


Computer science → Fundamentals → Dev tools → Build Tools → Gradle

Dependency management

Theory

Practice

 100% completed, 0 problems solved ▾

Theory

🕒 13 minutes reading

Unskip this topic

Start practicing

Small programs that you write when learning a language may not use any external libraries at all. When you need some functionality, you will find it in the standard library or create it yourself. However, it is quite difficult to develop a real application that doesn't use any libraries because they save tons of your time and provide solutions tested by millions of people around the world.

In this topic, we will learn how to add external libraries to our applications using Gradle.

§1. How to add dependencies?

In Gradle terminology, all external libraries are called **dependencies**. As a rule, they are packaged in JAR files. Gradle can automatically download them and add them to the project. It saves a lot of time and solves possible conflicts between versions of libraries.

Where do we get these dependencies and how do we add them to a project? To use a class, you need to have it locally, and your JVM must know that you have it. If you want to do it manually, you need to find and download such jars on your own and add them to the classpath of your project. Sounds quite tedious, right?

Fortunately, dependency management is one of the key features of Gradle. You don't even need a plugin for it. To add an external library to a project, you need to do exactly two steps:

1. **Define a repository** where to search for dependencies.
2. **Define a dependency** that you want to include in your project.

Now, let's consider these steps in more detail.

§2. Repository definition

Repositories are just places where libraries are stored. Any project can use zero or more repositories.

There are different possible formats of repositories:


- a Maven compatible repository (e.g.: [Maven Central](#), [JCenter](#), [Google](#))
- an Ivy compatible repository;
- local (flat) directories.

It's possible to host repositories like **Maven** or **JCenter** locally in your organization, but that is out of the scope of this tutorial. We will only consider public online versions of them.

Gradle has four aliases that you can use when adding Maven compatible repositories to the project.


- `mavenCentral()` : fetches dependencies from the [Maven Central Repository](#).
- `jcenter()` : fetches dependencies from the [Bintray's JCenter Maven repository](#).
- `mavenLocal()` : fetches dependencies from the local Maven repository.

1 required topic




✓ [Building apps using Gradle](#) ▾

10 dependent topics



✓ [Basic project structure](#) ▾

[Modules](#) ▾



[Connecting to a database with JDBC](#) ▾

[Unit testing with JUnit](#) ▾

[Mockito](#) ▾

[JUnit](#) ▾

[First steps to coroutines](#) ▾

[Kotlinx serialization](#) ▾

[Reducing boilerplate code with Lombok](#) ▾

[Kotlinx datetime library](#) ▾

- `google()` : fetches dependencies from [Google Maven repository](#).

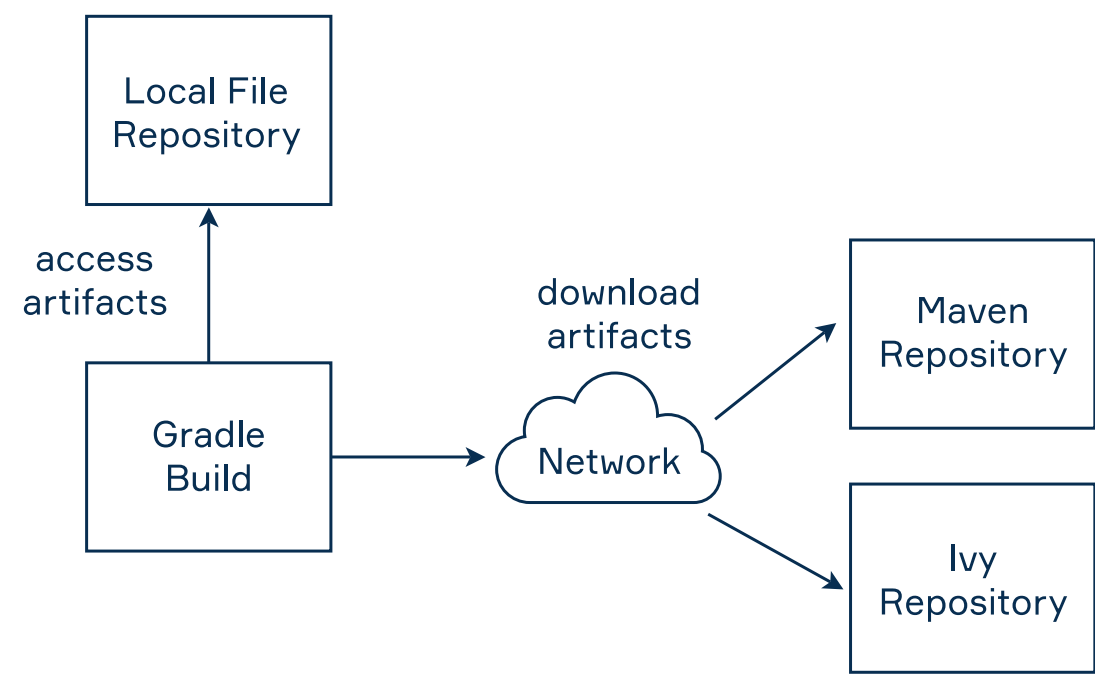
Defining a repository in Gradle is a piece of cake. Just add this to your `build.gradle` file:

```
1 repositories {
2     mavenCentral()
3     jcenter()
4 }
```

Also, you can just download the jars you need and place them into some directory on your computer, commonly in the `libs` folder of your project. This comes in handy when the jars you need are not available in public repositories.

```
repositories {
    flatDir {
        dirs 'lib'
    }
}
```

The following picture demonstrates how to add dependencies from different repositories using Gradle.

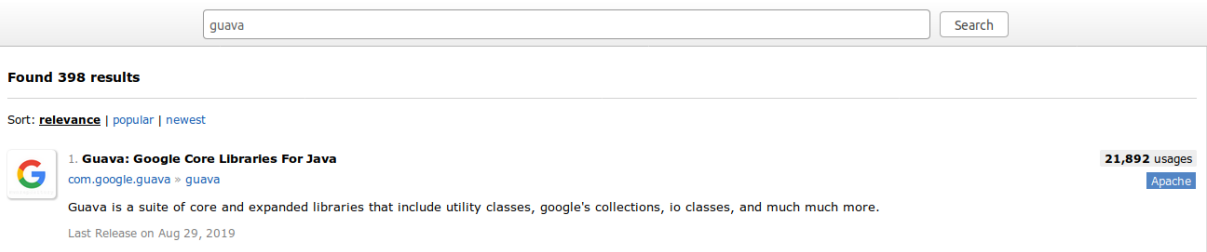


Now let's find out how to download dependencies from these repositories.

§3. Dependencies definition

To add a new dependency to your project, first, you need to identify it by the following attributes: `group`, `name` and `version`. All these attributes are required when you use Maven compatible repositories. If you use other repositories, some attributes might be optional. For example, a flat repository only needs a name and a version.

There are several ways to find these attributes. Some library maintainers are nice enough to list them on their website or git repository. Also, you can just search for them in your preferred repository.



Maven Central search example

All the dependencies are grouped into a named set of dependencies called **configurations**. Each of them has different characteristics and determines the following points:

- the availability of dependencies on building steps;
- the need to include dependencies in the final build artifact;
- the visibility of dependencies for programmers who use your project as a library.

The `'java'` and `'kotlin'` plugins add a number of these configurations to your project. There are four of them:

- `implementation` configuration means that the dependency is available at compile-time and it can't be exposed to people who use your compiled code as an external library in their own projects. This configuration is considered the default one.
- `compileOnly` configuration is used to define dependencies that should only be available at compile-time, but you do not need them at runtime.
- `runtimeOnly` is used to define dependencies that you need only during runtime, and not at compile time.
- `api` configuration is similar to `implementation`, but will be exposed to the programmers who use your compiled code as a library in their projects.

In an existing project, you may see `compile` and `runtime` configurations as well, but they are deprecated now. Consider using `implementation` and `runtimeOnly` instead.

There also exist the same configurations only with the `test` prefix (e.g. `testImplementation`). Since tests are compiled and run separately and are not included in the final JAR (as well as their dependencies), they have their own set of dependencies. It helps decrease the size of a JAR, which is especially useful in Android development.

Please, note, that at this moment, you do not need to understand everything about configurations. Usually, the type of dependency configuration is already specified when you copy dependency info from an online repository or a website. If you would like to read more about configurations, [see the official Gradle docs](#).

When we decided on what dependencies we want and settled on their configurations, we are ready to add them to our `build.gradle`, which is as simple as adding repositories.

```
1 dependencies {
2     // This dependency is used by the application.
3     implementation group: 'com.google.guava', name: 'guava',
version: '28.0-jre'
4
5     // Use JUnit test framework only for testing
6     testImplementation 'junit:junit:4.12'
7
8     // It is only needed to compile the application
9     compileOnly 'org.projectlombok:lombok:1.18.4'
10 }
```

Here, we add these three dependencies as an example:

- **Guava** library, provides useful collections and utils for a project;
- **JUnit**, used for testing purposes;
- **Lombok**, modifies bytecode at compile time and isn't necessary anymore after compilation.

As you may have noticed, there are two ways of declaring dependencies: the one where we explicitly declare group, name, and version, and the one where we just list them separated by the colon. Both are perfectly fine and are up to your preferences. Note that Groovy syntax is flexible and you can use either single or

double quotes for the dependency string and optionally enclose it in parentheses.
All the following declarations are equally valid:

```

1 // 1
2 implementation("com.google.guava:guava:28.0-jre")
3
4 // 2
5 implementation "com.google.guava:guava:28.0-jre"
6
7 // 3
8 implementation 'com.google.guava:guava:28.0-jre'
9
1
0 // 4
1
1 def guava_version = "28.0-jre"
1
2 implementation "com.google.guava:guava:$guava_version"

```

After adding dependencies in the `build.gradle` file, you can use the libraries in your source code, but do not forget to import them. Gradle will automatically download the libraries from repositories when building the project.

§4. Colorful world

As an example of using external libraries, we take a look at a program that prints colored text messages.

1. In the **dependencies** section of the `build.gradle` file we need to include JCDP library:

```

1 implementation group: 'com.diagonunes', name: 'JCDP', version:
'2.0.3.1'

```

2. And then import and use it inside the source code. Here are Java and Kotlin examples.

Java:

```

1 package org.hyperskill.gradleapp;
2
3 import com.diagonunes.jcdp.color.ColoredPrinter;
4 import com.diagonunes.jcdp.color.api.Ansi;
5
6 public class App {
7     public static void main(String[] args) {
8         ColoredPrinter printer = new ColoredPrinter
9             .Builder(1, false).build();
10
11
12         printer.print("Hello, colorful world!",
13
14             Ansi.Attribute.BOLD, Ansi.FColor.BLUE,
15             Ansi.BColor.WHITE);
16
17     }
18 }

```

Kotlin:

```
1 package com.hyperskill.gradleapp
2
3 import com.diogonunes.jcdp.color.ColoredPrinter
4 import com.diogonunes.jcdp.color.api.Ansi
5
6 fun main(args: Array<String>) {
7     val printer = ColoredPrinter.Builder(1, false).build()
8
9     printer.print("Hello, colorful world!",
10
11                 Ansi.Attribute.BOLD, Ansi.FColor.BLUE,
12                 Ansi.BColor.WHITE)
13 }
14 }
```

Both programs print the same colored message: **Hello, colorful world!**

§5. Conclusion

You've learned only the basics of dependency management, but this is enough to write programs with external libraries. As you may have noticed, Gradle is a very flexible tool for managing dependencies. It allows you to choose repositories where to download them and also to configure when to use dependencies: during compile-time, during runtime, or when testing, and so on. As an example, we examined a program that prints colored messages using an external library.

Report a typo

441 users liked this piece of theory. 43 didn't like it. What about you?



Start practicing

Unskip this topic

Table of contents:

- 1 Dependency management
- §1. How to add dependencies?
- §2. Repository definition
- §3. Dependencies definition
- §4. Colorful world
- §5. Conclusion
- Discussion

Comments (59)

Useful links (4)

Show discussion



Tracks

Pricing

For organizations

About

Contribute

Careers

Become beta tester

Be the first to see what's new

Terms Support

Made with by Hyperskill and JetBrains