Gradle User Manual

Version 7.6

Version 7.6

Table of Contents

About Gradle

What is Gradle?

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

Design

Gradle bases its design on the following fundamentals:

High performance

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

JVM foundation

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

Conventions

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

Extensibility

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

IDE support

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

Insight

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

Terminology

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

Projects

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

Tasks

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

Tasks themselves consist of:

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

Plugins

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

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

Build Phases

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

Initialization

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

Configuration

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

Execution

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

Builds

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

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

Getting Started

Getting Started

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

Before you start

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

Installation

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

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

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

Try Gradle

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

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

- Building Swift libraries
- Creating build scans

There are more samples available on the samples pages.

Command line vs IDEs

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

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

Executing Gradle builds

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

Here are some useful steps to follow:

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

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

3. Find out what tasks you can run.

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

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

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

5. Run the task that you are interested in.

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

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

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

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

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

Authoring Gradle builds

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

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

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

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

Integrating 3rd-party tools with Gradle

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

There are two main modes of integration:

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

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

Installing Gradle

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

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

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

Prerequisites

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

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

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

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

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

Installing with a package manager

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

```
 sdk install gradle
```

Homebrew is "the missing package manager for macOS".

```
 brew install gradle
```

MacPorts is another package manager for macOS.

```
 sudo port install gradle
```

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

 Proceed to next steps

Installing manually

Step 1. Download the latest Gradle distribution

The distribution ZIP file comes in two flavors:

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

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

Step 2. Unpack the distribution

Linux & MacOS users

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

```
 mkdir /opt/gradle
 unzip -d /opt/gradle gradle-7.6-bin.zip
 ls /opt/gradle/gradle-7.6
LICENSE  NOTICE  bin  README  init.d  lib  media
```

Microsoft Windows users

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

Open a second **File Explorer** window and go to the directory where the Gradle distribution was downloaded. Double-click the ZIP archive to expose the content. Drag the content folder `gradle-7.6` to your newly created `C:\Gradle` folder.

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

Step 3. Configure your system environment

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

Linux & MacOS users

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

```
 export PATH=$PATH:/opt/gradle/gradle-7.6/bin
```

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

Microsoft Windows users

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

Under `System Variables` select `Path`, then click `Edit`. Add an entry for `C:\Gradle\gradle-7.6\bin`. Click OK to save.

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

 Proceed to next steps

Verifying installation

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

```
 gradle -v

------------------------------------------------------------
Gradle 7.6
------------------------------------------------------------

(environment specific information)
```

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

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

Next steps

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

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

Troubleshooting builds

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

documentation from help.gradle.org.

Troubleshooting Gradle installation

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

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

You should see something like this:

```
 gradle --version

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

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

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

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

Command not found: gradle

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

JAVA_HOME is set to an invalid directory

If you get something like:

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

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

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

Permission denied

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

Other installation failures

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

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

Debugging dependency resolution

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

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

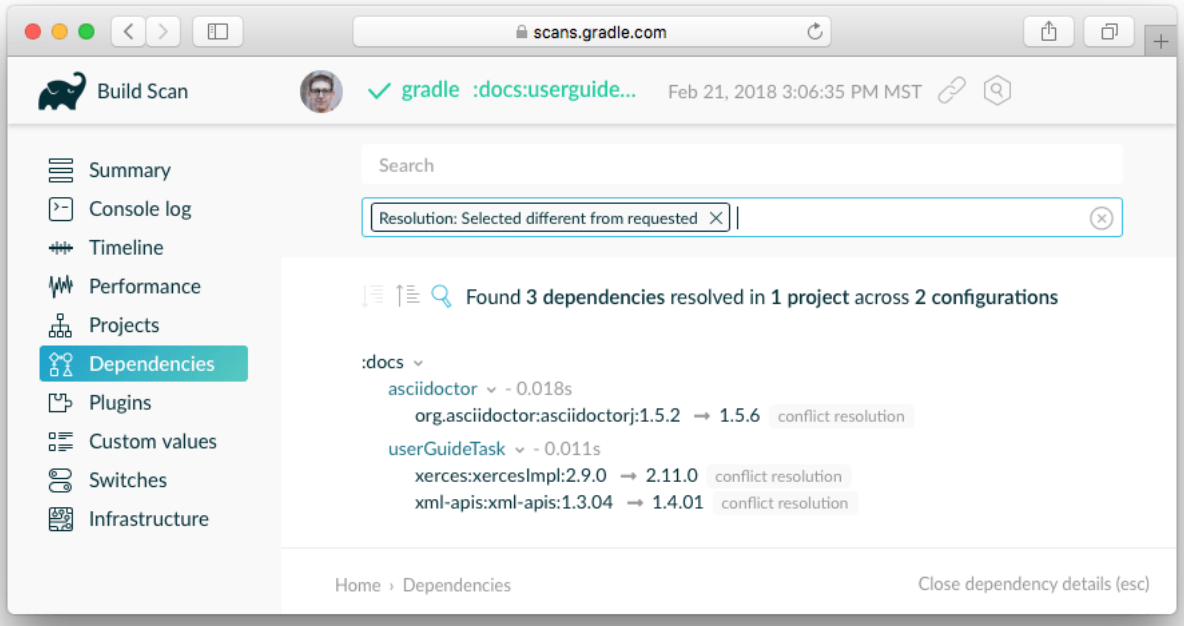

Figure 1. Debugging dependency conflicts with build scans

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

Troubleshooting slow Gradle builds

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

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

Debugging build logic

Attaching a debugger to your build

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

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

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

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

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

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

Figure 2. Interactive debugging of a build script

Adding and changing logging

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

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

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

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

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

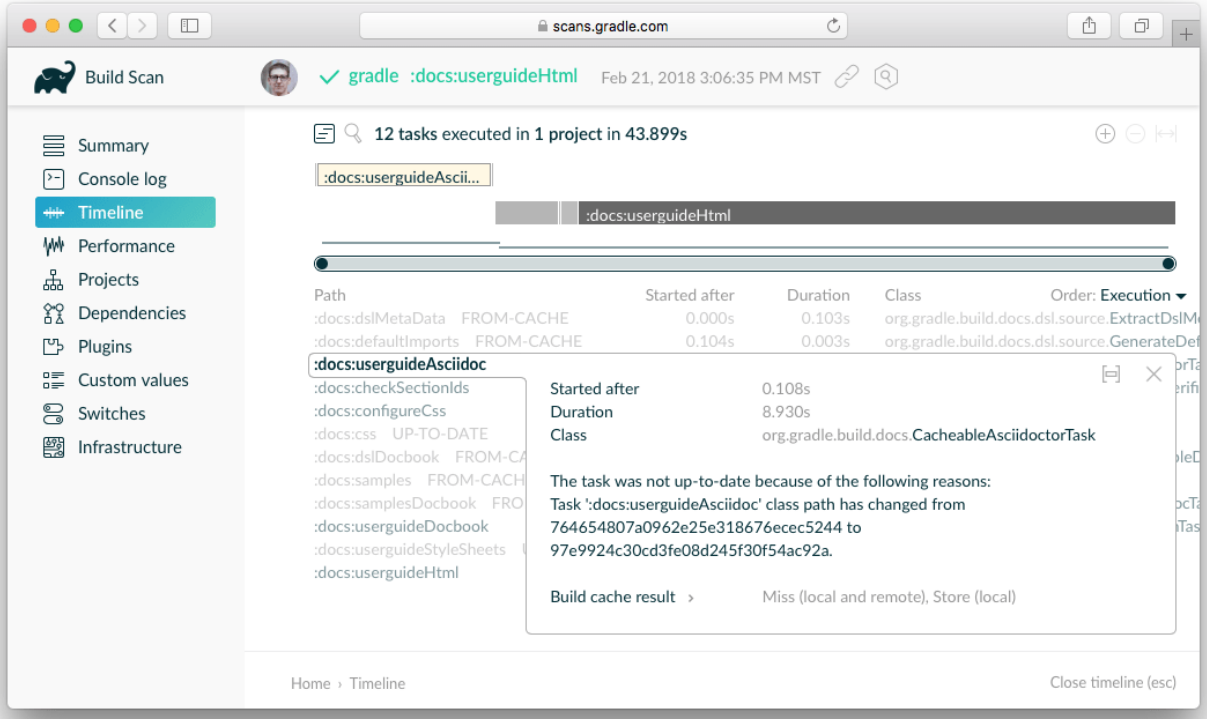

Figure 3. Debugging incremental build with a build scan

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

Debugging IDE integration

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

Refreshing IntelliJ IDEA

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

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

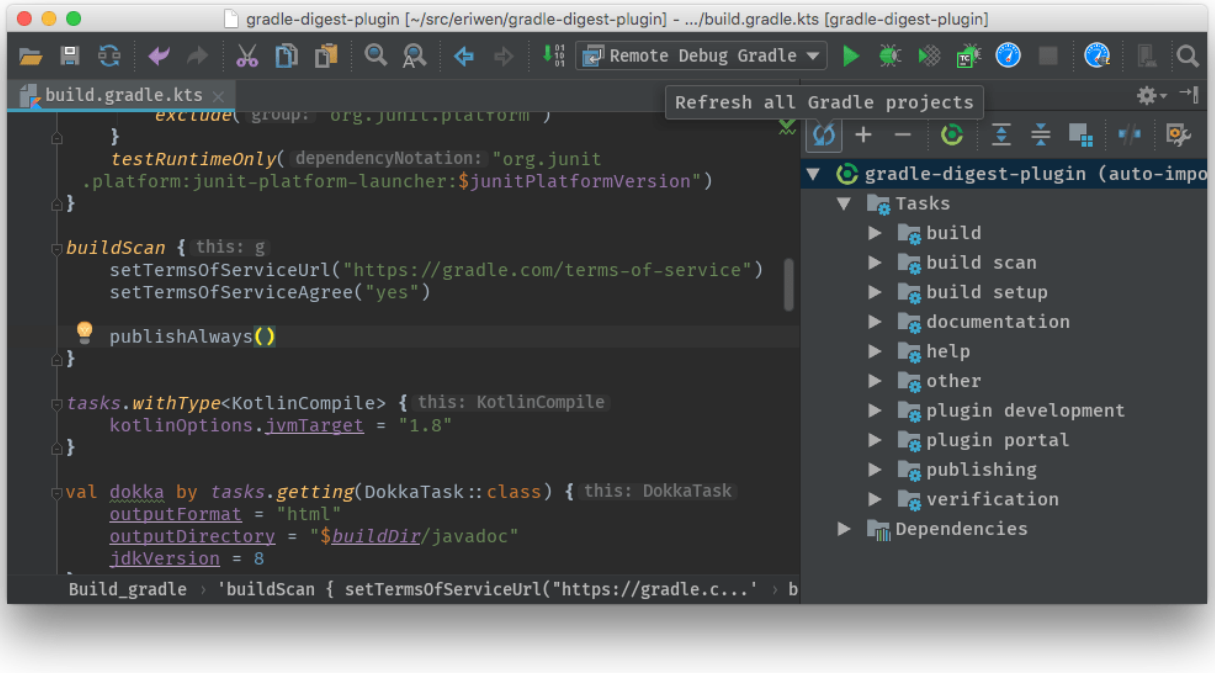

Figure 4. Refreshing a Gradle project in IntelliJ IDEA

Refreshing Eclipse (using Buildship)

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

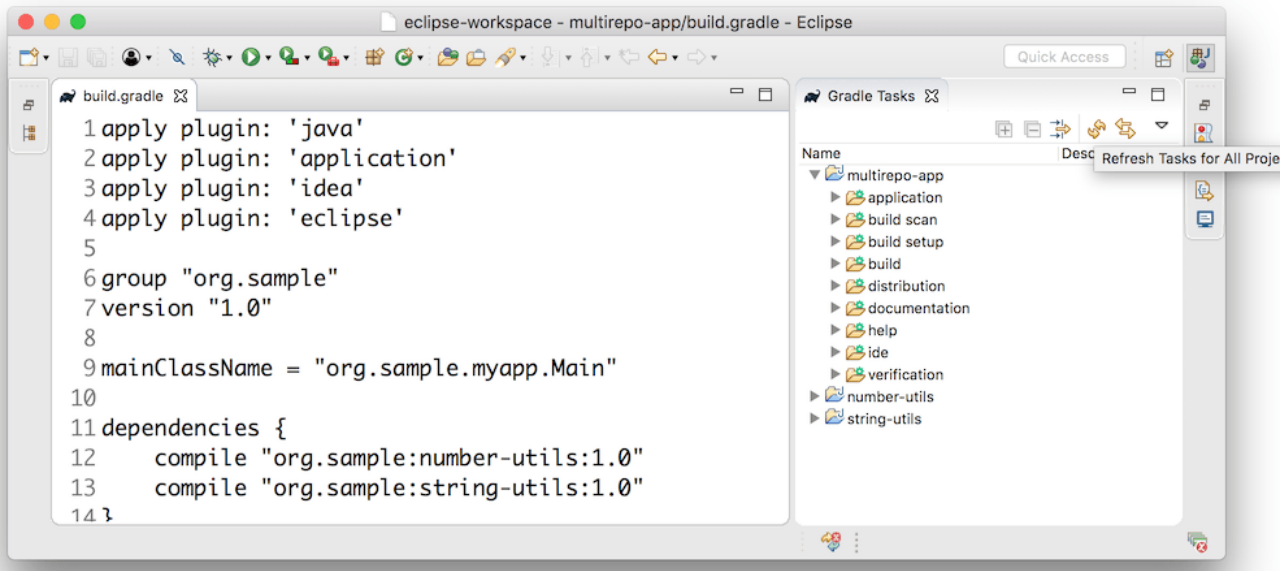

Figure 5. Refreshing a Gradle project in Eclipse Buildship

Troubleshooting daemon connection issues

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

```
$ gradle help

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

FAILURE: Build failed with an exception.

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

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

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

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

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

Getting additional help

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

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

Compatibility Matrix

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

Java

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

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

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

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

Table 1. Java Compatibility

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

Kotlin

Gradle is tested with Kotlin 1.3.72 through 1.7.10.

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

Table 2. Embedded Kotlin version

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

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

Groovy

Gradle is tested with Groovy 1.5.8 through 4.0.0.

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

Android

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