Item item) {
      tasks.create('showData') {
         doLast {
            println """
Item '$item.name'
   quantity:    $item.quantity
   price:       $item.price
   temperature: $item.temperature"""
         }
      }
   }
}

apply plugin: ItemRules

model {
   item {
      price = "${price * (quantity < 10 ? 2 : 0.5)}"
   }
}
```

NOTE The code for this example can be found at `samples/modelRules/modelDslCoercion` in the '-all' distribution of Gradle.

In the above example, an `Item` is created and is initialized in `setDefaults()` by providing the path to the data file. In the `item()` method the resolved `File` is parsed to extract and set the data. In the DSL block at the end, the price is adjusted based on the quantity; if there are fewer than 10 remaining the price is doubled, otherwise it is reduced by 50%. The `GString` expression is a valid value since it resolves to a `float` value in string form.

Finally, in `createDataTask()` we add the `showData` task to display all of the configured values.

Declaring input dependencies

Rules declared in the DSL may *depend* on other model elements through the use of a special syntax, which is of the form:

```
$.«path-to-model-element»
```

Paths are a period separated list of identifiers. To directly depend on the `firstName` of the person, the following could be used:

```
$.person.firstName
```

Example: a DSL rule using inputs

build.gradle

```
model {
    tasks {
        hello(Task) {
            def p = $.person
            doLast {
                println "Hello $p.firstName $p.lastName!"
            }
        }
    }
}
```

NOTE	The code for this example can be found at `samples/modelRules/modelDsl` in the '-all' distribution of Gradle.

In the above snippet, the `$.person` construct is an input reference. The construct returns the value of the model element at the specified path, as its default type (i.e. the type advertised by the Model Report). It may appear anywhere in the rule that an expression may normally appear. It is not limited to the right hand side of variable assignments.

The input element is guaranteed to be fully configured before the rule executes. That is, all of the rules that mutate the element are guaranteed to have been previously executed, leaving the target element in its final, immutable, state.

Most model elements enforce immutability when being used as inputs. Any attempt to mutate such an element will result in a runtime error. However, some legacy type objects do not currently implement such checks. Regardless, it is always invalid to attempt to mutate an input to a rule.

Using `ModelMap<T>` as an input

When you use a ModelMap as input, each item in the map is made available as a property.

The model report

The built-in ModelReport task displays a hierarchical view of the elements in the model space. Each item prefixed with a `+` on the model report is a model element and the visual nesting of these elements correlates to the model path (e.g. `tasks.help`). The model report displays the following details about each model element:

Table 15. Model report - model element details

Detail	Description
Type	This is the underlying type of the model element and is typically a fully qualified class name.
Value	Is conditionally displayed on the report when a model element can be represented as a string.
Creator	Every model element has a creator. A creator signifies the origin of the model element (i.e. what created the model element).
Rules	Is a listing of the rules, excluding the creator rule, which are executed for a given model element. The order in which the rules are displayed reflects the order in which they are executed.

Example: Model task output

Output of `gradle model`

```
> gradle model

> Task :model

------------------------------------------------------------
Root project
------------------------------------------------------------

+ person
      | Type:       Person
      | Creator:    PersonRules#person(Person)
      | Rules:
            person { ... } @ build.gradle line 97, column 3
            PersonRules#setFirstName(Person)
    + age
          | Type:       int
          | Value:      0
          | Creator:    PersonRules#person(Person)
    + children
          | Type:       org.gradle.model.ModelSet<Person>
          | Creator:    PersonRules#person(Person)
    + employed
          | Type:       boolean
          | Value:      false
          | Creator:    PersonRules#person(Person)
```

```
    + father
          | Type:       Person
          | Value:      null
          | Creator:    PersonRules#person(Person)
    + firstName
          | Type:       java.lang.String
          | Value:      John
          | Creator:    PersonRules#person(Person)
    + homeDirectory
          | Type:       java.io.File
          | Value:      null
          | Creator:    PersonRules#person(Person)
    + id
          | Type:       java.lang.Long
          | Value:      null
          | Creator:    PersonRules#person(Person)
    + lastName
          | Type:       java.lang.String
          | Value:      Smith
          | Creator:    PersonRules#person(Person)
    + maritalStatus
          | Type:       MaritalStatus
          | Creator:    PersonRules#person(Person)
    + mother
          | Type:       Person
          | Value:      null
          | Creator:    PersonRules#person(Person)
    + userGroups
          | Type:       java.util.List<java.lang.String>
          | Value:      null
          | Creator:    PersonRules#person(Person)
+ tasks
      | Type:       org.gradle.model.ModelMap<org.gradle.api.Task>
      | Creator:    Project.<init>.tasks()
      | Rules:
            PersonRules#createHelloTask(ModelMap<Task>, Person)
    + buildEnvironment
          | Type:       org.gradle.api.tasks.diagnostics.BuildEnvironmentReportTask
          | Value:      task ':buildEnvironment'
          | Creator:    Project.<init>.tasks.buildEnvironment()
          | Rules:
                copyToTaskContainer
    + components
          | Type:       org.gradle.api.reporting.components.ComponentReport
          | Value:      task ':components'
          | Creator:    Project.<init>.tasks.components()
          | Rules:
                copyToTaskContainer
    + dependencies
          | Type:       org.gradle.api.tasks.diagnostics.DependencyReportTask
          | Value:      task ':dependencies'
```

```
          | Creator:    Project.<init>.tasks.dependencies()
          | Rules:
              copyToTaskContainer
    + dependencyInsight
          | Type:       org.gradle.api.tasks.diagnostics.DependencyInsightReportTask
          | Value:      task ':dependencyInsight'
          | Creator:    Project.<init>.tasks.dependencyInsight()
          | Rules:
              copyToTaskContainer
    + dependentComponents
          | Type:       org.gradle.api.reporting.dependents.DependentComponentsReport
          | Value:      task ':dependentComponents'
          | Creator:    Project.<init>.tasks.dependentComponents()
          | Rules:
              copyToTaskContainer
    + hello
          | Type:       org.gradle.api.Task
          | Value:      task ':hello'
          | Creator:    PersonRules#createHelloTask(ModelMap<Task>, Person) >
create(hello)
          | Rules:
              copyToTaskContainer
    + help
          | Type:       org.gradle.configuration.Help
          | Value:      task ':help'
          | Creator:    Project.<init>.tasks.help()
          | Rules:
              copyToTaskContainer
    + init
          | Type:       org.gradle.buildinit.tasks.InitBuild
          | Value:      task ':init'
          | Creator:    Project.<init>.tasks.init()
          | Rules:
              copyToTaskContainer
    + model
          | Type:       org.gradle.api.reporting.model.ModelReport
          | Value:      task ':model'
          | Creator:    Project.<init>.tasks.model()
          | Rules:
              copyToTaskContainer
    + projects
          | Type:       org.gradle.api.tasks.diagnostics.ProjectReportTask
          | Value:      task ':projects'
          | Creator:    Project.<init>.tasks.projects()
          | Rules:
              copyToTaskContainer
    + properties
          | Type:       org.gradle.api.tasks.diagnostics.PropertyReportTask
          | Value:      task ':properties'
          | Creator:    Project.<init>.tasks.properties()
          | Rules:
```

```
                copyToTaskContainer
    + tasks
          | Type:       org.gradle.api.tasks.diagnostics.TaskReportTask
          | Value:      task ':tasks'
          | Creator:    Project.<init>.tasks.tasks()
          | Rules:
                copyToTaskContainer
    + wrapper
          | Type:       org.gradle.api.tasks.wrapper.Wrapper
          | Value:      task ':wrapper'
          | Creator:    Project.<init>.tasks.wrapper()
          | Rules:
                copyToTaskContainer
```

Limitations and future direction

The rule engine that was part of the Software Model will be deprecated. Everything under the model block will be ported as extensions to the current model. Native users will no longer have a separate extension model compared to the rest of the Gradle community, and they will be able to make use of the new variant aware dependency management. For more information, see the blog post on the state and future of the software model.

Implementing model rules in a plugin

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 the software model

CAUTION	Rule based configuration will be deprecated. New plugins should not use this concept.

Introduction

One of the strengths of Gradle has always been its extensibility, and its adaptability to new domains. The software model takes this extensibility to a new level, enabling the deep modeling of specific domains via richly typed DSLs. The following chapter describes how the model and the corresponding DSLs can be extended to support domains like the Play Framework or native software development. Before reading this you should be familiar with the Gradle software model rule based configuration and concepts.

The following build script is an example of using a custom software model for building Markdown based documentation:

Example: an example of using a custom software model

build.gradle

```
import sample.documentation.DocumentationComponent
import sample.documentation.TextSourceSet
import sample.markdown.MarkdownSourceSet

apply plugin:sample.documentation.DocumentationPlugin
apply plugin:sample.markdown.MarkdownPlugin

model {
    components {
        docs(DocumentationComponent) {
            sources {
                reference(TextSourceSet)
                userguide(MarkdownSourceSet) {
                    generateIndex = true
                    smartQuotes = true
                }
            }
        }
    }
}
```

NOTE	The code for this example can be found at `samples/customModel/languageType/` in the '-all' distribution of Gradle.

The rest of this chapter is dedicated to explaining what is going on behind this build script.

Concepts

A custom software model type has a public type, a base interface and internal views. Multiple such types then collaborate to define a custom software model.

Public type and base interfaces

Extended types declare a *public type* that extends a *base interface*:

- Components extend the ComponentSpec base interface
- Binaries extend the BinarySpec base interface
- Source sets extend the LanguageSourceSet base interface

The *public type* is exposed to build logic.

Internal views

Adding internal views to your model type, you can make some data visible to build logic via a public type, while hiding the rest of the data behind the internal view types. This is covered in a dedicated section below.

Components all the way down

Components are composed of other components. A source set is just a special kind of component representing sources. It might be that the sources are provided, or generated. Similarly, some components are composed of different binaries, which are built by tasks. All buildable components are built by tasks. In the software model, you will write rules to generate both binaries from components and tasks from binaries.

Components

To declare a custom component type one must extend ComponentSpec, or one of the following, depending on the use case:

- SourceComponentSpec represents a component which has sources
- VariantComponentSpec represents a component which generates different binaries based on context (target platforms, build flavors, ...). Such a component generally produces multiple binaries.
- GeneralComponentSpec is a convenient base interface for components that are built from sources and variant-aware. This is the typical case for a lot of software components, and therefore it should be in most of the cases the base type to be extended.

The core software model includes more types that can be used as base for extension. For example: LibrarySpec and ApplicationSpec can also be extended in this manner. Theses are no-op extensions of `GeneralComponentSpec` used to describe a software model better by distinguishing libraries and applications components. TestSuiteSpec should be used for all components that describe a test suite.

Example: Declare a custom component

DocumentationComponent.groovy

```
@Managed
interface DocumentationComponent extends GeneralComponentSpec {}
```

Types extending `ComponentSpec` are registered via a rule annotated with ComponentType:

Example: Register a custom component

DocumentationPlugin.groovy

```
class DocumentationPlugin extends RuleSource {
    @ComponentType
    void registerComponent(TypeBuilder<DocumentationComponent> builder) {}
}
```

Binaries

To declare a custom binary type one must extend BinarySpec.

Example: Declare a custom binary

DocumentationBinary.groovy

```
@Managed
interface DocumentationBinary extends BinarySpec {
    File getOutputDir()
    void setOutputDir(File outputDir)
}
```

Types extending `BinarySpec` are registered via a rule annotated with ComponentType:

Example: Register a custom binary

DocumentationPlugin.groovy

```
class DocumentationPlugin extends RuleSource {
    @ComponentType
    void registerBinary(TypeBuilder<DocumentationBinary> builder) {}
}
```

Source sets

To declare a custom source set type one must extend LanguageSourceSet.

Example: Declare a custom source set

MarkdownSourceSet.groovy

```
@Managed
interface MarkdownSourceSet extends LanguageSourceSet {
    boolean isGenerateIndex()
    void setGenerateIndex(boolean generateIndex)

    boolean isSmartQuotes()
    void setSmartQuotes(boolean smartQuotes)
}
```

Types extending `LanguageSourceSet` are registered via a rule annotated with ComponentType:

Example: Register a custom source set

MarkdownPlugin.groovy

```
class MarkdownPlugin extends RuleSource {
    @ComponentType
    void registerMarkdownLanguage(TypeBuilder<MarkdownSourceSet> builder) {}
}
```

Setting the *language name* is mandatory.

Putting it all together

Generating binaries from components

Binaries generation from components is done via rules annotated with ComponentBinaries. This rule generates a `DocumentationBinary` named `exploded` for each `DocumentationComponent` and sets its `outputDir` property:

Example: Generates documentation binaries

DocumentationPlugin.groovy

```
class DocumentationPlugin extends RuleSource {
    @ComponentBinaries
    void generateDocBinaries(ModelMap<DocumentationBinary> binaries,
VariantComponentSpec component, @Path("buildDir") File buildDir) {
        binaries.create("exploded") { binary ->
            outputDir = new File(buildDir, "${component.name}/${binary.name}")
        }
    }
}
```

Generating tasks from binaries

Tasks generation from binaries is done via rules annotated with BinaryTasks. This rule generates a `Copy` task for each `TextSourceSet` of each `DocumentationBinary`:

Example: Generates tasks for text source sets

DocumentationPlugin.groovy

```
class DocumentationPlugin extends RuleSource {
    @BinaryTasks
    void generateTextTasks(ModelMap<Task> tasks, final DocumentationBinary binary) {
        binary.inputs.withType(TextSourceSet) { textSourceSet ->
            def taskName = binary.tasks.taskName("compile", textSourceSet.name)
            def outputDir = new File(binary.outputDir, textSourceSet.name)
            tasks.create(taskName, Copy) {
                from textSourceSet.source
                destinationDir = outputDir
            }
        }
    }
}
```

This rule generates a `MarkdownCompileTask` task for each `MarkdownSourceSet` of each `DocumentationBinary`:

Example: Register a custom source set

MarkdownPlugin.groovy

```
class MarkdownPlugin extends RuleSource {
    @BinaryTasks
    void processMarkdownDocumentation(ModelMap<Task> tasks, final DocumentationBinary
binary) {
        binary.inputs.withType(MarkdownSourceSet) { markdownSourceSet ->
            def taskName = binary.tasks.taskName("compile", markdownSourceSet.name)
            def outputDir = new File(binary.outputDir, markdownSourceSet.name)
            tasks.create(taskName, MarkdownHtmlCompile) { compileTask ->
                compileTask.source = markdownSourceSet.source
                compileTask.destinationDir = outputDir
                compileTask.smartQuotes = markdownSourceSet.smartQuotes
                compileTask.generateIndex = markdownSourceSet.generateIndex
            }
        }
    }
}
```

See the sample source for more on the `MarkdownCompileTask` task.

Using your custom model

This build script demonstrate usage of the custom model defined in the sections above:

Example: an example of using a custom software model

build.gradle

```
import sample.documentation.DocumentationComponent
import sample.documentation.TextSourceSet
import sample.markdown.MarkdownSourceSet

apply plugin:sample.documentation.DocumentationPlugin
apply plugin:sample.markdown.MarkdownPlugin

model {
    components {
        docs(DocumentationComponent) {
            sources {
                reference(TextSourceSet)
                userguide(MarkdownSourceSet) {
                    generateIndex = true
                    smartQuotes = true
                }
            }
        }
    }
}
```

NOTE	The code for this example can be found at `samples/customModel/languageType/` in the '-all' distribution of Gradle.

And in the components reports for such a build script we can see our model types properly registered:

Example: components report

Output of `gradle -q components`

```
> gradle -q components

------------------------------------------------------------
Root project
------------------------------------------------------------

DocumentationComponent 'docs'
-----------------------------

Source sets
    Markdown source 'docs:userguide'
        srcDir: src/docs/userguide
    Text source 'docs:reference'
        srcDir: src/docs/reference

Binaries
    DocumentationBinary 'docs:exploded'
        build using task: :docsExploded

Note: currently not all plugins register their components, so some components may not
be visible here.
```

About internal views

Internal views can be added to an already registered type or to a new custom type. In other words, using internal views, you can attach extra properties to already registered components, binaries and source sets types like `JvmLibrarySpec`, `JarBinarySpec` or `JavaSourceSet` and to the custom types you write.

Let's start with a simple component public type and its internal view declarations:

Example: public type and internal view declaration

build.gradle

```
@Managed interface MyComponent extends ComponentSpec {
    String getPublicData()
    void setPublicData(String data)
}
@Managed interface MyComponentInternal extends MyComponent {
    String getInternalData()
    void setInternalData(String internal)
}
```

The type registration is as follows:

Example: type registration

build.gradle

```
class MyPlugin extends RuleSource {
    @ComponentType
    void registerMyComponent(TypeBuilder<MyComponent> builder) {
        builder.internalView(MyComponentInternal)
    }
}
```

The `internalView(type)` method of the type builder can be called several times. This is how you would add several internal views to a type.

Now, let's mutate both public and internal data using some rule:

Example: public and internal data mutation

build.gradle

```
class MyPlugin extends RuleSource {
    @Mutate
    void mutateMyComponents(ModelMap<MyComponentInternal> components) {
        components.all { component ->
            component.publicData = "Some PUBLIC data"
            component.internalData = "Some INTERNAL data"
        }
    }
}
```

Our `internalData` property should not be exposed to build logic. Let's check this using the `model` task on the following build file:

Example: Build script and model report output

build.gradle

```
apply plugin: MyPlugin
model {
    components {
        my(MyComponent)
    }
}
```

Output of `gradle -q model`

```
> gradle -q model

------------------------------------------------------------
```

```
Root project
------------------------------------------------------------

+ components
      | Type:       org.gradle.platform.base.ComponentSpecContainer
      | Creator:    ComponentBasePlugin.PluginRules#components(ComponentSpecContainer)
      | Rules:
            components { ... } @ build.gradle line 53, column 5
            MyPlugin#mutateMyComponents(ModelMap<MyComponentInternal>)
    + my
          | Type:       MyComponent
          | Creator:    components { ... } @ build.gradle line 53, column 5 >
create(my)
          | Rules:
                MyPlugin#mutateMyComponents(ModelMap<MyComponentInternal>) > all()
        + publicData
              | Type:       java.lang.String
              | Value:      Some PUBLIC data
              | Creator:    components { ... } @ build.gradle line 53, column 5 >
create(my)
+ tasks
      | Type:       org.gradle.model.ModelMap<org.gradle.api.Task>
      | Creator:    Project.<init>.tasks()
    + assemble
          | Type:       org.gradle.api.DefaultTask
          | Value:      task ':assemble'
          | Creator:    Project.<init>.tasks.assemble()
          | Rules:
              copyToTaskContainer
    + build
          | Type:       org.gradle.api.DefaultTask
          | Value:      task ':build'
          | Creator:    Project.<init>.tasks.build()
          | Rules:
              copyToTaskContainer
    + buildEnvironment
          | Type:       org.gradle.api.tasks.diagnostics.BuildEnvironmentReportTask
          | Value:      task ':buildEnvironment'
          | Creator:    Project.<init>.tasks.buildEnvironment()
          | Rules:
              copyToTaskContainer
    + check
          | Type:       org.gradle.api.DefaultTask
          | Value:      task ':check'
          | Creator:    Project.<init>.tasks.check()
          | Rules:
              copyToTaskContainer
    + clean
          | Type:       org.gradle.api.tasks.Delete
          | Value:      task ':clean'
          | Creator:    Project.<init>.tasks.clean()
```

```
      | Rules:
            copyToTaskContainer
+ components
      | Type:       org.gradle.api.reporting.components.ComponentReport
      | Value:      task ':components'
      | Creator:    Project.<init>.tasks.components()
      | Rules:
            copyToTaskContainer
+ dependencies
      | Type:       org.gradle.api.tasks.diagnostics.DependencyReportTask
      | Value:      task ':dependencies'
      | Creator:    Project.<init>.tasks.dependencies()
      | Rules:
            copyToTaskContainer
+ dependencyInsight
      | Type:       org.gradle.api.tasks.diagnostics.DependencyInsightReportTask
      | Value:      task ':dependencyInsight'
      | Creator:    Project.<init>.tasks.dependencyInsight()
      | Rules:
            copyToTaskContainer
+ dependentComponents
      | Type:       org.gradle.api.reporting.dependents.DependentComponentsReport
      | Value:      task ':dependentComponents'
      | Creator:    Project.<init>.tasks.dependentComponents()
      | Rules:
            copyToTaskContainer
+ help
      | Type:       org.gradle.configuration.Help
      | Value:      task ':help'
      | Creator:    Project.<init>.tasks.help()
      | Rules:
            copyToTaskContainer
+ init
      | Type:       org.gradle.buildinit.tasks.InitBuild
      | Value:      task ':init'
      | Creator:    Project.<init>.tasks.init()
      | Rules:
            copyToTaskContainer
+ model
      | Type:       org.gradle.api.reporting.model.ModelReport
      | Value:      task ':model'
      | Creator:    Project.<init>.tasks.model()
      | Rules:
            copyToTaskContainer
+ projects
      | Type:       org.gradle.api.tasks.diagnostics.ProjectReportTask
      | Value:      task ':projects'
      | Creator:    Project.<init>.tasks.projects()
      | Rules:
            copyToTaskContainer
+ properties
```

```
        | Type:       org.gradle.api.tasks.diagnostics.PropertyReportTask
        | Value:      task ':properties'
        | Creator:    Project.<init>.tasks.properties()
        | Rules:
              copyToTaskContainer
    + tasks
        | Type:       org.gradle.api.tasks.diagnostics.TaskReportTask
        | Value:      task ':tasks'
        | Creator:    Project.<init>.tasks.tasks()
        | Rules:
              copyToTaskContainer
    + wrapper
        | Type:       org.gradle.api.tasks.wrapper.Wrapper
        | Value:      task ':wrapper'
        | Creator:    Project.<init>.tasks.wrapper()
        | Rules:
              copyToTaskContainer
```

We can see in this report that `publicData` is present and that `internalData` is not.

Groovy Projects

Groovy Quickstart

To build a Groovy project, you use the *Groovy plugin*. This plugin extends the Java plugin to add Groovy compilation capabilities to your project. Your project can contain Groovy source code, Java source code, or a mix of the two. In every other respect, a Groovy project is identical to a Java project, which we have already seen in the Java projects tutorial.

A basic Groovy project

Let's look at an example. To use the Groovy plugin, add the following to your build file:

Example 406. Groovy plugin

build.gradle

```
plugins {
    id 'groovy'
}
```

build.gradle.kts

```
plugins {
    groovy
}
```

NOTE The code for this example can be found at `samples/groovy/quickstart` in the '-all' distribution of Gradle.

This will also apply the Java plugin to the project, if it has not already been applied. The Groovy plugin extends the `compile` task to look for source files in directory `src/main/groovy`, and the `compileTest` task to look for test source files in directory `src/test/groovy`. The compile tasks use joint compilation for these directories, which means they can contain a mixture of Java and Groovy source files.

To use the Groovy compilation tasks, you must also declare the Groovy version to use and where to find the Groovy libraries. You do this by adding a dependency to the `groovy` configuration. The `compile` configuration inherits this dependency, so the Groovy libraries will be included in classpath when compiling Groovy and Java source. For our sample, we will use Groovy 2.2.0 from the public Maven repository:

Example 407. Dependency on Groovy

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")
}
```

Here is our complete build file:

Example 408. Groovy example - complete build file

build.gradle

```
plugins {
    id 'groovy'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.codehaus.groovy:groovy-all:2.4.15'
}
```

build.gradle.kts

```
plugins {
    groovy
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.codehaus.groovy:groovy-all:2.4.15")
}
```

Running `gradle build` will compile, test and JAR your project.

Summary

This chapter describes a very simple Groovy project. Usually, a real project will require more than this. Because a Groovy project *is* a Java project, whatever you can do with a Java project, you can also do with a Groovy project.

You can find out more about the Groovy plugin, and you can find more sample Groovy projects in the `samples/groovy` directory in the Gradle distribution.

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.

Usage

To use the Groovy plugin, include the following in your build script:

Example 409. 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.

`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 16. Groovy plugin - additional task dependencies

Task name	Depends on
`classes`	`compileGroovy`
`testClasses`	`compileTestGroovy`
`sourceSetClasses`	`compileSourceSetGroovy`

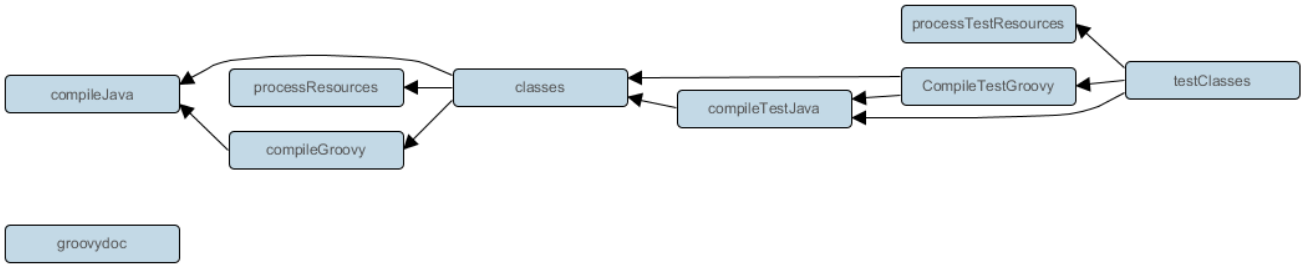


Figure 26. Groovy plugin - tasks

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. [10: Gradle uses the same conventions as introduced by Russel Winder's Gant tool.] 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 410. Custom Groovy source layout

build.gradle

```
sourceSets {
    main {
        groovy {
            srcDirs = ['src/groovy']
        }
    }

    test {
        groovy {
            srcDirs = ['test/groovy']
        }
    }
}
```

build.gradle.kts

```
sourceSets {
    main {
        withConvention(GroovySourceSet::class) {
            groovy {
                setSrcDirs(listOf("src/groovy"))
            }
        }
    }

    test {
        withConvention(GroovySourceSet::class) {
            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 411. 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 412. 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 413. 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 414. 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 convention properties 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` — *SourceDirectorySet* *(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 extends the JavaCompile task (see the relevant Java Plugin section). The GroovyCompile task supports most configuration options of the official Groovy compiler.

Table 17. 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
destinationDir	File.	sourceSet.groovy.outputDir
groovyClasspath	FileCollection	groovy configuration if non-empty; Groovy library found on classpath otherwise

Compiling and testing for Java 6 or Java 7

The Groovy compiler will always be executed with the same version of Java that was used to start Gradle. You should set sourceCompatibility and targetCompatibility to 1.6 or 1.7. If you also have Java source files, you can follow the same steps as for the Java plugin to ensure the correct Java compiler is used.

Example: Configure Java 6 build for Groovy

gradle.properties

```
# in $HOME/.gradle/gradle.properties
java6Home=/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

build.gradle

```
java {
    sourceCompatibility = JavaVersion.VERSION_1_6
    targetCompatibility = JavaVersion.VERSION_1_6
}

assert hasProperty('java6Home') : "Set the property 'java6Home' in your your
gradle.properties pointing to a Java 6 installation"
def javaExecutablesPath = new File(java6Home, 'bin')
def javaExecutables = [:].withDefault { execName ->
    def executable = new File(javaExecutablesPath, execName)
    assert executable.exists() : "There is no $execName executable in
$javaExecutablesPath"
    executable
}
tasks.withType(AbstractCompile) {
    options.with {
        fork = true
        forkOptions.javaHome = file(java6Home)
    }
}
tasks.withType(Javadoc) {
    executable = javaExecutables.javadoc
}
tasks.withType(Test) {
    executable = javaExecutables.java
}
tasks.withType(JavaExec) {
    executable = javaExecutables.java
}
```

build.gradle.kts

```
java {
    sourceCompatibility = JavaVersion.VERSION_1_6
    targetCompatibility = JavaVersion.VERSION_1_6
}

require(hasProperty("java6Home")) { "Set the property 'java6Home' in your
your gradle.properties pointing to a Java 6 installation" }
val java6Home: String by project
val javaExecutablesPath = File(java6Home, "bin")
fun javaExecutable(execName: String): String {
    val executable = File(javaExecutablesPath, execName)
    require(executable.exists()) { "There is no $execName executable in
$javaExecutablesPath" }
    return executable.toString()
}
tasks.withType<JavaCompile> {
    options.apply {
        isFork = true
        forkOptions.javaHome = file(java6Home)
    }
}
tasks.withType<Javadoc> {
    executable = javaExecutable("javadoc")
}
tasks.withType<Test> {
    executable = javaExecutable("java")
}
tasks.withType<JavaExec> {
    executable = javaExecutable("java")
}
```

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 415. 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 Java source files.

`codenarcTest` — *CodeNarc*

Runs CodeNarc against the test Java source files.

`codenarcSourceSet` — *CodeNarc*

Runs CodeNarc against the given source set's Java 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 18. CodeNarc plugin - dependency configurations

Name	Meaning
`codenarc`	The CodeNarc libraries to use

Configuration

See the CodeNarcExtension class in the API documentation.

Java Projects

Java Quickstart

The Java plugin

As we have seen, Gradle is a general-purpose build tool. It can build pretty much anything you care to implement in your build script. Out-of-the-box, however, it doesn't build anything unless you add code to your build script to do so.

Most Java projects are pretty similar as far as the basics go: you need to compile your Java source files, run some unit tests, and create a JAR file containing your classes. It would be nice if you didn't have to code all this up for every project. Luckily, you don't have to. Gradle solves this problem through the use of *plugins*. A plugin is an extension to Gradle which configures your project in some way, typically by adding some pre-configured tasks which together do something useful. Gradle ships with a number of plugins, and you can easily write your own and share them with others. One such plugin is the *Java plugin*. This plugin adds some tasks to your project which will compile and unit test your Java source code, and bundle it into a JAR file.

The Java plugin is convention based. This means that the plugin defines default values for many aspects of the project, such as where the Java source files are located. If you follow the convention in your project, you generally don't need to do much in your build script to get a useful build. Gradle allows you to customize your project if you don't want to or cannot follow the convention in some way. In fact, because support for Java projects is implemented as a plugin, you don't have to use the plugin at all to build a Java project, if you don't want to.

We have in-depth coverage with many examples about the Java plugin, dependency management and multi-project builds in later chapters. In this chapter we want to give you an initial idea of how to use the Java plugin to build a Java project.

A basic Java project

Let's look at a simple example. To use the Java plugin, add the following to your build file:

Example 416. Using the Java plugin

build.gradle

```
plugins {
    id 'java'
}
```

build.gradle.kts

```
plugins {
    java
}
```

NOTE The code for this example can be found at `samples/java/quickstart` in the '-all' distribution of Gradle.

This is all you need to define a Java project. This will apply the Java plugin to your project, which adds a number of tasks to your project.

TIP

What tasks are available?

You can use `gradle tasks` to list the tasks of a project. This will let you see the tasks that the Java plugin has added to your project.

Gradle expects to find your production source code under `src/main/java` and your test source code under `src/test/java`. In addition, any files under `src/main/resources` will be included in the JAR file as resources, and any files under `src/test/resources` will be included in the classpath used to run the tests. All output files are created under the `build` directory, with the JAR file ending up in the `build/libs` directory.

Building the project

The Java plugin adds quite a few tasks to your project. However, there are only a handful of tasks that you will need to use to build the project. The most commonly used task is the `build` task, which does a full build of the project. When you run `gradle build`, Gradle will compile and test your code, and create a JAR file containing your main classes and resources:

Example: Building a Java project

Output of `gradle build`

```
> gradle build
> Task :compileJava
> Task :processResources
> Task :classes
> Task :jar
> Task :assemble
> Task :compileTestJava
> Task :processTestResources
> Task :testClasses
> Task :test
> Task :check
> Task :build

BUILD SUCCESSFUL in 0s
6 actionable tasks: 6 executed
```

Some other useful tasks are:

clean

Deletes the `build` directory, removing all built files.

assemble

Compiles and jars your code, but does not run the unit tests. Other plugins add more artifacts to this task. For example, if you use the War plugin, this task will also build the WAR file for your project.

check

Compiles and tests your code. Other plugins add more checks to this task. For example, if you use the `checkstyle` plugin, this task will also run Checkstyle against your source code.

External dependencies

Usually, a Java project will have some dependencies on external JAR files. To reference these JAR files in the project, you need to tell Gradle where to find them. In Gradle, artifacts such as JAR files, are located in a *repository*. A repository can be used for fetching the dependencies of a project, or for publishing the artifacts of a project, or both. For this example, we will use the public Maven repository:

Example 417. Adding Maven repository

build.gradle

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

build.gradle.kts

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

Let's add some dependencies. Here, we will declare that our production classes have a compile-time dependency on commons collections, and that our test classes have a compile-time dependency on junit:

Example 418. Adding dependencies

build.gradle

```
dependencies {
    implementation group: 'commons-collections', name: 'commons-collections',
version: '3.2.2'
    testImplementation group: 'junit', name: 'junit', version: '4.+'
}
```

build.gradle.kts

```
dependencies {
    implementation(group = "commons-collections", name = "commons-
collections", version = "3.2.2")
    testImplementation(group = "junit", name = "junit", version = "4.+")
}
```

You can find out more in Dependency Management for Java Projects.

Customizing the project

The Java plugin adds a number of properties to your project. These properties have default values which are usually sufficient to get started. It's easy to change these values if they don't suit. Let's look at this for our sample. Here we will specify the version number for our Java project, along with some attributes to the JAR manifest.

Example 419. Customization of MANIFEST.MF

build.gradle

```
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart',
                   'Implementation-Version': version
    }
}
```

build.gradle.kts

```
version = "1.0"
tasks.jar {
    manifest {
        attributes(
            "Implementation-Title" to "Gradle Quickstart",
            "Implementation-Version" to version)
    }
}
```

TIP

What properties are available?

You can use `gradle properties` to list the properties of a project. This will allow you to see the properties added by the Java plugin, and their default values.

The tasks which the Java plugin adds are regular tasks, exactly the same as if they were declared in the build file. This means you can use any of the mechanisms shown in earlier chapters to customize these tasks. For example, you can set the properties of a task, add behaviour to a task, change the dependencies of a task, or replace a task entirely. In our sample, we will configure the `test` task, which is of type Test, to add a system property when the tests are executed:

Example 420. Adding a test system property

build.gradle

```
test {
    systemProperties 'property': 'value'
}
```

build.gradle.kts

```
tasks.test {
    systemProperties("property" to "value")
}
```

Publishing the JAR file

Usually the JAR file needs to be published somewhere. To do this, you need to tell Gradle where to publish the JAR file. In Gradle, artifacts such as JAR files are published to repositories. In our sample, we will publish to a local directory. You can also publish to a remote location, or multiple locations.

Example 421. Publishing the JAR file

build.gradle

```
uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

build.gradle.kts

```
tasks.getByName<Upload>("uploadArchives") {
    repositories {
        flatDir {
            dirs("repos")
        }
    }
}
```

To publish the JAR file, run `gradle uploadArchives`.

Creating an Eclipse project

To create the Eclipse-specific descriptor files, like `.project`, you need to add another plugin to your build file:

Example 422. Eclipse plugin

build.gradle

```
plugins {
    id 'eclipse'
}
```

build.gradle.kts

```
plugins {
    eclipse
}
```

Now execute `gradle eclipse` command to generate Eclipse project files. More information about the `eclipse` task can be found in the Eclipse Plugin chapter.

Summary

Here's the complete build file for our sample:

Example 423. Java example - complete build file

build.gradle

```
plugins {
    id 'java'
    id 'eclipse'
}
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart',
                   'Implementation-Version': version
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation group: 'commons-collections', name: 'commons-collections',
version: '3.2.2'
    testImplementation group: 'junit', name: 'junit', version: '4.+'
}

test {
    systemProperties 'property': 'value'
}

uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
// end:complete-example[]
```

build.gradle.kts

```
plugins {
    java
    eclipse
}
version = "1.0"
tasks.jar {
    manifest {
        attributes(
            "Implementation-Title" to "Gradle Quickstart",
            "Implementation-Version" to version)
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation(group = "commons-collections", name = "commons-
collections", version = "3.2.2")
    testImplementation(group = "junit", name = "junit", version = "4.+")
}

tasks.test {
    systemProperties("property" to "value")
}

tasks.getByName<Upload>("uploadArchives") {
    repositories {
        flatDir {
            dirs("repos")
        }
    }
}
```

Multi-project Java build

Now let’s look at a typical multi-project build. Below is the layout for the project:

Multi-project build - hierarchical layout

```
.
├─── api
├─── services
│    ├─── shared
│    └─── webservice
└─── shared
```

Here we have four projects. Project `api` produces a JAR file which is shipped to the client to provide them a Java client for your XML webservice. Project `webservice` is a webapp which returns XML. Project `shared` contains code used both by `api` and `webservice`. Project `services/shared` has code that depends on the `shared` project.

Defining a multi-project build

To define a multi-project build, you need to create a *settings file*. The settings file lives in the root directory of the source tree, and specifies which projects to include in the build. It must be called `settings.gradle` (with Groovy) or `settings.gradle.kts` (with Kotlin). For this example, we are using a simple hierarchical layout. Here is the corresponding settings file:

Example 424. Multi-project build - settings file

settings.gradle

```
include "shared", "api", "services:webservice", "services:shared"
```

settings.gradle.kts

```
include("shared", "api", "services:webservice", "services:shared")
```

You can find out more about the settings file in Authoring Multi-Project Builds.

Common configuration

For most multi-project builds, there is some configuration which is common to all projects. In our sample, we will define this common configuration in the root project, using a technique called *configuration injection*. Here, the root project is like a container and the `subprojects` method iterates over the elements of this container - the projects in this instance - and injects the specified configuration. This way we can easily define the manifest content for all archives, and some common dependencies:

Example 425. Multi-project build - common configuration

build.gradle

```
subprojects {
    apply plugin: 'java'
    apply plugin: 'eclipse-wtp'

    repositories {
       mavenCentral()
    }

    dependencies {
        testImplementation 'junit:junit:4.12'
    }

    version = '1.0'

    jar {
        manifest.attributes provider: 'gradle'
    }
}
```

build.gradle.kts

```
subprojects {
    apply(plugin = "java")
    apply(plugin = "eclipse-wtp")

    repositories {
        mavenCentral()
    }

    dependencies {
        "testImplementation"("junit:junit:4.12")
    }

    version = "1.0"

    tasks.getByName<Jar>("jar") {
        manifest.attributes("provider" to "gradle")
    }
}
```

Notice that our sample applies the Java plugin to each subproject. This means the tasks and

configuration properties we have seen in the previous section are available in each subproject. So, you can compile, test, and JAR all the projects by running `gradle build` from the root project directory.

Also note that these plugins are only applied within the `subprojects` section, not at the root level, so the root build will not expect to find Java source files in the root project, only in the subprojects.

Dependencies between projects

You can add dependencies between projects in the same build, so that, for example, the JAR file of one project is used to compile another project. In the `api` build file we will add a dependency on the `shared` project. Due to this dependency, Gradle will ensure that project `shared` always gets built before project `api`.

Example 426. Multi-project build - dependencies between projects

api/build.gradle

```
dependencies {
    implementation project(':shared')
}
```

api/build.gradle.kts

```
dependencies {
    implementation(project(":shared"))
}
```

Creating a distribution

We also add a distribution, that gets shipped to the client:

Example 427. Multi-project build - distribution file

api/build.gradle

```
task dist(type: Zip) {
    dependsOn spiJar
    from 'src/dist'
    into('libs') {
        from spiJar.archivePath
        from configurations.runtimeClasspath
    }
}

artifacts {
   archives dist
}
```

api/build.gradle.kts

```
val dist = task<Zip>("dist") {
    val spiJar = tasks.getByName<Jar>("spiJar")
    dependsOn(spiJar)
    from("src/dist")
    into("libs") {
        from(spiJar.archivePath)
        from(configurations.runtimeClasspath)
    }
}

artifacts {
    add("archives", dist)
}
```

Where to next?

In this chapter, you have seen how to do some of the things you commonly need to build a Java based project. This chapter is not exhaustive, and there are many other things you can do with Java projects in Gradle. You can find out more about the Java plugin, and you can find more sample Java projects in the `samples/java` directory in the Gradle distribution.

Otherwise, continue on to Dependency Management for Java 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 Quickstart first as it will give you a good overview of the basics.

Introduction

The simplest build script for a Java project applies the Java Plugin and optionally sets the project version and Java compatibility versions:

Example 428. Applying the Java Plugin

build.gradle

```
plugins {
    id 'java'
}

sourceCompatibility = '1.8'
targetCompatibility = '1.8'
version = '1.2.1'
```

build.gradle.kts

```
plugins {
    java
}

java {
    sourceCompatibility = JavaVersion.VERSION_1_8
    targetCompatibility = JavaVersion.VERSION_1_8
}

version = "1.2.1"
```

By applying the Java 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 compatibility options mitigate against problems with the project being built with different Java compiler versions, and 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 Plugin also integrates the above tasks into the standard Base Plugin lifecycle tasks:

- `jar` is attached to `assemble` [11: In fact, any artifact added to the `archives` configuration will be built by `assemble`]
- `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:

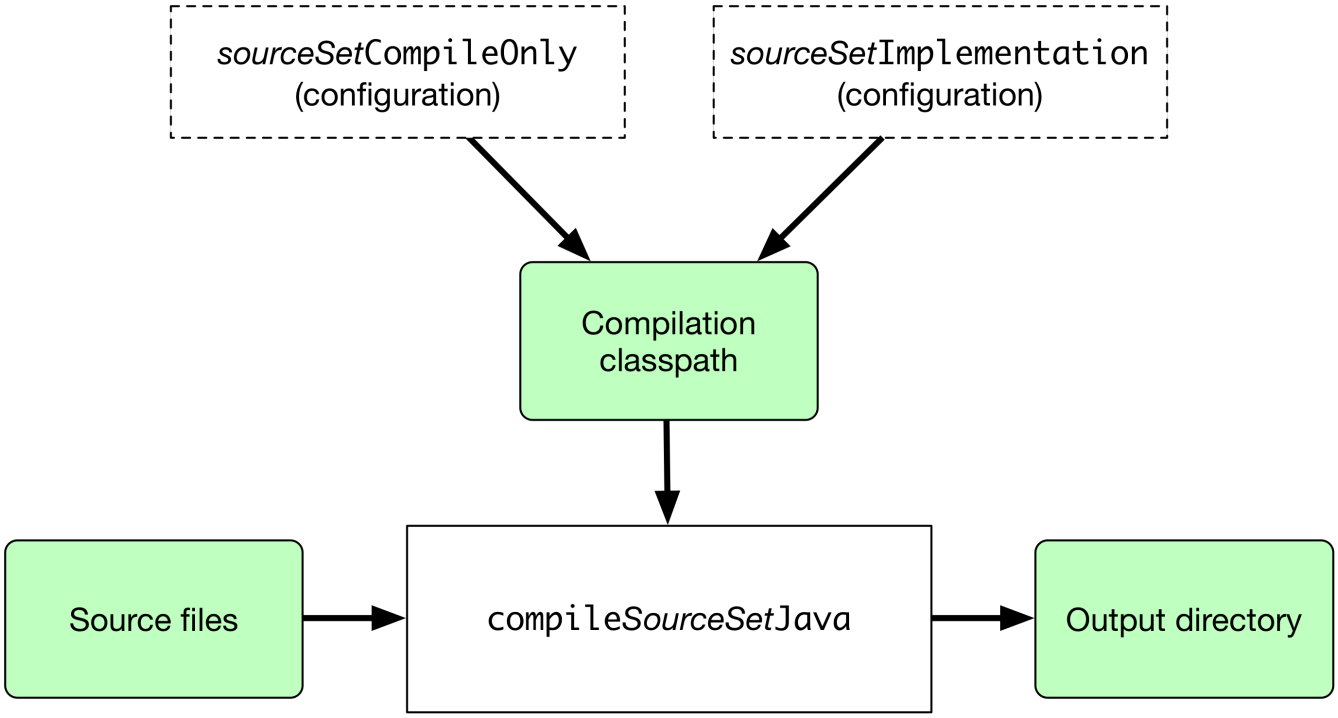


Figure 27. Source sets and Java compilation

The shaded boxes represent properties of the source set itself. On top of that, the Java 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 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:

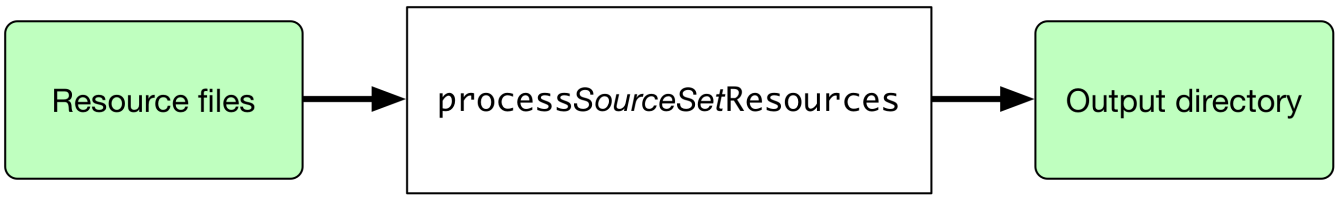


Figure 28. Processing non-source files for a source set

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 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 429. 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 creates an additional configuration — `api` — for dependencies that are required for compiling both the module and any modules that depend on it.

NOTE

Why no `compile` configuration?

The Java 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 — although it won't be going away any time soon — 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 430. 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 431. 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 432. 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.

Two common options for the Java compiler are defined at the project level:

`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.

If you need or want more than one compilation task for any reason, you can either create a new source set or simply define a new task of type JavaCompile. We look at setting up a new source set next.

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 is not supported.

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.

The following sample shows how the `build.gradle` needs to be adjusted. In order to be able to make the build machine-independent, the location of the old Java home and target version should be configured in `GRADLE_USER_HOME/gradle.properties` [12: For more details on `gradle.properties` see Gradle configuration properties] in the user's home directory on each developer machine, as shown in the example.

Example: Configure Java 7 build

gradle.properties

```
# in $HOME/.gradle/gradle.properties
javaHome=/Library/Java/JavaVirtualMachines/jdk1.7.0_80.jdk/Contents/Home
targetJavaVersion=1.7
```

build.gradle

```
assert hasProperty('javaHome'): "Set the property 'javaHome' in your your
gradle.properties pointing to a Java 6 or 7 installation"
assert hasProperty('targetJavaVersion'): "Set the property
'targetJavaVersion' in your your gradle.properties to '1.6' or '1.7'"

java {
    sourceCompatibility = JavaVersion.toVersion(targetJavaVersion)
}

def javaExecutablesPath = new File(javaHome, 'bin')
def javaExecutables = [:].withDefault { execName ->
    def executable = new File(javaExecutablesPath, execName)
    assert executable.exists(): "There is no ${execName} executable in
${javaExecutablesPath}"
    executable
}
tasks.withType(AbstractCompile) {
    options.with {
        fork = true
        forkOptions.javaHome = file(javaHome)
    }
}
tasks.withType(Javadoc) {
    executable = javaExecutables.javadoc
}
tasks.withType(Test) {
    executable = javaExecutables.java
}
tasks.withType(JavaExec) {
    executable = javaExecutables.java
}
```

build.gradle.kts

```
require(hasProperty("javaHome")) { "Set the property 'javaHome' in your your
gradle.properties pointing to a Java 6 or 7 installation" }
require(hasProperty("targetJavaVersion")) { "Set the property
'targetJavaVersion' in your your gradle.properties to '1.6' or '1.7'" }

val javaHome: String by project
val targetJavaVersion: String by project

java {
    sourceCompatibility = JavaVersion.toVersion(targetJavaVersion)
}

val javaExecutablesPath = File(javaHome, "bin")
fun javaExecutable(execName: String): String {
    val executable = File(javaExecutablesPath, execName)
    require(executable.exists()) { "There is no ${execName} executable in
${javaExecutablesPath}" }
    return executable.toString()
}
tasks.withType<JavaCompile> {
    options.apply {
        isFork = true
        forkOptions.javaHome = file(javaHome)
    }
}
tasks.withType<Javadoc> {
    executable = javaExecutable("javadoc")
}
tasks.withType<Test> {
    executable = javaExecutable("java")
}
tasks.withType<JavaExec> {
    executable = javaExecutable("java")
}
```

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 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 `Copy` 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 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 Plugin.

The one and only packaging feature provided by the Java Plugin directly is a `jar` task that packages all the compiled production classes and resources into a single JAR. This JAR is then added as an artifact — as opposed to a dependency — in the `archives` configuration, hence why it is automatically built by the `assemble` task.

If you want any other JAR or alternative archive built, you either have to apply an appropriate plugin or create the task manually. For example, if you want a task that generates a 'sources' JAR, define your own `Jar` task like so:

Example 433. Defining a custom task to create a 'sources' JAR

build.gradle

```
task sourcesJar(type: Jar) {
    classifier = 'sources'
    from sourceSets.main.allJava
}
```

build.gradle.kts

```
task<Jar>("sourcesJar") {
    classifier = "sources"
    from(sourceSets.main.get().allJava)
}
```

See Jar for more details on the configuration options available to you. And note that you need to use `classifier` rather than `appendix` here for correct publication of the JAR.

If you instead want to create an 'uber' (AKA 'fat') JAR, then you can use a task definition like this:

Example 434. 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'
}

task uberJar(type: Jar) {
    appendix = '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")
}

task<Jar>("uberJar") {
    appendix = "uber"

    from(sourceSets.main.get().output)

    dependsOn(configurations.runtimeClasspath)
    from(Callable {
        configurations.runtimeClasspath.filter { it.name.endsWith("jar")
}.map { zipTree(it) }
    })
}
```

There are several options for publishing a JAR once it has been created:

- the Maven Publish Plugin
- the Ivy Publish Plugin
- the `uploadArchives` task — the original publishing mechanism — which works with both Ivy and (if you apply the Maven Plugin) Maven

The former two "Publish" plugins are the preferred options.

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 435. Customization of MANIFEST.MF

build.gradle

```
jar {
    manifest {
        attributes("Implementation-Title": "Gradle",
                   "Implementation-Version": version)
    }
}
```

build.gradle.kts

```
tasks.jar {
    manifest {
        attributes(
            "Implementation-Title" to "Gradle",
            "Implementation-Version" to version
        )
    }
}
```

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 436. Creating a manifest object.

build.gradle

```
ext.sharedManifest = manifest {
    attributes("Implementation-Title": "Gradle",
               "Implementation-Version": version)
}
task fooJar(type: Jar) {
    manifest = project.manifest {
        from sharedManifest
    }
}
```

build.gradle.kts

```
val sharedManifest = the<JavaPluginConvention>().manifest {
    attributes (
        "Implementation-Title" to "Gradle",
        "Implementation-Version" to version
    )
}

task<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 437. Separate MANIFEST.MF for a particular archive

build.gradle

```
task barJar(type: 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

```
task<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 438. Saving a MANIFEST.MF to disk

build.gradle

```
jar.manifest.writeTo("$buildDir/mymanifest.mf")
```

build.gradle.kts

```
tasks.getByName<Jar>("jar").manifest.writeTo("$buildDir/mymanifest.mf")
```

Generating API documentation

The Java 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 439. Using a custom doclet with Javadoc

build.gradle

```
configurations {
    asciidoclet
}

dependencies {
    asciidoclet 'org.asciidoctor:asciidoclet:1.+'
}

task 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.+")
}

task("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 440. Defining a custom Javadoc task

build.gradle

```
task testJavadoc(type: Javadoc) {
    source = sourceSets.test.allJava
}
```

build.gradle.kts

```
task<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 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 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*.

NOTE The Java Library Plugin automatically applies the standard Java Plugin as well.

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 see a basic, practical example of building a Java library in the corresponding *guide*.

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

Note that you will need to explicitly apply the Java Plugin in your build script.

You can see a basic example of building a Java application in the corresponding *guide*.

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.

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, we look at the basics of JVM testing in Gradle.

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 4.x and changes the maximum heap size for the tests' JVM to 1 gigabyte:

Example 441. A basic configuration for the 'test' task

build.gradle

```
dependencies {
    testImplementation 'junit:junit:4.12'
}

test {
    useJUnit()

    maxHeapSize = '1G'
}
```

build.gradle.kts

```
dependencies {
    testImplementation("junit:junit:4.12")
}

tasks.test {
    useJUnit()

    maxHeapSize = "1G"
}
```

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 442. 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.

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 443. Changing the default test report and results directories

build.gradle

```
reporting.baseDir = "my-reports"
testResultsDirName = "$buildDir/my-test-results"

task 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")

task("showDirs") {
    doLast {

logger.quiet(rootDir.toPath().relativize((project.properties["reportsDir"] as
File).toPath()).toString())

logger.quiet(rootDir.toPath().relativize((project.properties["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 444. Creating a unit test report for subprojects

build.gradle

```
subprojects {
    apply plugin: 'java'

    // Disable the test report for the individual test task
    test {
        reports.html.enabled = false
    }
}

task testReport(type: TestReport) {
    destinationDir = file("$buildDir/reports/allTests")
    // Include the results from the `test` task in all subprojects
    reportOn subprojects*.test
}
```

build.gradle.kts

```
subprojects {
    apply(plugin = "java")

    // Disable the test report for the individual test task
    tasks.getByName<Test>("test") {
        reports.html.isEnabled = false
    }
}

task<TestReport>("testReport") {
    destinationDir = file("$buildDir/reports/allTests")
    // Include the results from the `test` task in all subprojects
    reportOn(subprojects.map { it.tasks["test"] })
}
```

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.

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.

JUnit 4.8 introduced the concept of categories for grouping JUnit 4 tests classes and methods. [13: The JUnit wiki contains a detailed description on how to work with JUnit categories: https://github.com/junit-team/junit/wiki/Categories.] 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 445. 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 446. 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. [14: The TestNG documentation contains more details about test groups: http://testng.org/doc/documentation-main.html#test-groups.] 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 447. Grouping TestNG tests

build.gradle

```
test {
    useTestNG {
        excludeGroups 'integrationTests'
        includeGroups 'unitTests'
    }
}
```

build.gradle.kts

```
tasks.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 448. Enabling JUnit Platform to run your tests

build.gradle

```
test {
    useJUnitPlatform()
}
```

build.gradle.kts

```
tasks.test {
    useJUnitPlatform()
}
```

See Test.useJUnitPlatform() for more details.

NOTE There are some known limitations of using JUnit 5 with Gradle, for example that tests in static nested classes won't be discovered and classes are still displayed by their class name instead of `@DisplayName`. These will be fixed in future version of Gradle. If you find more, please tell us at https://github.com/gradle/gradle/issues/new

Compiling and executing JUnit Jupiter tests

To enable JUnit Jupiter support in Gradle, all you need to do is add the following dependencies:

Example 449. JUnit Jupiter dependencies

build.gradle

```
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.1.0'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.1.0'
}
```

build.gradle.kts

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.1.0")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:5.1.0")
}
```

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 450. JUnit Vintage dependencies

build.gradle

```
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.1.0'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.1.0'
    testCompileOnly 'junit:junit:4.12'
    testRuntimeOnly 'org.junit.vintage:junit-vintage-engine:5.1.0'
}
```

build.gradle.kts

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.1.0")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:5.1.0")
    testCompileOnly("junit:junit:4.12")
    testRuntimeOnly("org.junit.vintage:junit-vintage-engine:5.1.0")
}
```

In this way, you can use `gradle test` to test JUnit 3/4 tests on JUnit Platform, without the need to rewrite them.

A sample of mixed tests can be found at `samples/testing/junitplatform/mix` in the '-all' distribution of Gradle.

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 451. 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")
    }
}
```

A test engine filtering sample can be found at `samples/testing/junitplatform/engine` in the '-all' distribution of Gradle.

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. [15: The TestNG documentation contains more details about test ordering when working with `testng.xml` files: http://testng.org/doc/documentation-main.html#testng-xml.] 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 452. 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 453. 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 taking these steps:

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 454. 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 'junit:junit:4.12'
}
```

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("junit:junit:4.12")
}
```

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*

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 455. Defining a working integration test task

build.gradle

```
task integrationTest(type: Test) {
    description = 'Runs integration tests.'
    group = 'verification'

    testClassesDirs = sourceSets.intTest.output.classesDirs
    classpath = sourceSets.intTest.runtimeClasspath
    shouldRunAfter test
}

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")
}

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.

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(org.gradle.api.specs.Spec) method. The following sample skips the `test` task if the project has a property called `mySkipTests`:

Example 456. Skipping the unit tests based on a project property

build.gradle

```
test.onlyIf { !project.hasProperty('mySkipTests') }
```

build.gradle.kts

```
tasks.test { onlyIf { !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 force tests to run in this situation 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.

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.

Usage

To use the Java plugin, include the following in your build script:

Example 457. Using the Java plugin

build.gradle

```
plugins {
    id 'java'
}
```

build.gradle.kts

```
plugins {
    java
}
```

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 458. 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 sec:specifying_multiple_filesthis section.

`java.outputDir` — `File`

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 459. Assembling a JAR for a source set

build.gradle

```
task intTestJar(type: Jar) {
    from sourceSets.intTest.output
}
```

build.gradle.kts

```
task<Jar>("intTestJar") {
    from(sourceSets["intTest"].output)
}
```

Generating Javadoc for a source set:

Example 460. Generating the Javadoc for a source set

build.gradle

```
task intTestJavadoc(type: Javadoc) {
    source sourceSets.intTest.allJava
}
```

build.gradle.kts

```
task<Javadoc>("intTestJavadoc") {
    source(sourceSets["intTest"].allJava)
}
```

Adding a test suite to run the tests in a source set:

Example 461. Running tests in a source set

build.gradle

```
task intTest(type: Test) {
    testClassesDirs = sourceSets.intTest.output.classesDirs
    classpath = sourceSets.intTest.runtimeClasspath
}
```

build.gradle.kts

```
task<Test>("intTest") {
    testClassesDirs = sourceSets["intTest"].output.classesDirs
    classpath = sourceSets["intTest"].runtimeClasspath
}
```

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` — *Copy*

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.

`uploadArchives` — *Upload*

Depends on: `jar`, and any other task that produces an artifact attached to the `archives` configuration

Uploads artifacts in the `archives` configuration — including the production JAR file — to the configured repositories.

`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 theeir `testRuntimeClasspath` configurations

  Performs a full build of the project and all projects which depend upon it.

`build`*`ConfigName`* — *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.

`upload`*`ConfigName`* — *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.

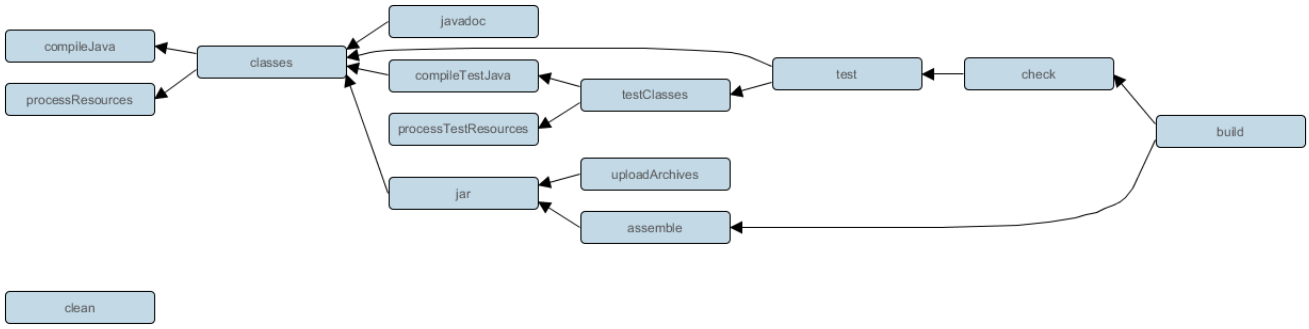


Figure 29. Java plugin - tasks

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.

`compile` *(Deprecated)*

Compile time dependencies. Superseded by `implementation`.

`implementation` *extends* `compile`

Implementation only dependencies.

`compileOnly`

Compile time only dependencies, not used at runtime.

`compileClasspath` *extends* `compile, compileOnly, implementation`

Compile classpath, used when compiling source. Used by task `compileJava`.

`annotationProcessor`

Annotation processors used during compilation.

`runtime` *(Deprecated) extends* `compile`

Runtime dependencies. Superseded by `runtimeOnly`.

`runtimeOnly`

Runtime only dependencies.

`runtimeClasspath` *extends* `runtimeOnly, runtime, implementation`

Runtime classpath contains elements of the implementation, as well as runtime only elements.

`testCompile` *(Deprecated) extends* `compile`

Additional dependencies for compiling tests. Superseded by `testImplementation`.

`testImplementation` *extends* `testCompile, implementation`

Implementation only dependencies for tests.

`testCompileOnly`

Additional dependencies only for compiling tests, not used at runtime.

`testCompileClasspath` *extends* `testCompile, testCompileOnly, testImplementation`

Test compile classpath, used when compiling test sources. Used by task `compileTestJava`.

`testRuntime` *(Deprecated) extends* `runtime, testCompile`

Additional dependencies for running tests only. Used by task `test`. Superseded by `testRuntimeOnly`.

`testRuntimeOnly` *extends* `runtimeOnly`

Runtime only dependencies for running tests. Used by task `test`.

`testRuntimeClasspath` *extends* `testRuntimeOnly, testRuntime, testImplementation`

Runtime classpath for running tests.

`archives`

Artifacts (e.g. jars) produced by this project. Used by tasks `uploadArchives`.

`default` *extends* `runtime`

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.

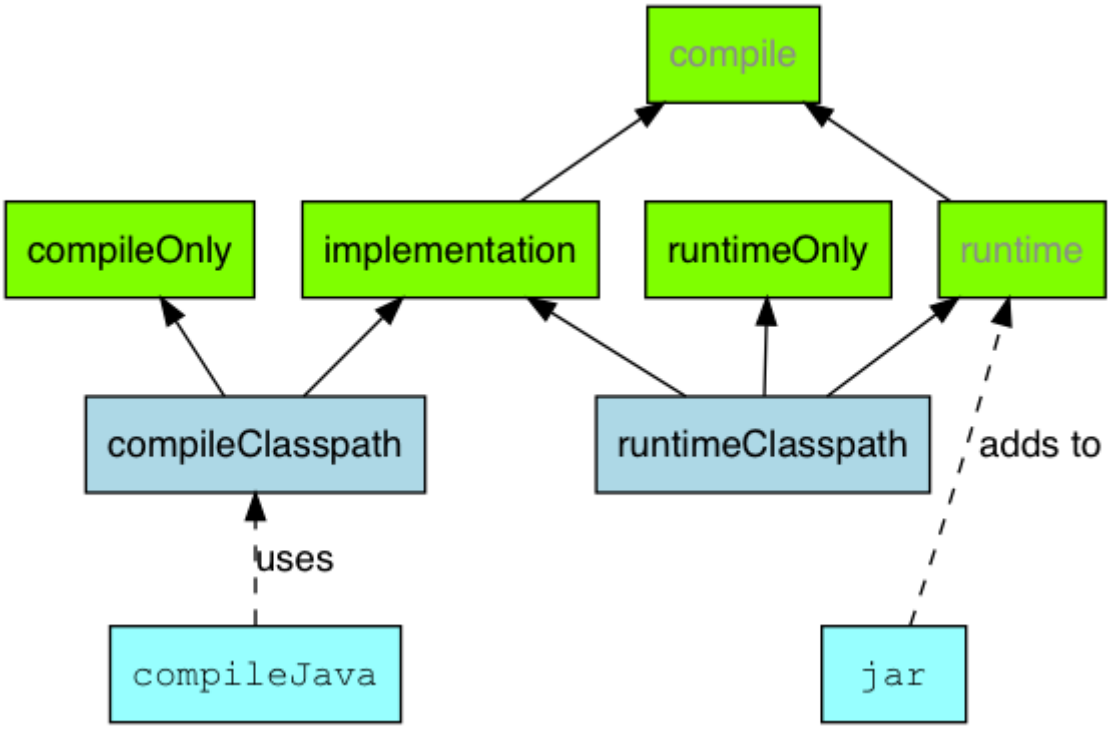


Figure 30. Java plugin - main source set dependency configurations

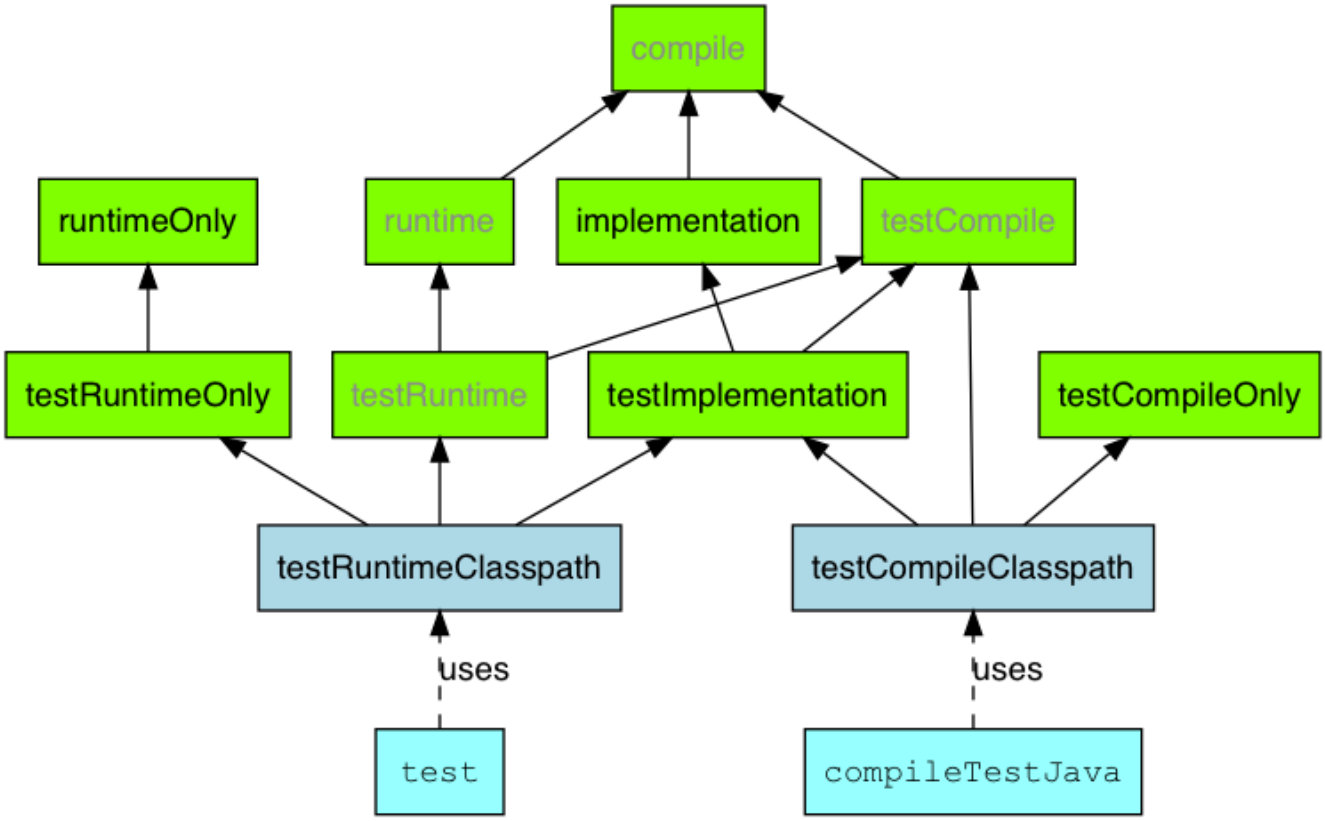


Figure 31. Java plugin - test source set dependency configurations

For each source set you add to the project, the Java plugins adds the following dependency configurations:

SourceSet dependency configurations

`sourceSetCompile` *(Deprecated)*

Compile time dependencies for the given source set. Superseded by `sourceSetImplementation`.

`sourceSetImplementation` *extends* `sourceSetCompile`

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* `compileSourceSetJava`

Compile classpath, used when compiling source. Used by `sourceSetCompile`, `sourceSetCompileOnly`, `sourceSetImplementation`.

`sourceSetAnnotationProcessor`

Annotation processors used during compilation of this source set.

`sourceSetRuntime`*(Deprecated)*

Runtime dependencies for the given source set. Used by `sourceSetCompile`. Superseded by `sourceSetRuntimeOnly`.

`sourceSetRuntimeOnly`

Runtime only dependencies for the given source set.

`sourceSetRuntimeClasspath` *extends* `sourceSetRuntimeOnly`, `sourceSetRuntime`, `sourceSetImplementation`

Runtime classpath contains elements of the implementation, as well as runtime only elements.

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.

Convention properties

The Java Plugin adds a number of convention properties to the project, shown below. You can use these properties in your build script as though they were properties of the project object.

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 convention properties

`(read-only) SourceSetContainer sourceSets`

Contains the project's source sets. Default value: Not null SourceSetContainer

`JavaVersion sourceCompatibility`

Java version compatibility to use when compiling Java source. Default value: version of the current JVM in use JavaVersion. Can also set using a String or a Number, e.g. `'1.5'` or `1.5`.

`JavaVersion targetCompatibility`

Java version to generate classes for. Default value: `sourceCompatibility`. Can also set using a String or Number, e.g. `'1.5'` or `1.5`.

`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.

These properties are provided by convention objects of type JavaPluginConvention, and

BasePluginConvention.

Javadoc

The `javadoc` task is an instance of Javadoc. It supports the core Javadoc options and the options of the standard doclet described in the reference documentation of the Javadoc executable. For a complete list of supported Javadoc options consult the API documentation of the following classes: CoreJavadocOptions and StandardJavadocDocletOptions.

Javadoc properties

`FileCollection classpath`
: Default value: `sourceSets.main.output` + `sourceSets.main.compileClasspath`

`FileTree source`
: Default value: `sourceSets.main.allJava`. Can set using anything described in Understanding implicit conversion to file collections.

`File destinationDir`
: Default value: `docsDir/javadoc`

`String title`
: Default value: The name and version of the project

Clean

The `clean` task is an instance of Delete. It simply removes the directory denoted by its `dir` property.

Clean properties

`File dir`
: Default value: `buildDir`

Resources

The Java plugin uses the Copy task for resource handling. It adds an instance for each source set in the project. You can find out more about the copy task in File copying in depth.

ProcessResources properties

`Object srcDirs`
: Default value: `sourceSet.resources`. Can set using anything described in Understanding implicit conversion to file collections.

`File destinationDir`
: Default value: `sourceSet.output.resourcesDir`. Can set using anything described in file paths in depth.

CompileJava

The Java plugin adds a JavaCompile instance for each source set in the project. Some of the most common configuration options are shown below.

Compile properties

`FileCollection classpath`

Default value: `sourceSet.compileClasspath`

`FileTree source`

Default value: `sourceSet.java`. Can set using anything described in Understanding implicit conversion to file collections.

`File destinationDir`

Default value: `sourceSet.java.outputDir`

By default, the Java compiler runs in the Gradle process. Setting `options.fork` to `true` causes compilation to occur in a separate process. In the case of the Ant javac task, this means that a new process will be forked for each compile task, which can slow down compilation. Conversely, Gradle's direct compiler integration (see above) will reuse the same compiler process as much as possible. In both cases, all fork options specified with `options.forkOptions` will be honored.

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.
- Since constants can be inlined, any change to a constant will result in Gradle recompiling all source files. For that reason, you should try to minimize the use of constants in your source code and replace them with static methods where possible.
- 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 desing 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.

Known issues

- If a compile task fails due to a compile error, it will do a full compilation again the next time it is invoked.
- 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.

Incremental annotation processing

Starting with Gradle 4.7, the incremental compiler also supports incremental annotation processing. 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 category separated by a comma.

Example: Registering incremental annotation processors

processor/src/main/resources/META-INF/gradle/incremental.annotation.processors

```
EntityProcessor,isolating
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/ServiceRegistryProcessor.java

```
@Override
public Set<String> getSupportedOptions() {
    return Collections.singleton("org.gradle.annotation.processing.aggregating");
}
```

Both categories have the following limitations:

- 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, Gradle will recompile all source files. See gradle/issues/4702

"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/EntityProcessor.java

```
Set<? extends Element> entities = roundEnv.getElementsAnnotatedWith(entityAnnotation);
for (Element entity : entities) {
    createRepository((TypeElement) entity);
}
```

"Isolating" processors have the following 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 ore 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/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();
```

"Aggregating" processors 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.

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.

Compile 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 462. 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")
}
```

Test

The `test` task is an instance of Test. It automatically detects and executes all unit tests in the `test` source set. It also generates a report once test execution is complete. JUnit and TestNG are both supported. Have a look at Test for the complete API.

See the Testing in Java & JVM projects chapter for more details.

Jar

The `jar` task creates a JAR file containing the class files and resources of the project. The JAR file is declared as an artifact in the `archives` dependency configuration. This means that the JAR is available in the classpath of a dependent project. If you upload your project into a repository, this JAR is declared as part of the dependency descriptor. You can learn more about how to work with archives in Archive creation in depth and artifact configurations in Legacy Publishing.

Manifest

Each jar or war object has a `manifest` property with a separate instance of Manifest. When the archive is generated, a corresponding `MANIFEST.MF` file is written into the archive.

Example 463. Customization of MANIFEST.MF

build.gradle

```
jar {
    manifest {
        attributes("Implementation-Title": "Gradle",
                   "Implementation-Version": version)
    }
}
```

build.gradle.kts

```
tasks.jar {
    manifest {
        attributes(
            "Implementation-Title" to "Gradle",
            "Implementation-Version" to version
        )
    }
}
```

You can create stand-alone instances of a `Manifest`. You can use that for example, to share manifest information between jars.

Example 464. Creating a manifest object.

build.gradle

```
ext.sharedManifest = manifest {
    attributes("Implementation-Title": "Gradle",
               "Implementation-Version": version)
}
task fooJar(type: Jar) {
    manifest = project.manifest {
        from sharedManifest
    }
}
```

build.gradle.kts

```
val sharedManifest = the<JavaPluginConvention>().manifest {
    attributes (
        "Implementation-Title" to "Gradle",
        "Implementation-Version" to version
    )
}

task<Jar>("fooJar") {
    manifest = project.the<JavaPluginConvention>().manifest {
        from(sharedManifest)
    }
}
```

You can merge other manifests into any `Manifest` object. The other manifests might be either described by a file path or, like in the example above, by a reference to another `Manifest` object.

Example 465. Separate MANIFEST.MF for a particular archive

build.gradle

```
task barJar(type: 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

```
task<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 by 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. The merge is not immediately triggered by the from statement. It is done lazily, either when generating the jar, or by calling `writeTo` or `effectiveManifest`

You can easily write a manifest to disk.

Example 466. Saving a MANIFEST.MF to disk

build.gradle

```
jar.manifest.writeTo("$buildDir/mymanifest.mf")
```

build.gradle.kts

```
tasks.getByName<Jar>("jar").manifest.writeTo("$buildDir/mymanifest.mf")
```

The Java Library Plugin

The Java Library plugin expands the capabilities of the Java plugin 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 467. 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 468. Declaring API and implementation dependencies

build.gradle

```
dependencies {
    api 'commons-httpclient:commons-httpclient:3.1'
    implementation 'org.apache.commons:commons-lang3:3.5'
}
```

build.gradle.kts

```
dependencies {
    api("commons-httpclient:commons-httpclient:3.1")
    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` configuration still exists but should not be used as it will not offer the guarantees that the `api` and `implementation` configurations provide.

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 `compile` scoped dependencies, while the runtime classpath adds the `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. 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.

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

```
// The following types can appear anywhere in the code
// but say nothing about API or implementation usage
import org.apache.commons.httpclient.*;
import org.apache.commons.httpclient.methods.*;
import org.apache.commons.lang3.exception.ExceptionUtils;
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) {
        GetMethod method = new GetMethod(url);
        try {
            int statusCode = doGet(method);
            return method.getResponseBody();

        } catch (Exception e) {
            ExceptionUtils.rethrow(e); // this dependency is internal only
        } finally {
            method.releaseConnection();
        }
        return null;
    }

    // GetMethod is used in a private method, so doesn't belong to the API
    private int doGet(GetMethod method) throws Exception {
        int statusCode = client.executeMethod(method);
        if (statusCode != HttpStatus.SC_OK) {
            System.err.println("Method failed: " + method.getStatusLine());
        }
        return statusCode;
    }
}
```

The *public* constructor of `HttpClientWrapper` uses `HttpClient` as a parameter, so it is exposed to consumers and therefore belongs to the API. Note that `GetMethod` is used in the signature of a *private* method, and so it doesn'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 `commons-httpclient` is an API dependency, whereas `commons-lang` is an implementation dependency. This conclusion translates into the following declaration in the build script:

Example 469. Declaring API and implementation dependencies

build.gradle

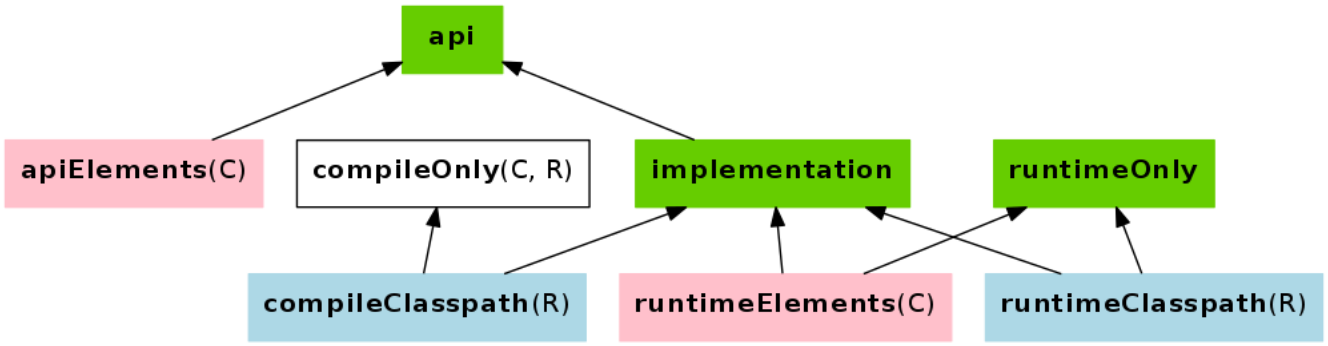
```
dependencies {
    api 'commons-httpclient:commons-httpclient:3.1'
    implementation 'org.apache.commons:commons-lang3:3.5'
}
```

build.gradle.kts

```
dependencies {
    api("commons-httpclient:commons-httpclient:3.1")
    implementation("org.apache.commons:commons-lang3:3.5")
}
```

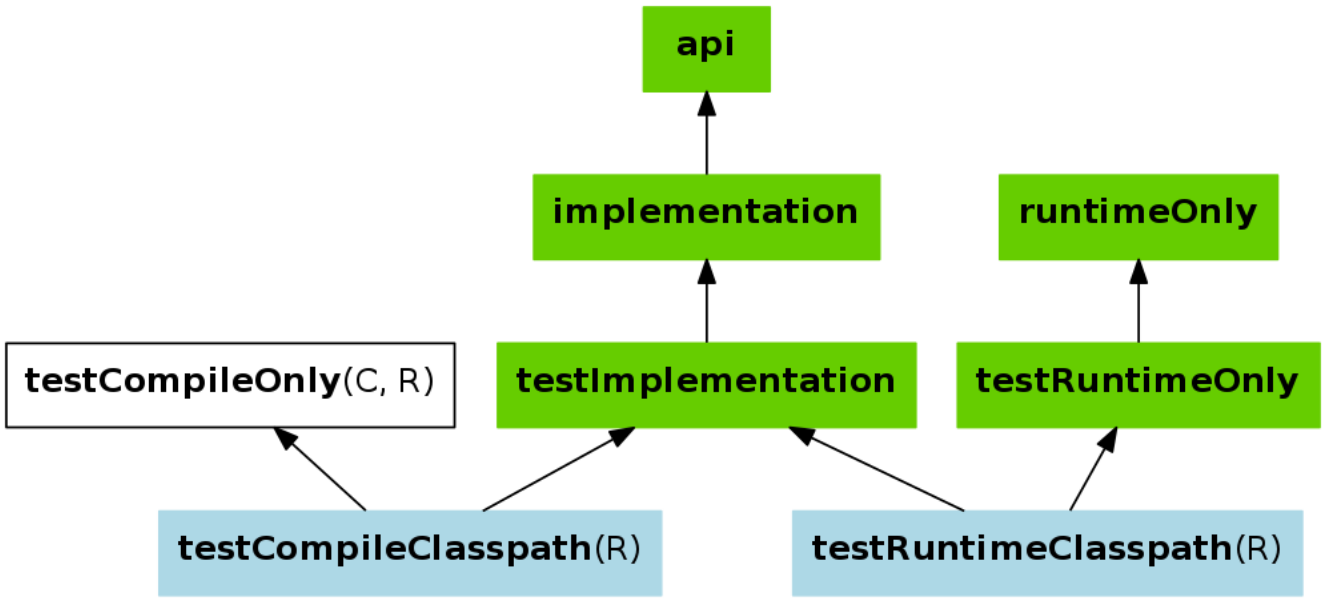
The Java Library plugin configurations

The following graph describes the main configurations 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
- The configurations in *white* are configurations inherited from the Java plugin

And the next graph describes the test configurations setup:



NOTE The *compile*, *testCompile*, *runtime* and *testRuntime* configurations inherited from the Java plugin are still available but are deprecated. You should avoid using them, as they are only kept for backwards compatibility.

The role of each configuration is described in the following tables:

Table 19. Java Library plugin - configurations used to declare dependencies

Configuration name	Role	Consumable?	Resolvable?	Description
`api`	Declaring API dependencies	no	no	This is where you should declare dependencies which are transitively exported to consumers, for compile.
`implementation`	Declaring implementation dependencies	no	no	This is where you should declare dependencies which are purely internal and not meant to be exposed to consumers.
`compileOnly`	Declaring compile only dependencies	yes	yes	This is where you should declare dependencies which are only required at compile time, but should not leak into the runtime. This typically includes dependencies which are shaded when found at runtime.
`runtimeOnly`	Declaring runtime dependencies	no	no	This is where you should declare dependencies which are only required at runtime, and not at compile time.
`testImplementation`	Test dependencies	no	no	This is where you should declare dependencies which are used to compile tests.

Configuration name	Role	Consumable?	Resolvable?	Description
`testCompileOnly`	Declaring test compile only dependencies	yes	yes	This is where you should 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 should declare dependencies which are only required at test runtime, and not at test compile time.

Table 20. 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 21. 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

Known issues

Compatibility with other plugins

At the moment the Java Library plugin is only wired to behave correctly with the `java` plugin. Other plugins, such as the Groovy plugin, may not behave correctly. In particular, if the Groovy plugin is used in addition to the `java-library` plugin, then consumers may not get the Groovy classes when

they consume the library. To workaround this, you need to explicitly wire the Groovy compile dependency, like this:

Example 470. Configuring the Groovy plugin to work with Java Library

a/build.gradle

```
configurations {
    apiElements {
        outgoing.variants.getByName('classes').artifact(
            file: compileGroovy.destinationDir,
            type: ArtifactTypeDefinition.JVM_CLASS_DIRECTORY,
            builtBy: compileGroovy)
    }
}
```

a/build.gradle.kts

```
configurations {
    apiElements {
        val compileGroovy = tasks.compileGroovy.get()
        outgoing.variants["classes"].artifact(mapOf(
            "file" to compileGroovy.destinationDir,
            "type" to ArtifactTypeDefinition.JVM_CLASS_DIRECTORY,
            "builtBy" to compileGroovy)
        )
    }
}
```

Increased memory usage for consumers

When a project uses the Java Library plugin, consumers will use the output classes directory of this project directly on their compile classpath, instead of the jar file if the project uses the Java plugin. 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.

The Java Library Distribution Plugin

NOTE	The Java library distribution plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

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 471. 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 472. Configure the distribution name

build.gradle

```
distributions {
    main {
        baseName = 'my-name'
    }
}
```

build.gradle.kts

```
distributions {
    main {
        baseName = "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 473. Include files in the distribution

build.gradle

```
distributions {
    main {
        baseName = 'my-name'
        contents {
            from 'src/dist'
        }
    }
}
```

build.gradle.kts

```
distributions {
    main {
        baseName = "my-name"
        contents {
            from("src/dist")
        }
    }
}
```

Dependency Management for Java Projects

This chapter explains how to apply basic dependency management concepts to Java-based projects. For a detailed introduction to dependency management, see Introduction to Dependency

Management.

Dissecting a typical build script

Let's have a look at a very simple build script for a Java-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 474. Dependency declarations for a Java-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 475. 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 Managing 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 476. 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 477. 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

Dependency configurations are also used to publish files. Gradle calls these files *publication artifacts*, or usually just *artifacts*. As a user you will need to tell Gradle where to publish the artifacts. You do this by declaring repositories for the `uploadArchives` task. Here's an example of publishing to a Maven repository:

Example 478. Publishing to a Maven repository

build.gradle

```
plugins {
    id 'maven'
}

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
        }
    }
}
```

build.gradle.kts

```
plugins {
    maven
}

tasks.getByName<Upload>("uploadArchives") {
    repositories.withGroovyBuilder {
        "mavenDeployer" {
            "repository"("url" to "file://localhost/tmp/myRepo/")
        }
    }
}
```

Now, when you run `gradle uploadArchives`, Gradle will build the JAR file, generate a `.pom` file and upload the artifacts.

To learn more about publishing artifacts, have a look at Legacy Publishing.

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()`. [16: 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]

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 479. Using an Ant task

build.gradle

```
task hello {
    doLast {
        String greeting = 'hello from Ant'
        ant.echo(message: greeting)
    }
}
```

build.gradle.kts

```
task("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 480. Passing nested text to an Ant task

build.gradle

```
task hello {
    doLast {
        ant.echo('hello from Ant')
    }
}
```

build.gradle.kts

```
task("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 481. Passing nested elements to an Ant task

build.gradle

```
task zip {
    doLast {
        ant.zip(destfile: 'archive.zip') {
            fileset(dir: 'src') {
                include(name: '**.xml')
                exclude(name: '**.java')
            }
        }
    }
}
```

build.gradle.kts

```
task("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 482. Using an Ant type

build.gradle

```
task 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

task("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 483. Using a custom Ant task

build.gradle

```
task check {
    doLast {
        ant.taskdef(resource: 'checkstyletask.properties') {
            classpath {
                fileset(dir: 'libs', includes: '*.jar')
            }
        }
        ant.checkstyle(config: 'checkstyle.xml') {
            fileset(dir: 'src')
        }
    }
}
```

build.gradle.kts

```
task("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 484. 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 485. Using a custom Ant task and dependency management together

build.gradle

```
task 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

```
task("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 486. 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 487. Task that depends on Ant target

build.gradle

```
ant.importBuild 'build.xml'

task intro(dependsOn: hello) {
    doLast {
        println 'Hello, from Gradle'
    }
}
```

build.gradle.kts

```
ant.importBuild("build.xml")

task("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 488. 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.getByName("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 489. Ant target that depends on Gradle task

build.gradle

```
ant.importBuild 'build.xml'

task intro {
    doLast {
        println 'Hello, from Gradle'
    }
}
```

build.gradle.kts

```
ant.importBuild("build.xml")

task("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 490. Renaming imported Ant targets

build.gradle

```
ant.importBuild('build.xml') { antTargetName ->
    'a-' + antTargetName
}
```

build.gradle.kts

```
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 491. 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 492. 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 493. 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 494. 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 22. Ant message priority mapping

Ant Message Priority	Gradle Log Level
VERBOSE	`DEBUG`
DEBUG	`DEBUG`
INFO	`INFO`
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 495. Fine tuning Ant logging

build.gradle

```
ant.lifecycleLogLevel = "INFO"

task hello {
    doLast {
        ant.echo(level: "info", message: "hello from info priority!")
    }
}
```

build.gradle.kts

```
ant.lifecycleLogLevel = AntBuilder.AntMessagePriority.INFO

task("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.

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 496. 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 23. ANTLR plugin - additional task dependencies

Task name	Depends on
`compileJava`	`generateGrammarSource`

Task name	Depends on
`compileTestJava`	`generateTestGrammarSource`
`compile__SourceSet__Java`	`generate__SourceSet__GrammarSource`

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 497. 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.

Convention properties

The ANTLR plugin does not add any convention properties.

Source set properties

The ANTLR plugin adds the following properties to each source set in the project.

`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.*

`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 498. 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.

Usage

To use the application plugin, include the following in your build script:

Example 499. 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 500. Configure the application main class

build.gradle

```
application {
    mainClassName = "org.gradle.sample.Main"
}
```

build.gradle.kts

```
application {
    mainClassName = "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 501. 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 502. Configure custom directory for start scripts

build.gradle

```
application {
    executableDir = "custom_bin_dir"
}
```

build.gradle.kts

```
application {
    executableDir = "custom_bin_dir"
}
```

The 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 24. Distribution content

Location	Content
(root dir)	`src/dist`
`lib`	All runtime dependencies and main source set class files.
`bin`	Start scripts (generated by `createStartScripts` 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 503. Include output from other tasks in the application distribution

build.gradle

```
task createDocs {
    def docs = file("$buildDir/docs")
    outputs.dir docs
    doLast {
        docs.mkdirs()
        new File(docs, "readme.txt").write("Read me!")
    }
}

distributions {
    main {
        contents {
            from(createDocs) {
                into "docs"
            }
        }
    }
}
```

build.gradle.kts

```
task("createDocs") {
    val docs = file("$buildDir/docs")
    outputs.dir(docs)
    doLast {
        docs.mkdirs()
        File(docs, "readme.txt").writeText("Read me!")
    }
}

distributions {
    main {
        contents {
            from(tasks["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.

Example: Automatically creating files for distribution

Output of `gradle distZip`

```
> gradle distZip
> Task :createDocs
> Task :compileJava
> Task :processResources NO-SOURCE
> Task :classes
> Task :jar
> Task :startScripts
> Task :distZip

BUILD SUCCESSFUL in 0s
5 actionable tasks: 5 executed
```

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.

Convention properties

The application plugin adds some properties to the project, which you can use to configure its behaviour. See the Project class in the API documentation.

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 504. 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 25. 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 505. 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 506. Customizing the HTML report

build.gradle

```
tasks.withType(Checkstyle) {
    reports {
        xml.enabled false
        html.enabled true
        html.stylesheet resources.text.fromFile('config/xsl/checkstyle-
custom.xsl')
    }
}
```

build.gradle.kts

```
tasks.withType<Checkstyle> {
    reports {
        xml.isEnabled = false
        html.isEnabled = true
        html.stylesheet = resources.text.fromFile("config/xsl/checkstyle-
custom.xsl")
    }
}
```

View a sample Checkstyle stylesheet.

The FindBugs Plugin

The FindBugs plugin performs quality checks on your project's Java source files using FindBugs and generates reports from these checks.

WARNING	Since FindBugs is unmaintained and does not support bytecode compiled for Java 9 and above, the FindBugs plugin has been deprecated and is scheduled to be removed in Gradle 6.0. Please consider using the SpotBugs plugin instead.

Usage

To use the FindBugs plugin, include the following in your build script:

Example 507. Using the FindBugs plugin

build.gradle

```
plugins {
    id 'findbugs'
}
```

build.gradle.kts

```
plugins {
    findbugs
}
```

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 Findbugs will run with the same Java version used to run Gradle.

Tasks

The FindBugs plugin adds the following tasks to the project:

`findbugsMain` — *FindBugs*

Depends on: `classes`

Runs FindBugs against the production Java source files.

`findbugsTest` — *FindBugs*

Depends on: `testClasses`

Runs FindBugs against the test Java source files.

`findbugsSourceSet` — *FindBugs*

Depends on: `sourceSetClasses`

Runs FindBugs against the given source set's Java source files.

The FindBugs plugin adds the following dependencies to tasks defined by the Java plugin.

Table 26. FindBugs plugin - additional task dependencies

Task name	Depends on
`check`	All FindBugs tasks, including `findbugsMain` and `findbugsTest`.

Dependency management

The FindBugs plugin adds the following dependency configurations:

Table 27. FindBugs plugin - dependency configurations

Name	Meaning
`findbugs`	The FindBugs libraries to use

Configuration

See the FindBugsExtension class in the API documentation.

Customizing the HTML report

The HTML report generated by the FindBugs task can be customized using a XSLT stylesheet, for example to highlight specific errors or change its appearance:

Example 508. Customizing the HTML report

build.gradle

```
tasks.withType(FindBugs) {
    reports {
        xml.enabled false
        html.enabled true
        html.stylesheet resources.text.fromFile('config/xsl/findbugs-
custom.xsl')
    }
}
```

build.gradle.kts

```
tasks.withType<FindBugs> {
    reports {
        xml.isEnabled = false
        html.isEnabled = true
        html.stylesheet = resources.text.fromFile("config/xsl/findbugs-
custom.xsl")
    }
}
```

View a sample FindBugs stylesheet.

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 509. 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 that depends on the `test` task. The report is available at `$buildDir/reports/jacoco/test`. By default, a HTML report is generated.

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 510. Configuring JaCoCo plugin settings

build.gradle

```
jacoco {
    toolVersion = "0.8.2"
    reportsDir = file("$buildDir/customJacocoReportDir")
}
```

build.gradle.kts

```
jacoco {
    toolVersion = "0.8.2"
    reportsDir = file("$buildDir/customJacocoReportDir")
}
```

Table 28. Gradle defaults for JaCoCo properties

Property	Gradle default
reportsDir	`$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.

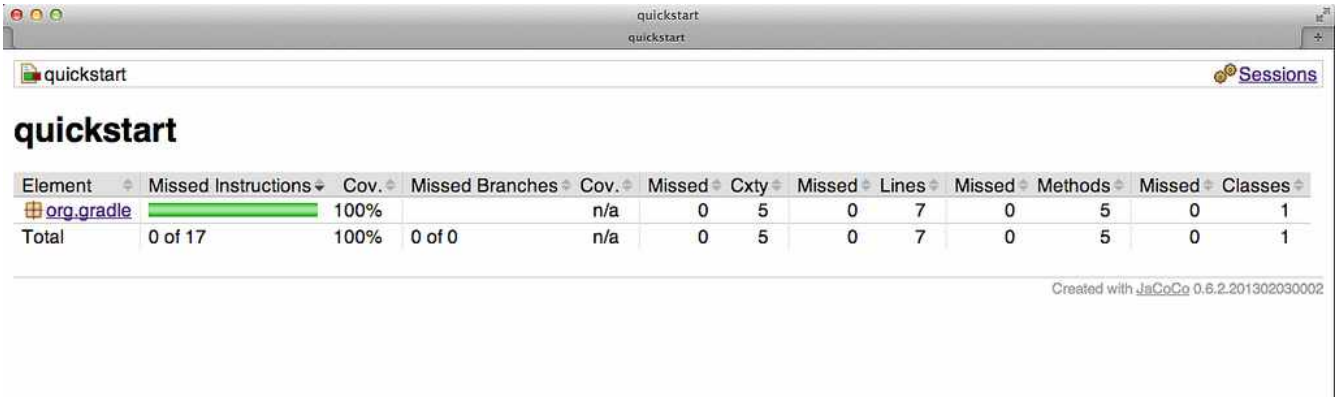
Example 511. Configuring test task

build.gradle

```
jacocoTestReport {
    reports {
        xml.enabled false
        csv.enabled false
        html.destination file("${buildDir}/jacocoHtml")
    }
}
```

build.gradle.kts

```
tasks.jacocoTestReport {
    reports {
        xml.isEnabled = false
        csv.isEnabled = false
        html.destination = file("${buildDir}/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 512. 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 {
            enabled = false
            element = "CLASS"
            includes = listOf("org.gradle.*")

            limit {
                counter = "LINE"
                value = "TOTALCOUNT"
                maximum = "0.3".toBigDecimal()
            }
        }
    }
}
```

NOTE The code for this example can be found at `samples/testing/jacoco/quickstart` in the '-all' distribution of Gradle.

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 513. Configuring test task

build.gradle

```
test {
    jacoco {
        destinationFile = file("$buildDir/jacoco/jacocoTest.exec")
        classDumpDir = file("$buildDir/jacoco/classpathdumps")
    }
}
```

build.gradle.kts

```
tasks.test {
    extensions.configure(JacocoTaskExtension::class) {
        destinationFile = file("$buildDir/jacoco/jacocoTest.exec")
        classDumpDir = file("$buildDir/jacoco/classpathdumps")
    }
}
```

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

```
test {
    jacoco {
        enabled = true
        destinationFile = file("$buildDir/jacoco/$name.exec")
        includes = []
        excludes = []
        excludeClassLoaders = []
        includeNoLocationClasses = false
        sessionId = "<auto-generated value>"
        dumpOnExit = true
        classDumpDir = null
        output = Output.FILE
        address = "localhost"
        port = 6300
        jmx = false
    }
}
```

```
tasks.getByName<Test>("test") {
    extensions.configure<JacocoTaskExtension::class) {
        isEnabled = true
        destinationFile = file("$buildDir/jacoco/$name.exec")
        includes = listOf()
        excludes = listOf()
        excludeClassLoaders = listOf()
        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 514. Using application plugin to generate code coverage data

build.gradle

```
plugins {
    id 'application'
    id 'jacoco'
}

application {
    mainClassName = 'org.gradle.MyMain'
}

jacoco {
    applyTo run
}

task applicationCodeCoverageReport(type:JacocoReport) {
    executionData run
    sourceSets sourceSets.main
}
```

build.gradle.kts

```
plugins {
    application
    jacoco
}

application {
    mainClassName = "org.gradle.MyMain"
}

jacoco {
    applyTo(tasks.run.get())
}

task<JacocoReport>("applicationCodeCoverageReport") {
    executionData(tasks.run.get())
    sourceSets(sourceSets.main.get())
}
```

NOTE The code for this example can be found at `samples/testing/jacoco/application` in the '-all' distribution of Gradle.

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 29. 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.

The JDepend Plugin

The JDepend plugin performs quality checks on your project's source files using JDepend and generates reports from these checks.

WARNING	Since JDepend is unmaintained and does not support bytecode compiled for Java 8 and above, the JDepend plugin has been deprecated and is scheduled to be removed in Gradle 6.0.

Usage

To use the JDepend plugin, include the following in your build script:

Example 515. Using the JDepend plugin

build.gradle

```
plugins {
    id 'jdepend'
}
```

build.gradle.kts

```
plugins {
    jdepend
}
```

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 JDepend will run with the same Java version used to run Gradle.

Tasks

The JDepend plugin adds the following tasks to the project:

`jdependMain` — *JDepend*

Depends on: `classes`

Runs JDepend against the production Java source files.

`jdependTest` — *JDepend*

Depends on: `testClasses`

Runs JDepend against the test Java source files.

`jdependSourceSet` — *JDepend*

Depends on: `sourceSetClasses`

Runs JDepend against the given source set's Java source files.

The JDepend plugin adds the following dependencies to tasks defined by the Java plugin.

Additional task dependencies

`check`

All JDepend tasks, including `jdependMain` and `jdependTest`.

Dependency management

The JDepend plugin adds the following dependency configurations:

Dependency configurations

`jdepend`
: The JDepend libraries to use

Configuration

See the JDependExtension class in the API documentation.

The OSGi Plugin

CAUTION	This plugin is deprecated and will be removed in the next major Gradle release. New builds should not use this plugin. The OSGi plugin makes heavy use of the BND tool. A separate plugin implementation is maintained by the BND authors that has more advanced features.

The OSGi plugin provides a factory method to create an OsgiManifest object. `OsgiManifest` extends Manifest. To learn more about generic manifest handling, see more about Java manifests. If the Java plugins is applied, the OSGi plugin replaces the manifest object of the default jar with an `OsgiManifest` object. The replaced manifest is merged into the new one.

Usage

To use the OSGi plugin, include the following in your build script:

Example 516. Using the OSGi plugin

build.gradle

```
plugins {
    id 'osgi'
}
```

build.gradle.kts

```
plugins {
    osgi
}
```

Implicitly applied plugins

Applies the Java base plugin.

Tasks

The OSGi plugin adds the following tasks to the project:

`osgiClasses` — *Sync*

Depends on: `classes`

Copies all classes from the main source set to a single directory that is processed by BND.

Convention object

The OSGi plugin adds the following convention object: OsgiPluginConvention

Convention properties

The OSGi plugin does not add any convention properties to the project.

Convention methods

The OSGi plugin adds the following methods. For more details, see the API documentation of the convention object.

Table 30. OSGi methods

Method	Return Type	Description
osgiManifest()	OsgiManifest	Returns an OsgiManifest object.
osgiManifest(Closure cl)	OsgiManifest	Returns an OsgiManifest object configured by the closure.

The classes in the classes dir are analyzed regarding their package dependencies and the packages they expose. Based on this the *Import-Package* and the *Export-Package* values of the OSGi Manifest are calculated. If the classpath contains jars with an OSGi bundle, the bundle information is used to specify version information for the *Import-Package* value. Beside the explicit properties of the `OsgiManifest` object you can add instructions.

Example 517. Configuration of OSGi MANIFEST.MF file

build.gradle

```
jar {
    manifest { // the manifest of the default jar is of type OsgiManifest
        name = 'overwrittenSpecialOsgiName'
        instruction 'Private-Package',
            'org.mycomp.package1',
            'org.mycomp.package2'
        instruction 'Bundle-Vendor', 'MyCompany'
        instruction 'Bundle-Description', 'Platform2: Metrics 2 Measures
Framework'
        instruction 'Bundle-DocURL', 'http://www.mycompany.com'
    }
}
task fooJar(type: Jar) {
    manifest = osgiManifest {
        instruction 'Bundle-Vendor', 'MyCompany'
    }
}
```

build.gradle.kts

```
tasks.withType<Jar> {
    manifest {
        // the manifest of the default jar is of type OsgiManifest
        (manifest as? OsgiManifest)?.apply {
            name = "overwrittenSpecialOsgiName"
            instruction("Private-Package",
                "org.mycomp.package1",
                "org.mycomp.package2")
            instruction("Bundle-Vendor", "MyCompany")
            instruction("Bundle-Description", "Platform2: Metrics 2 Measures
Framework")
            instruction("Bundle-DocURL", "http://www.mycompany.com")
        }
    }
}
task<Jar>("fooJar") {
    manifest = osgi.osgiManifest {
        instruction("Bundle-Vendor", "MyCompany")
    }
}
```

The first argument of the instruction call is the key of the property. The other arguments form the value. To learn more about the available instructions have a look at the BND tool.

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 518. 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 31. 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 32. PMD plugin - dependency configurations

Name	Meaning
`pmd`	The PMD libraries to use

Configuration

See the PmdExtension class in the API documentation.

Java Web Projects

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 519. 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 32. War plugin - tasks

Dependency management

The War plugin adds two dependency configurations:

- `providedCompile`
- `providedRuntime`

These two configurations have the same scope as the respective `compile` and `runtime` configurations, except that they are not added to the WAR archive.

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 `compile` 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

`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.

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` [17: The `runtime` configuration extends the `compile` configuration.] 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 520. Customization of war plugin

build.gradle

```
configurations {
   moreLibs
}

repositories {
   flatDir { dirs "lib" }
   jcenter()
}

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:4.12"
    moreLibs ":otherLib:1.0"
}

war {
    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") }
    jcenter()
}

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:4.12")
    moreLibs(":otherLib:1.0")
}

tasks.war {
    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.

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 521. 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

`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.

These properties are provided by a EarPluginConvention convention object.

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 522. 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 {
    appDirName '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")
}

ear {
    appDirName = "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 { // 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.

Building Play applications

NOTE	Support for building Play applications is currently incubating. Please be aware that the DSL, APIs and other configuration may change in later Gradle versions.

Play is a modern web application framework. The Play plugin adds support for building, testing and running Play applications with Gradle.

The Play plugin makes use of the Gradle software model.

Usage

To use the Play plugin, include the following in your build script to apply the `play` plugin and add the Lightbend repositories:

Example: Using the Play plugin

build.gradle

```
plugins {
    id 'play'
}

repositories {
    jcenter()
    maven {
        name "lightbend-maven-release"
        url "https://repo.lightbend.com/lightbend/maven-releases"
    }
    ivy {
        name "lightbend-ivy-release"
        url "https://repo.lightbend.com/lightbend/ivy-releases"
        layout "ivy"
    }
}
```

Note that defining the Lightbend repositories is necessary. In future versions of Gradle, this will be replaced with a more convenient syntax.

Limitations

The Play plugin currently has a few limitations.

- Gradle does not yet support aggregate reverse routes introduced in Play 2.4.x.
- A given project may only define a single Play application. This means that a single project cannot build more than one Play application. However, a multi-project build can have many projects that each define their own Play application.
- Play applications can only target a single "platform" (combination of Play, Scala and Java version) at a time. This means that it is currently not possible to define multiple variants of a Play application that, for example, produce jars for both Scala 2.10 and 2.11. This limitation may be lifted in future Gradle versions.
- Support for generating IDE configurations for Play applications is limited to IDEA.

Software Model

The Play plugin uses a *software model* to describe a Play application and how to build it. The Play software model extends the base Gradle software model to add support for building Play applications. A Play application is represented by a PlayApplicationSpec component type. The plugin automatically creates a single PlayApplicationBinarySpec instance when it is applied. Additional Play components cannot be added to a project.

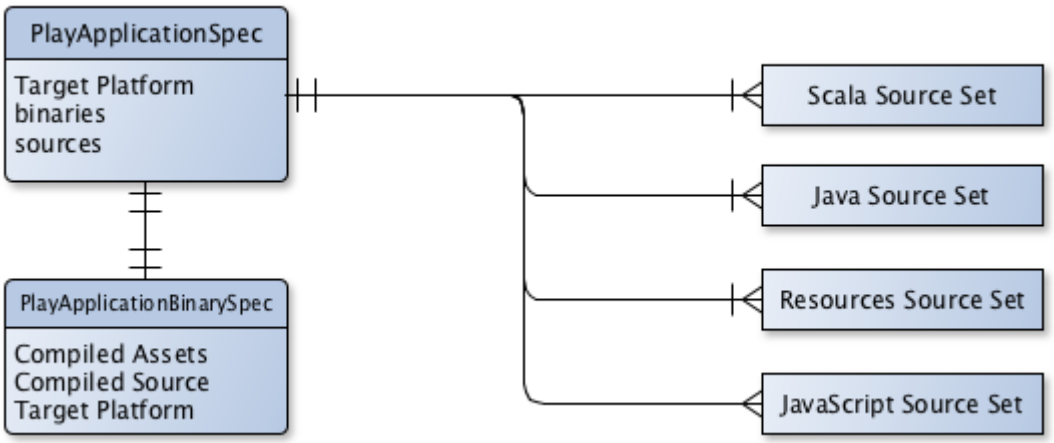


Figure 33. Play plugin - software model

The Play application component

A Play application component describes the application to be built and consists of several configuration elements. One type of element that describes the application are the source sets that define where the application controller, route, template and model class source files should be found. These source sets are logical groupings of files of a particular type and a default source set for each type is created when the `play` plugin is applied.

Table 33. Default Play source sets

Source Set	Type	Directory	Filters
java	JavaSourceSet	app	**/*.java

Source Set	Type	Directory	Filters
scala	ScalaLanguageSourceSet	app	**/*.scala
routes	RoutesSourceSet	conf	routes, *.routes
twirlTemplates	TwirlSourceSet	app	**/*.scala.*
javaScript	JavaScriptSourceSet	app/assets	**/*.js

These source sets can be configured or additional source sets can be added to the Play component. See Configuring Play for further information.

Another element of configuring a Play application is the *platform*. To build a Play application, Gradle needs to understand which versions of Play, Scala and Java to use. The Play component specifies this requirement as a PlayPlatform. If these values are not configured, a default version of Play, Scala and Java will be used. See Targeting a certain version of Play for information on configuring the Play platform.

Note that only a single platform can be specified for a given Play component. This means that only a single version of Play, Scala and Java can be used to build a Play component. In other words, a Play component can only produce one set of outputs, and those outputs will be built using the versions specified by the platform configured on the component.

The Play application binary

A Play application component is compiled and packaged to produce a set of outputs which are represented by a PlayApplicationBinarySpec. The Play binary specifies the jar files produced by building the component as well as providing elements by which additional content can be added to those jar files. It also exposes the tasks involved in building the component and creating the binary.

See Configuring Play for examples of configuring the Play binary.

Project Layout

The Play plugin follows the typical Play application layout. You can configure source sets to include additional directories or change the defaults.

```
├─── app                → Application source code.
│    ├─── assets          → Assets that require compilation.
│    │    └─── javascripts → JavaScript source code to be minified.
│    ├─── controllers     → Application controller source code.
│    ├─── models          → Application business source code.
│    └─── views           → Application UI templates.
├─── build.gradle        → Your project's build script.
├─── conf                → Main application configuration file and routes files.
├─── public              → Public assets.
│    ├─── images          → Application image files.
│    ├─── javascripts     → Typically JavaScript source code.
│    └─── stylesheets     → Typically CSS source code.
└─── test                → Test source code.
```

Tasks

The Play plugin hooks into the normal Gradle lifecycle tasks such as `assemble`, `check` and `build`, but it also adds several additional tasks which form the lifecycle of a Play project:

Play Plugin — lifecycle tasks

`playBinary` — *Task*

Depends on: All compile tasks for source sets added to the Play application.

Performs a build of just the Play application.

`dist` — *Task*

Depends on: `createPlayBinaryZipDist`, `createPlayBinaryTarDist`

Assembles the Play distribution.

`stage` — *Task*

Depends on: `stagePlayBinaryDist`

Stages the Play distribution.

The plugin also provides tasks for running, testing and packaging your Play application:

Play Plugin — running and testing tasks

`runPlayBinary` — *PlayRun*

Depends on: `playBinary` to build Play application.

Runs the Play application for local development. See how this works with continuous build.

`testPlayBinary` — *Test*

Depends on: `playBinary` to build Play application and `compilePlayBinaryTests`.

Runs JUnit/TestNG tests for the Play application.

For the different types of sources in a Play application, the plugin adds the following compilation tasks:

Play Plugin — source set tasks

`compilePlayBinaryScala` — *PlatformScalaCompile*

Depends on: Scala and Java

Compiles all Scala and Java sources defined by the Play application.

`compilePlayBinaryPlayTwirlTemplates` — *TwirlCompile*

Depends on: Twirl templates

Compiles Twirl templates with the Twirl compiler. Gradle supports all of the built-in Twirl template formats (HTML, XML, TXT and JavaScript). Twirl templates need to match the pattern `*.scala.*`.

`compilePlayBinaryPlayRoutes` — *RoutesCompile*

Depends on: Play Route files

Compiles routes files into Scala sources.

`minifyPlayBinaryJavaScript` — *JavaScriptMinify*

Depends on: JavaScript files

Minifies JavaScript files with the Google Closure compiler.

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
------------------------------------------------------------

Play Application 'play'
-----------------------

Source sets
    Java source 'play:java'
        srcDir: app
        includes: **/*.java
    JavaScript source 'play:javaScript'
        srcDir: app/assets
        includes: **/*.js
    JVM resources 'play:resources'
        srcDir: conf
    Routes source 'play:routes'
        srcDir: conf
        includes: routes, *.routes
    Scala source 'play:scala'
        srcDir: app
        includes: **/*.scala
    Twirl template source 'play:twirlTemplates'
        srcDir: app
        includes: **/*.scala.*

Binaries
    Play Application Jar 'play:binary'
        build using task: :playBinary
        target platform: Play Platform (Play 2.6.15, Scala: 2.12, Java: Java SE 8)
        toolchain: Default Play Toolchain
        classes dir: build/playBinary/classes
        resources dir: build/playBinary/resources
        JAR file: build/playBinary/lib/basic.jar

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
```

Running a Play application

The `runPlayBinary` task starts the Play application under development. During development it is

beneficial to execute this task as a continuous build. Continuous build is a generic feature that supports automatically re-running a build when inputs change. The `runPlayBinary` task is "continuous build aware" in that it behaves differently when run as part of a continuous build.

When not run as part of a continuous build, the `runPlayBinary` task will *block* the build. That is, the task will not complete as long as the application is running. When running as part of a continuous build, the task will start the application if not running and otherwise propagate any changes to the code of the application to the running instance. This is useful for quickly iterating on your Play application with an edit->rebuild->refresh cycle. Changes to your application will not take affect until the end of the overall build.

To enable continuous build, run Gradle with `-t runPlayBinary` or `--continuous runPlayBinary`.

Users of Play used to such a workflow with Play's default build system should note that compile errors are handled differently. If a build failure occurs during a continuous build, the Play application will not be reloaded. Instead, you will be presented with an exception message. The exception message will only contain the overall cause of the build failure. More detailed information will only be available from the console.

Configuring a Play application

Targeting a certain version of Play

By default, Gradle uses Play 2.6.15, Scala 2.12 and the version of Java used to start the build. A Play application can select a different version by specifying a target PlayApplicationSpec.platform(java.lang.Object) on the Play application component.

Example: Selecting a version of the Play Framework

build.gradle

```
model {
    components {
        play {
            platform play: '2.6.15', scala: '2.12', java: '1.8'
            injectedRoutesGenerator = true
        }
    }
}
```

The following versions of Play and Scala are supported:

Table 34. Play supported versions

Play	Scala	Java
2.6.x	2.11 and 2.12	1.8
2.5.x	2.11	1.8
2.4.x	2.10 and 2.11	1.8

Play	Scala	Java
2.3.x	2.10 and 2.11	1.6, 1.7 and 1.8

Adding dependencies

You can add compile, test and runtime dependencies to a Play application through Configuration created by the Play plugin.

If you are coming from SBT, the Play SBT plugin provides short names for common dependencies. For instance, if your project has a dependency on `ws`, you will need to add a dependency to `com.typesafe.play:play-ws_2.11:2.3.9` where `2.11` is your Scala version and `2.3.9` is your Play framework version.

Other dependencies that have short names, such as `jacksons` may actually be multiple dependencies. For those dependencies, you will need to work out the dependency coordinates from a dependency report.

- `play` is used for compile time dependencies.
- `playTest` is used for test compile time dependencies.
- `playRun` is used for run time dependencies.

Example: Adding dependencies to a Play application

build.gradle

```
dependencies {
    play "commons-lang:commons-lang:2.6"
    play "com.typesafe.play:play-guice_2.12:2.6.15"
    play "ch.qos.logback:logback-classic:1.2.3"
}
```

Play 2.6 has a more modular architecture and, because of that, you may need to add some dependencies manually. For example, Guice support was moved to a separated module. Considering the following definition for a Play 2.6 project:

Example: A Play 2.6 project

build.gradle

```
model {
    components {
        play {
            platform play: '2.6.7', scala: '2.12', java: '1.8'
            injectedRoutesGenerator = true
        }
    }
}
```

You can add Guice dependency like:

Example: Adding Guice dependency in Play 2.6 project

build.gradle

```
dependencies {
    play "com.typesafe.play:play-guice_2.12:2.6.7"
}
```

Of course, pay attention to keep the Play version and Scala version for the dependency consistent with the platform versions.

Configuring the default source sets

You can further configure the default source sets to do things like add new directories, add filters, etc.

Example: Configuring extra source sets to a Play application

build.gradle

```
model {
    components {
        play {
            sources {
                java {
                    source.srcDir "additional/java"
                }
                javaScript {
                    source {
                        srcDir "additional/javascript"
                        exclude "**/old_*.js"
                    }
                }
            }
        }
    }
}
```

Adding extra source sets

If your Play application has additional sources that exist in non-standard directories, you can add extra source sets that Gradle will automatically add to the appropriate compile tasks.

Example: Adding extra source sets to a Play application

build.gradle

```
model {
    components {
        play {
            sources {
                extraJava(JavaSourceSet) {
                    source.srcDir "extra/java"
                }
                extraTwirl(TwirlSourceSet) {
                    source.srcDir "extra/twirl"
                }
                extraRoutes(RoutesSourceSet) {
                    source.srcDir "extra/routes"
                }
            }
        }
    }
}
```

Configuring compiler options

If your Play application requires additional Scala compiler flags, you can add these arguments directly to the Scala compiler task.

Example: Configuring Scala compiler options

build.gradle

```
model {
    components {
        play {
            binaries.all {
                tasks.withType(PlatformScalaCompile) {
                    scalaCompileOptions.additionalParameters = ["-feature", "
-language:implicitConversions"]
                }
            }
        }
    }
}
```

Configuring routes style

NOTE	The injected router is only supported in Play Framework 2.4 or better.

If your Play application's router uses dependency injection to access your controllers, you'll need to configure your application to *not* use the default static router. Under the covers, the Play plugin is using the `InjectedRoutesGenerator` instead of the default `StaticRoutesGenerator` to generate the

router classes.

Example: Configuring routes style

build.gradle

```
model {
    components {
        play {
            injectedRoutesGenerator = true
        }
    }
}
```

Configuring Twirl templates

A custom Twirl template format can be configured independently for each Twirl source set. See the TwirlSourceSet for an example.

Injecting a custom asset pipeline

Gradle Play support comes with a simplistic asset processing pipeline that minifies JavaScript assets. However, many organizations have their own custom pipeline for processing assets. You can easily hook the results of your pipeline into the Play binary by utilizing the PublicAssets property on the binary.

Example: Configuring a custom asset pipeline

build.gradle

```
model {
    components {
        play {
            binaries.all { binary ->
                tasks.create("addCopyrightToPlay${binary.name.capitalize()}Assets",
AddCopyrights) { copyrightTask ->
                    source "raw-assets"
                    copyrightFile = project.file('copyright.txt')
                    destinationDir = project.file("${buildDir}/play${binary.name
.capitalize()}/addCopyRights")

                    // Hook this task into the binary
                    binary.assets.addAssetDir destinationDir
                    binary.assets.builtBy copyrightTask
                }
            }
        }
    }
}

class AddCopyrights extends SourceTask {
    @InputFile
    File copyrightFile

    @OutputDirectory
    File destinationDir

    @TaskAction
    void generateAssets() {
        String copyright = copyrightFile.text
        getSource().each { File file ->
            File outputFile = new File(destinationDir, file.name)
            outputFile.text = "${copyright}\n${file.text}"
        }
    }
}
```

Multi-project Play applications

Play applications can be built in multi-project builds as well. Simply apply the `play` plugin in the appropriate subprojects and create any project dependencies on the `play` configuration.

Example: Configuring dependencies on Play subprojects

build.gradle

```
dependencies {
    play project(":admin")
    play project(":user")
    play project(":util")
}
```

See the `play/multiproject` sample provided in the Gradle distribution for a working example.

Packaging a Play application for distribution

Gradle provides the capability to package your Play application so that it can easily be distributed and run in a target environment. The distribution package (zip file) contains the Play binary jars, all dependencies, and generated scripts that set up the classpath and run the application in a Play-specific Netty container.

The distribution can be created by running the `dist` lifecycle task and places the distribution in the `$buildDir/distributions` directory. Alternatively, one can validate the contents by running the `stage` lifecycle task which copies the files to the `$buildDir/stage` directory using the layout of the distribution package.

Play Plugin — distribution tasks

`createPlayBinaryStartScripts` — *CreateStartScripts*

Generates scripts to run the Play application distribution.

`stagePlayBinaryDist` — *Copy*

Depends on: `playBinary`, `createPlayBinaryStartScripts`

Copies all jar files, dependencies and scripts into a staging directory.

`createPlayBinaryZipDist` — *Zip*

Bundles the Play application as a standalone distribution packaged as a zip.

`createPlayBinaryTarDist` — *Tar*

Bundles the Play application as a standalone distribution packaged as a tar.

`stage` — *Task*

Depends on: `stagePlayBinaryDist`

Lifecycle task for staging a Play distribution.

`dist` — *Task*

Depends on: `createPlayBinaryZipDist`, `createPlayBinaryTarDist`

Lifecycle task for creating a Play distribution.

Adding additional files to your Play application distribution

You can add additional files to the distribution package using the Distribution API.

Example: Add extra files to a Play application distribution

build.gradle

```
model {
    distributions {
        playBinary {
            contents {
                from("README.md")
                from("scripts") {
                    into "bin"
                }
            }
        }
    }
}
```

Building a Play application with an IDE

If you want to generate IDE metadata configuration for your Play project, you need to apply the appropriate IDE plugin. Gradle supports generating IDE metadata for IDEA only for Play projects at this time.

To generate IDEA's metadata, apply the `idea` plugin along with the `play` plugin.

Example: Applying both the Play and IDEA plugins

build.gradle

```
plugins {
    id 'play'
    id 'idea'
}
```

Source code generated by routes and Twirl templates cannot be generated by IDEA directly, so changes made to those files will not affect compilation until the next Gradle build. You can run the Play application with Gradle in continuous build to automatically rebuild and reload the application whenever something changes.

Resources

For additional information about developing Play applications:

- Play types in the Gradle DSL Guide:
  - PlayApplicationBinarySpec

    - PlayApplicationSpec
    - PlayPlatform
    - JvmClasses
    - PublicAssets
    - PlayDistributionContainer
    - JavaScriptMinify
    - PlayRun
    - RoutesCompile
    - TwirlCompile
- Play Framework Documentation.

Scala Projects

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.

Usage

To use the Scala plugin, include the following in your build script:

Example 523. 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.

`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 Scala plugin adds the following dependencies to tasks added by the Java plugin.

Table 35. Scala plugin - additional task dependencies

Task name	Depends on
`classes`	`compileScala`
`testClasses`	`compileTestScala`
`__sourceSet__Classes`	`compile__SourceSet__Scala`

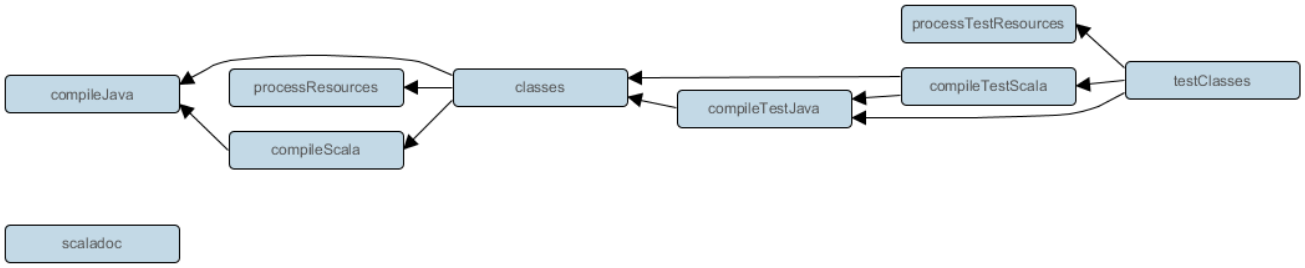


Figure 34. Scala plugin - tasks

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 524. 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: See Automatic configuration of Scala classpath.]

If Scala is used for production code, the `scala-library` dependency should be added to the `compile` configuration:

Example 525. 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:4.12'
}
```

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:4.12")
}
```

If Scala is only used for test code, the `scala-library` dependency should be added to the `testCompile` configuration:

Example 526. 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 add an explicit dependency like `"com.typesafe.zinc:zinc:0.3.6"` to the `zinc` configuration. Gradle supports version 0.3.0 of Zinc and above; however, due to a regression in the Zinc compiler, versions 0.3.2 through 0.3.5.2 cannot be used.

Example 527. Declaring a version of the Zinc compiler to use

build.gradle

```
dependencies {
    zinc 'com.typesafe.zinc:zinc:0.3.9'
}
```

build.gradle.kts

```
dependencies {
    zinc("com.typesafe.zinc:zinc:0.3.9")
}
```

It is important to take care when declaring your `scala-library` dependency. 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 adding a compatible version of `scala-library` for you, but over-broad dependency resolution rules could force an incompatible version to be used instead.

For example, using `configurations.all` to force a particular version of `scala-library` would also override the version used by the Zinc compiler:

Example 528. Forcing a scala-library dependency for all configurations

build.gradle

```
configurations.all {
    resolutionStrategy.force "org.scala-lang:scala-library:2.11.12"
}
```

build.gradle.kts

```
configurations.all {
    resolutionStrategy.force("org.scala-lang:scala-library:2.11.12")
}
```

The best way to avoid this problem is to be more selective when configuring the `scala-library` dependency (such as not using a `configuration.all` rule or using a conditional to prevent the rule from being applied to the `zinc` configuration). Sometimes this rule may come from a plugin or other code that you do not have control over. In such a case, you can force a correct version of the library

on the `zinc` configuration only:

Example 529. Forcing a scala-library dependency for the zinc configuration

build.gradle

```
configurations.zinc {
    resolutionStrategy.force "org.scala-lang:scala-library:2.10.5"
}
```

build.gradle.kts

```
configurations.zinc.apply {
    resolutionStrategy.force("org.scala-lang:scala-library:2.10.5")
}
```

You can diagnose problems with the version of the Zinc compiler selected by running dependencyInsight for the `zinc` configuration.

Convention properties

The Scala plugin does not add any convention properties to the project.

Source set properties

The Scala plugin adds the following convention properties 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.

`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 convention properties are provided by a convention object of type ScalaSourceSet.

The Scala plugin also modifies some source set properties:

Table 36. 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 530. Adjusting memory settings

build.gradle

```
tasks.withType(ScalaCompile) {
    scalaCompileOptions.forkOptions.with {
        memoryMaximumSize = '1g'
        jvmArgs = ['-XX:MaxPermSize=512m']
    }
}
```

build.gradle.kts

```
tasks.withType<ScalaCompile> {
    scalaCompileOptions.forkOptions.apply {
        memoryMaximumSize = "1g"
        jvmArgs = listOf("-XX:MaxPermSize=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 531. Forcing all code to be compiled

build.gradle

```
tasks.withType(ScalaCompile) {
    scalaCompileOptions.with {
        force = true
    }
}
```

build.gradle.kts

```
tasks.withType<ScalaCompile> {
    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.

Compiling and testing for Java 6 or Java 7

The Scala compiler ignores Gradle's `targetCompatibility` and `sourceCompatibility` settings. In Scala

2.11, the Scala compiler always compiles to Java 6 compatible bytecode. In Scala 2.12, the Scala compiler always compiles to Java 8 compatible bytecode. If you also have Java source, you can follow the same steps as for the Java plugin to ensure the correct Java compiler is used.

gradle.properties

```
# in $HOME/.gradle/gradle.properties
java6Home=/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

build.gradle

```
java {
    sourceCompatibility = JavaVersion.VERSION_1_6
}

assert hasProperty('java6Home') : "Set the property 'java6Home' in your your
gradle.properties pointing to a Java 6 installation"
def javaExecutablesPath = new File(java6Home, 'bin')
def javaExecutables = [:].withDefault { execName ->
    def executable = new File(javaExecutablesPath, execName)
    assert executable.exists() : "There is no ${execName} executable in
${javaExecutablesPath}"
    executable
}

tasks.withType(AbstractCompile) {
    options.with {
        fork = true
        forkOptions.javaHome = file(java6Home)
    }
}
tasks.withType(Test) {
    executable = javaExecutables.java
}
tasks.withType(JavaExec) {
    executable = javaExecutables.java
}
tasks.withType(Javadoc) {
    executable = javaExecutables.javadoc
}
```

build.gradle.kts

```
java {
    sourceCompatibility = JavaVersion.VERSION_1_6
}

require(hasProperty("java6Home")) { "Set the property 'java6Home' in your
your gradle.properties pointing to a Java 6 installation" }
val java6Home: String by project
val javaExecutablesPath = File(java6Home, "bin")
fun javaExecutable(execName: String): String {
    val executable = File(javaExecutablesPath, execName)
    require(executable.exists()) { "There is no ${execName} executable in
${javaExecutablesPath}" }
    return executable.toString()
}

tasks.withType<ScalaCompile> {
    options.apply {
        isFork = true
        forkOptions.javaHome = file(java6Home)
    }
}
tasks.withType<Test> {
    executable = javaExecutable("java")
}
tasks.withType<JavaExec> {
    executable = javaExecutable("java")
}
tasks.withType<Javadoc> {
    executable = javaExecutable("javadoc")
}
```

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 532. Explicitly specify a target IntelliJ IDEA version

build.gradle

```
idea {
    targetVersion = '13'
}
```

build.gradle.kts

```
idea {
    targetVersion = "13"
}
```

Integrating Gradle

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 37. 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 533. Using the Eclipse plugin

build.gradle

```
plugins {
    id 'eclipse'
}
```

build.gradle.kts

```
plugins {
    eclipse
}
```

Example 534. 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 38. 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 535. 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 536. 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 537. 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 538. 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
            .let { children -> (0..children.length).map { children.item(it) }
}
            .filterIsInstance<Element>()
            .first { it.predicate() }

    asElement()
        .firstElement { tagName === "fixed" && getAttribute("facet") ==
"jst.java" }
        .setAttribute("facet", "jst2.java")
})
```

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 539. 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 540. 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 541. 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 542. 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 543. 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
                .let { children -> (0..children.length).map {
children.item(it) } }
                .filterIsInstance<Element>()
                .first { it.predicate() }

        asElement()
            .firstElement { tagName == "component" && getAttribute("name") ==
"VcsDirectoryMappings" }
            .firstElement { tagName == "mapping" }
            .setAttribute("vcs", "Git")
    })
}
```

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.

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. We provide several *samples* that live in `samples/toolingApi` in your Gradle distribution. These samples specify all of the required dependencies for the Tooling API with examples for querying information from Gradle builds and executing tasks from the Tooling API.

To use the Tooling API, add the following repository and dependency declarations to your build script:

Example 544. 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.

Gradle version and Java version compatibility

Provider side

The current version of Tooling API supports running builds using Gradle versions 2.6 and later.

Consumer side

The current version of Gradle supports running builds via Tooling API versions 3.0 and 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.

Java version

The Tooling API requires Java 8 or later. The Gradle version used by builds may have additional Java version requirements.

Extending Gradle

Writing Custom 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 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 plugin, as the Gradle API has been designed to work well with these languages. 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/groovy` directory (or `rootProjectDir/buildSrc/src/main/kotlin` if you prefer using Kotlin) . 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 545. 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 need to obtain some configuration from the build script. One method for doing this is to use *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 a Java Bean compliant class. Groovy is a good language choice to implement an extension object because plain old Groovy objects contain all the getter and setter methods that a Java Bean requires. Java and Kotlin are other good choices.

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 546. A custom plugin extension

build.gradle

```
class GreetingPluginExtension {
    String message = 'Hello from GreetingPlugin'
}

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        // Add the 'greeting' extension object
        def extension = project.extensions.create('greeting',
GreetingPluginExtension)
        // Add a task that uses configuration from the extension object
        project.task('hello') {
            doLast {
                println extension.message
            }
        }
    }
}

apply plugin: GreetingPlugin

// Configure the extension
greeting.message = 'Hi from Gradle'
```

build.gradle.kts

```
open class GreetingPluginExtension {
    var message = "Hello from GreetingPlugin"
}

class GreetingPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        // Add the 'greeting' extension object
        val extension =
project.extensions.create<GreetingPluginExtension>("greeting")
        // Add a task that uses configuration from the extension object
        project.task("hello") {
            doLast {
                println(extension.message)
            }
        }
    }
}

apply<GreetingPlugin>()

// Configure the extension
the<GreetingPluginExtension>().message = "Hi from Gradle"
```

Output of `gradle -q hello`

```
> gradle -q hello
Hi from Gradle
```

In this example, `GreetingPluginExtension` is a plain old Groovy/Kotlin object with a property called `message`. The extension object is added to the plugin list 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 block for each extension object, so you can group settings together. The following example shows you how this works.

Example 547. A custom plugin with configuration block

build.gradle

```
class GreetingPluginExtension {
    String message
    String greeter
}

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        def extension = project.extensions.create('greeting',
GreetingPluginExtension)
        project.task('hello') {
            doLast {
                println "${extension.message} from ${extension.greeter}"
            }
        }
    }
}

apply plugin: GreetingPlugin

// Configure the extension using a DSL block
greeting {
    message = 'Hi'
    greeter = 'Gradle'
}
```

build.gradle.kts

```
open class GreetingPluginExtension {
    var message: String? = null
    var greeter: String? = null
}

class GreetingPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        val extension =
project.extensions.create<GreetingPluginExtension>("greeting")
        project.task("hello") {
            doLast {
                println("${extension.message} from ${extension.greeter}")
            }
        }
    }
}

apply<GreetingPlugin>()

// Configure the extension using a DSL block
configure<GreetingPluginExtension> {
    message = "Hi"
    greeter = "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.

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. To do this, you can leverage the Project.file(java.lang.Object) method to resolve values to files as late as possible.

Example 548. Evaluating file properties lazily

build.gradle

```
class GreetingToFileTask extends DefaultTask {

    def destination

    File getDestination() {
        project.file(destination)
    }

    @TaskAction
    def greet() {
        def file = getDestination()
        file.parentFile.mkdirs()
        file.write 'Hello!'
    }
}

task greet(type: GreetingToFileTask) {
    destination = { project.greetingFile }
}

task sayGreeting(dependsOn: greet) {
    doLast {
        println file(greetingFile).text
    }
}

ext.greetingFile = "$buildDir/hello.txt"
```

build.gradle.kts

```
open class GreetingToFileTask : DefaultTask() {

    var destination: Any? = null

    fun getDestination(): File {
        return project.file(destination!!)
    }

    @TaskAction
    fun greet() {
        val file = getDestination()
        file.parentFile.mkdirs()
        file.writeText("Hello!")
    }
}

task<GreetingToFileTask>("greet") {
    destination = provider { project.extra["greetingFile"]!! }
}

task("sayGreeting") {
    dependsOn("greet")
    doLast {
        println(file(project.extra["greetingFile"]!!).readText())
    }
}

extra["greetingFile"] = "$buildDir/hello.txt"
```

Output of `gradle -q sayGreeting`

```
> gradle -q sayGreeting
Hello!
```

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 considered a best practice. The end user only interacts with the

exposed DSL defined by the extension. The imperative logic is hidden in the plugin implementation.

The extension declaration in the build script as well as the mapping between extension properties and custom task properties occurs during Gradle's configuration phase of the build lifecycle. To avoid evaluation order issues, the actual value of a mapped property has to be resolved during the execution phase. For more information please see the section on build phases. Gradle's API offers types for representing a property that should be lazily evaluated e.g. during execution time. Refer to Lazy Configuration for more information.

The following demonstrates the usage of the type for mapping an extension property to a task property:

Example 549. Mapping extension properties to task properties

build.gradle

```
class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        def extension = project.extensions.create('greeting',
GreetingPluginExtension, project)
        project.tasks.create('hello', Greeting) {
            message = extension.message
            outputFiles = extension.outputFiles
        }
    }
}

class GreetingPluginExtension {
    final Property<String> message
    final ConfigurableFileCollection outputFiles

    GreetingPluginExtension(Project project) {
        message = project.objects.property(String)
        message.set('Hello from GreetingPlugin')
        outputFiles = project.layout.configurableFiles()
    }

    void setOutputFiles(FileCollection outputFiles) {
        this.outputFiles.setFrom(outputFiles)
    }
}

class Greeting extends DefaultTask {
    final Property<String> message = project.objects.property(String)
    final ConfigurableFileCollection outputFiles = project.layout
.configurableFiles()

    void setOutputFiles(FileCollection outputFiles) {
        this.outputFiles.setFrom(outputFiles)
    }
```

```
    @TaskAction
    void printMessage() {
        outputFiles.each {
            logger.quiet "Writing message 'Hi from Gradle' to file"
            it.text = message.get()
        }
    }
}

apply plugin: GreetingPlugin

greeting {
    message = 'Hi from Gradle'
    outputFiles = layout.files('a.txt', 'b.txt')
}
```

build.gradle.kts

```
class GreetingPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        val extension =
project.extensions.create<GreetingPluginExtension>("greeting", project)
        project.tasks.create<Greeting>("hello") {
            message = extension.message
            outputFiles = extension.outputFiles
        }
    }
}

open class GreetingPluginExtension(private val messageProp: Property<String>,
                                   private val configurableOutputFiles:
ConfigurableFileCollection) {

    var message by messageProp
    var outputFiles: FileCollection by configurableOutputFiles

    constructor(project: Project): this(project.objects.property<String>(),
project.layout.configurableFiles()) {
        message = "Hello from GreetingPlugin"
    }
}

open class Greeting : DefaultTask() {
    private val configurableOutputFiles: ConfigurableFileCollection =
project.layout.configurableFiles()

    var message by project.objects.property<String>()
    var outputFiles: FileCollection by configurableOutputFiles

    @TaskAction
    fun printMessage() {
        outputFiles.forEach {
            logger.quiet("Writing message 'Hi from Gradle' to file")
            it.writeText(message)
        }
    }
}

apply<GreetingPlugin>()

configure<GreetingPluginExtension> {
    message = "Hi from Gradle"
    outputFiles = layout.files("a.txt", "b.txt")
}
```

NOTE	The code for this example can be found at `samples/userguide/tasks/mapExtensionPropertiesToTaskProperties` in the '-all' distribution of Gradle.

Output of `gradle -q hello`

```
> gradle -q hello
Writing message 'Hi from Gradle' to file
Writing message 'Hi from Gradle' to file
```

A standalone project

Now we will move our plugin 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 plugin classes. 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 550. A build for a custom plugin

build.gradle

```
plugins {
    id 'groovy'
}

dependencies {
    implementation gradleApi()
    implementation localGroovy()
}
```

build.gradle.kts

```
plugins {
    groovy
}

dependencies {
    implementation(gradleApi())
    implementation(localGroovy())
}
```

NOTE	The code for this example can be found at `samples/customPlugin` in the '-all' distribution of Gradle.

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.

Example: Wiring for a custom plugin

src/main/resources/META-INF/gradle-plugins/org.samples.greeting.properties

```
implementation-class=org.gradle.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.

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

guide 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 add the plugin classes to the build script's classpath. To do this, you use a "buildscript { }" block, as described in see Applying plugins using the buildscript block. The following example shows how you might do this when the JAR containing the plugin has been published to a local repository:

Example 551. Using a custom plugin in another project

build.gradle

```
buildscript {
    repositories {
        maven {
            url = uri(repoLocation)
        }
    }
    dependencies {
        classpath 'org.gradle:customPlugin:1.0-SNAPSHOT'
    }
}
apply plugin: 'org.samples.greeting'
```

build.gradle.kts

```
buildscript {
    repositories {
        maven {
            url = uri(repoLocation)
        }
    }
    dependencies {
        classpath("org.gradle:customPlugin:1.0-SNAPSHOT")
    }
}
apply(plugin = "org.samples.greeting")
```

Alternatively, you can use the plugins DSL (see Applying plugins using the plugins DSL) to apply the plugin:

Example 552. Applying a community plugin with the plugins DSL

build.gradle

```
plugins {
    id 'com.jfrog.bintray' version '0.4.1'
}
```

build.gradle.kts

```
plugins {
    id("com.jfrog.bintray") version "0.4.1"
}
```

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/groovy/org/gradle/GreetingPluginTest.groovy

```
class GreetingPluginTest {
    @Test
    public void greeterPluginAddsGreetingTaskToProject() {
        Project project = ProjectBuilder.builder().build()
        project.pluginManager.apply 'org.samples.greeting'

        assertTrue(project.tasks.hello instanceof GreetingTask)
    }
}
```

Using the Java Gradle Plugin Development Plugin

You can use the Java Gradle Plugin Development Plugin to eliminate some of the boilerplate declarations in your build script and provide some basic validations of plugin metadata. This plugin will automatically apply the Java Plugin, add the `gradleApi()` dependency to the compile configuration, and perform plugin metadata validations as part of the `jar` task execution, and generate plugin descriptors in the resulting JAR's `META-INF` directory.

Example 553. Using the Java Gradle Plugin Development plugin

build.gradle

```
plugins {
    id 'java-gradle-plugin'
    id 'groovy'
}

gradlePlugin {
    plugins {
        simplePlugin {
            id = 'org.samples.greeting'
            implementationClass = 'org.gradle.GreetingPlugin'
        }
    }
}
```

build.gradle.kts

```
plugins {
    `java-gradle-plugin`
    groovy
}

gradlePlugin {
    plugins {
        create("simplePlugin") {
            id = "org.samples.greeting"
            implementationClass = "org.gradle.GreetingPlugin"
        }
    }
}
```

When publishing plugins to custom plugin repositories using the Ivy or Maven publish plugins, the Java Gradle Plugin Development Plugin will also generate plugin marker artifacts named based on the plugin id which depend on the plugin's implementation artifact.

Providing a configuration DSL for the plugin

As we saw above, you can use an extension object to provide configuration for your plugin. Using an extension object also extends the Gradle DSL to add a project property and DSL block for the plugin. An extension object is simply a regular object, and so you can provide DSL elements nested inside this block by adding properties and methods to the extension object.

Gradle provides several conveniences to help create a well-behaved DSL for your plugin.

Nested DSL elements

When Gradle creates a task or extension object, Gradle *decorates* the implementation class to mix in DSL support. To create a nested DSL element you can use the ObjectFactory type to create objects that are similarly decorated. These decorated objects can then be made visible to the DSL through properties and methods of the plugin's extension:

Example 554. Nested DSL elements

build.gradle

```
class Person {
    String name
}

class GreetingPluginExtension {
    String message
    final Person greeter

    @javax.inject.Inject
    GreetingPluginExtension(ObjectFactory objectFactory) {
        // Create a Person instance
        greeter = objectFactory.newInstance(Person)
    }

    void greeter(Action<? super Person> action) {
        action.execute(greeter)
    }
}

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        // Create the extension, passing in an ObjectFactory for it to use
        def extension = project.extensions.create('greeting',
GreetingPluginExtension, project.objects)
        project.task('hello') {
            doLast {
                println "${extension.message} from ${extension.greeter.name}"
            }
        }
    }
}

apply plugin: GreetingPlugin

greeting {
    message = 'Hi'
    greeter {
        name = 'Gradle'
    }
}
```

build.gradle.kts

```
open class Person {
    var name: String? = null
}

open class GreetingPluginExtension(val greeter: Person) {
    var message: String? = null

    // Create a Person instance
    @javax.inject.Inject
    constructor(objectFactory: ObjectFactory):
this(objectFactory.newInstance<Person>())

    fun greeter(action: Action<in Person>) {
        action.execute(greeter)
    }
}

class GreetingPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        // Create the extension, passing in an ObjectFactory for it to use
        val extension =
project.extensions.create<GreetingPluginExtension>("greeting",
project.objects)
        project.task("hello") {
            doLast {
                println("${extension.message} from
${extension.greeter.name}")
            }
        }
    }
}

apply<GreetingPlugin>()

configure<GreetingPluginExtension> {
    message = "Hi"
    greeter {
        name = "Gradle"
    }
}
```

Output of `gradle -q hello`

```
> gradle -q hello
Hi from Gradle
```

In this example, the plugin passes the project's `ObjectFactory` to the extension object through its constructor. The constructor uses this to create a nested object and makes this object available to the DSL through the `greeter` property.

Configuring a collection of objects

Gradle provides some utility classes for maintaining collections of objects, intended to work well with the Gradle DSL.

Example 555. Managing a collection of objects

build.gradle

```
class Book {
    final String name
    File sourceFile

    Book(String name) {
        this.name = name
    }
}

class DocumentationPlugin implements Plugin<Project> {
    void apply(Project project) {
        // Create a container of Book instances
        def books = project.container(Book)
        books.all {
            sourceFile = project.file("src/docs/$name")
        }
        // Add the container as an extension object
        project.extensions.books = books
    }
}

apply plugin: DocumentationPlugin

// Configure the container
books {
    quickStart {
        sourceFile = file('src/docs/quick-start')
    }
    userGuide {

    }
    developerGuide {

    }
}

task books {
    doLast {
        books.each { book ->
            println "$book.name -> $book.sourceFile"
        }
    }
}
```

build.gradle.kts

```
class Book(val name: String) {
    var sourceFile: File? = null
}

class DocumentationPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        // Create a container of Book instances
        val books = project.container<Book>()
        books.all {
            sourceFile = project.file("src/docs/$name")
        }
        // Add the container as an extension object
        project.extensions.add("books", books)
    }
}

apply<DocumentationPlugin>()

// Configure the container
val books: NamedDomainObjectContainer<Book> by extensions

books {
    create("quickStart") {
        sourceFile = file("src/docs/quick-start")
    }
    create("userGuide")
    create("developerGuide")
}

task("books") {
    doLast {
        books.forEach { book ->
            println("${book.name} -> ${book.sourceFile}")
        }
    }
}
```

Output of `gradle -q books`

```
> gradle -q books
developerGuide -> /home/user/gradle/samples/groovy/src/docs/developerGuide
quickStart -> /home/user/gradle/samples/groovy/src/docs/quick-start
userGuide -> /home/user/gradle/samples/groovy/src/docs/userGuide
```

```
> gradle -q books
developerGuide -> /home/user/gradle/samples/kotlin/src/docs/developerGuide
quickStart -> /home/user/gradle/samples/kotlin/src/docs/quick-start
userGuide -> /home/user/gradle/samples/kotlin/src/docs/userGuide
```

The Project.container(java.lang.Class) methods create instances of NamedDomainObjectContainer, that have many useful methods for managing and configuring the objects. In order to use a type with any of the `project.container` methods, it MUST expose a property named “name” as the unique, and constant, name for the object. The `project.container(Class)` variant of the container method creates new instances by attempting to invoke the constructor of the class that takes a single string argument, which is the desired name of the object. See the above link for `project.container` method variants that allow custom instantiation strategies.

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 plugin, adds the `gradleApi()` dependency to the compile 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 test compile 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 556. 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 plugin and adds the `gradleApi()` dependency to the compile 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 557. 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 Maven or Ivy Publish Plugins publishing plugins to publish a Plugin Marker Artifact for each plugin.
- Moreover, if the Plugin Publishing Plugin is applied, it will publish each plugin using the same name, plugin id, display name, and description to the Gradle Plugin Portal (see Publishing Plugins to Gradle Plugin Portal for details).

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

NOTE	The Provider API is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

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 558. Using a read-only and configurable property

build.gradle

```
// A task that displays a greeting
class Greeting extends DefaultTask {
    // A configurable greeting
    @Input
    final Property<String> greeting = project.objects.property(String)

    // Read-only property calculated from the greeting
    @Internal
    final Provider<String> message = greeting.map { it + ' from Gradle' }

    @TaskAction
    void printMessage() {
        logger.quiet(message.get())
    }
}

task greeting(type: Greeting) {
    // Configure the greeting
    greeting.set('Hi')

    // Note that an assignment statement can be used instead of calling
Property.set()
    greeting = 'Hi'
}
```

build.gradle.kts

```
// A task that displays a greeting
open class Greeting : DefaultTask() {
    // Configurable by the user
    @Input
    val greeting: Property<String> = project.objects.property()

    // Read-only property calculated from the greeting
    @Internal
    val message: Provider<String> = greeting.map { it + " from Gradle" }

    @TaskAction
    fun printMessage() {
        logger.quiet(message.get())
    }
}

task<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 will generate 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.

Kotlin DSL conveniences will be added in a future release.

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 559. Connecting properties together

build.gradle

```
// A project extension
class MessageExtension {
    // A configurable greeting
    final Property<String> greeting

    @javax.inject.Inject
    MessageExtension(ObjectFactory objects) {
        greeting = objects.property(String)
    }
}

// A task that displays a greeting
class Greeting extends DefaultTask {
    // A configurable greeting
    @Input
    final Property<String> greeting = project.objects.property(String)

    // 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, project.objects)

// Create the greeting task
task greeting(type: Greeting) {
    // Attach the greeting from the project extension
    // Note that the values of the project extension have not been configured
yet
    greeting.set(project.messages.greeting)

    // Note that an assignment statement can be used instead of calling
Property.set()
    greeting = project.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
open class MessageExtension(objects: ObjectFactory) {
    // A configurable greeting
    val greeting: Property<String> = objects.property()
}

// A task that displays a greeting
open class Greeting : DefaultTask() {
    // Configurable by the user
    @Input
    val greeting: Property<String> = project.objects.property()

    // 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("messages", MessageExtension::class,
project.objects)

// Create the greeting task
task<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)
}

configure<MessageExtension> {
    // 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 39. 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 560. Using file and directory property

build.gradle

```
// A task that generates a source file and writes the result to an output
directory
class GenerateSource extends DefaultTask {
    // The configuration file to use to generate the source file
    @InputFile
    final RegularFileProperty configFile = project.objects.fileProperty()

    // The directory to write source files to
    @OutputDirectory
    final DirectoryProperty outputDir = project.objects.directoryProperty()

    @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
task generate(type: GenerateSource) {
    // Configure the locations, relative to the project and build directories
    configFile = project.layout.projectDirectory.file('src/main/config.txt')
    outputDir = project.layout.buildDirectory.dir('generated-source')

    // Note that a `File` instance can be used as a convenience to set a
location
    configFile = file('src/config.txt')
}

// Change the build directory
// Don't need to reconfigure the task properties. These are automatically
updated as the build directory changes
buildDir = 'output'
```

build.gradle.kts

```
// A task that generates a source file and writes the result to an output
directory
open class GenerateSource @javax.inject.Inject constructor(objects:
ObjectFactory): DefaultTask() {
    @InputFile
    val configFile: RegularFileProperty = objects.fileProperty()

    @OutputDirectory
    val outputDir: DirectoryProperty = objects.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
task<GenerateSource>("generate") {
    // Configure the locations, relative to the project and build directories
    configFile.set(project.layout.projectDirectory.file("src/config.txt"))
    outputDir.set(project.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
buildDir = file("output")
```

Output of `gradle print`

```
> gradle print

> Task :generate
configuration file = /home/user/gradle/samples/groovy/src/config.txt
output dir = /home/user/gradle/samples/groovy/output/generated-source

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Output of `gradle print`

```
> gradle print

> 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().

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 561. Implicit task dependency

build.gradle

```
class Producer extends DefaultTask {
    @OutputFile
    final RegularFileProperty outputFile = project.objects.fileProperty()

    @TaskAction
    void produce() {
        String message = 'Hello, World!'
        def output = outputFile.get().asFile
        output.text = message
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

class Consumer extends DefaultTask {
    @InputFile
    final RegularFileProperty inputFile = project.objects.fileProperty()

    @TaskAction
    void consume() {
        def input = inputFile.get().asFile
        def message = input.text
        logger.quiet("Read '${message}' from ${input}")
    }
}

task producer(type: Producer)
task consumer(type: 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
consumer.inputFile = producer.outputFile

// Set values for the producer lazily
// Don't need to update the consumer.inputFile property. This is
automatically updated as producer.outputFile changes
producer.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
buildDir = 'output'
```

build.gradle.kts

```
open class Producer : DefaultTask() {
    @OutputFile
    val outputFile: RegularFileProperty = project.objects.fileProperty()

    @TaskAction
    fun produce() {
        val message = "Hello, World!"
        val output = outputFile.get().asFile
        output.writeText( message)
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

open class Consumer : DefaultTask() {
    @InputFile
    val inputFile: RegularFileProperty = project.objects.fileProperty()

    @TaskAction
    fun consume() {
        val input = inputFile.get().asFile
        val message = input.readText()
        logger.quiet("Read '${message}' from ${input}")
    }
}

val producer by tasks.creating(Producer::class)
val consumer by tasks.creating(Consumer::class)

// 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
consumer.inputFile.set(producer.outputFile)

// Set values for the producer lazily
// Don't need to update the consumer.inputFile property. This is
automatically updated as producer.outputFile changes
producer.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
buildDir = file("output")
```

Output of `gradle consumer`

```
> gradle consumer

> Task :producer
Wrote 'Hello, World!' to /home/user/gradle/samples/groovy/output/file.txt

> Task :consumer
Read 'Hello, World!' from /home/user/gradle/samples/groovy/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.

Working with collections

In this section, we are going to explore lazy collections. They work exactly like any other `Provider` and, just like file providers, they have additional modeling around them. There are two provider interfaces available, one for `List` values and another for `Set` values:

- 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 562. List property

build.gradle

```
class Producer extends DefaultTask {
    @OutputFile
    final RegularFileProperty outputFile = project.objects.fileProperty()

    @TaskAction
    void produce() {
        String message = 'Hello, World!'
        def output = outputFile.get().asFile
        output.text = message
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

class Consumer extends DefaultTask {
    @InputFiles
    final ListProperty<RegularFile> inputFiles = project.objects.
listProperty(RegularFile).empty()

    @TaskAction
    void consume() {
        inputFiles.get().each { inputFile ->
            def input = inputFile.asFile
            def message = input.text
            logger.quiet("Read '${message}' from ${input}")
        }
    }
}

task producerOne(type: Producer)
task producerTwo(type: Producer)
task consumer(type: 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
consumer.inputFiles.add(producerOne.outputFile)
consumer.inputFiles.add(producerTwo.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 = layout.buildDirectory.file('one.txt')
producerTwo.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
buildDir = 'output'
```

build.gradle.kts

```
open class Producer : DefaultTask() {
    @OutputFile
    val outputFile: RegularFileProperty = project.objects.fileProperty()

    @TaskAction
    fun produce() {
        val message = "Hello, World!"
        val output = outputFile.get().asFile
        output.writeText( message)
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

open class Consumer : DefaultTask() {
    @InputFiles
    val inputFiles: ListProperty<RegularFile> =
project.objects.listProperty(RegularFile::class).empty()

    @TaskAction
    fun consume() {
        inputFiles.get().forEach { inputFile ->
            val input = inputFile.asFile
            val message = input.readText()
            logger.quiet("Read '${message}' from ${input}")
        }
    }
}

val producerOne by tasks.creating(Producer::class)
val producerTwo by tasks.creating(Producer::class)
val consumer by tasks.creating(Consumer::class)

// 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
consumer.inputFiles.add(producerOne.outputFile)
consumer.inputFiles.add(producerTwo.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
buildDir = file("output")
```

Output of `gradle consumer`

```
> gradle consumer

> Task :producerOne
Wrote 'Hello, World!' to /home/user/gradle/samples/groovy/output/one.txt

> Task :producerTwo
Wrote 'Hello, World!' to /home/user/gradle/samples/groovy/output/two.txt

> Task :consumer
Read 'Hello, World!' from /home/user/gradle/samples/groovy/output/one.txt
Read 'Hello, World!' from /home/user/gradle/samples/groovy/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
```

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 configure 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 a finalizeValue() method to make this explicit. Calling this method makes a property instance unmodifiable from that point on 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.

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.

The Provider API is incubating. Please create new issues at gradle/gradle to report bugs or to submit use cases for new features.

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...)

FileTree

Hierarchy of files

Factories

- Project.fileTree(Object) will produce a ConfigurableFileTree, or you can use Project.zipTree(Object) and Project.tarTree(Object)

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

- ProjectLayout.configurableFiles(Object...)

ConfigurableFileTree

Hierarchy of files

Factories

- Project.fileTree(Object)

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)

Reference

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.

Groovy

Provides support for building any type of Groovy project.

Scala

Provides support for building any type of Scala project.

Play

Proivdes support for building, testing and running Play applications.

ANTLR

Provides support for generating parsers using ANTLR.

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.

OSGi

Provides support for creating OSGi packages.

Maven Publish

Provides support for publishing artifacts to Maven-compatible repositories.

Ivy Publish

Provides support for publishing artifacts to Ivy-compatible repositories.

Legacy Maven Plugin

Provides support for publishing artifacts using the legacy mechanism to Maven-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.

FindBugs

Performs quality checks on your project's Java source files using FindBugs and generates associated reports.

PMD

Performs quality checks on your project's Java source files using PMD and generates associated reports.

JDepend

Performs quality checks on your project's Java source files using JDepend and generates associated reports.

JaCoCo

Provides code coverage metrics for your Java project using JaCoCo.

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.

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.

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

Gradle comes with a Visual Studio plugin that will generate VisualStudioProject and VisualStudioSolution files for your native projects.

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.

Licenses

License Information

Gradle Documentation

Copyright © 2007-2018 Gradle, Inc.

Gradle build tool source code is open-source and licensed under the Apache License 2.0.

Gradle user manual and DSL references 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.