3.1 Gradle

Table of Contents

7.1. What is dependency management?7.2. Declaring your dependencies7.3. Dependency configurations7.4. External dependencies7.5. Repositories7.6. Publishing artifacts7.7. Where to next?

This chapter introduces some of the basics of dependency management in Gradle.

7.1. What is dependency management?

Very roughly, dependency management is made up of two pieces. Firstly, Gradle needs to know about the things that your project needs to build or run, in order to find them. We call these incoming files the *dependencies* of the project. Secondly, Gradle needs to build and upload the things that your project produces. We call these outgoing files the *publications* of the project. Let's look at these two pieces in more detail:

Most projects are not completely self-contained. They need files built by other projects in order to be compiled or tested and so on. For example, in order to use Hibernate in my project, I need to include some Hibernate jars in the classpath when I compile my source. To run my tests, I might also need to include some additional jars in the test classpath, such as a particular JDBC driver or the Ehcache jars.

These incoming files form the dependencies of the project. Gradle allows you to tell it what the dependencies of your project are, so that it can take care of finding these dependencies, and making them available in your build. The dependencies might need to be downloaded from a remote Maven or Ivy repository, or located in a local directory, or may need to be built by another project in the same multi-project build. We call this process *dependency resolution*.

Note that this feature provides a major advantage over Ant. With Ant, you only have the ability to specify absolute or relative paths to specific jars to load. With Gradle, you simply declare the “names” of your dependencies, and other layers determine where to get those dependencies from. You can get similar behavior from Ant by adding Apache Ivy, but Gradle does it better.

Often, the dependencies of a project will themselves have dependencies. For example, Hibernate core requires several other libraries to be present on the classpath with it runs. So, when Gradle runs the tests for your project, it also needs to find these dependencies and make them available. We call these *transitive dependencies*.

The main purpose of most projects is to build some files that are to be used outside the project. For example, if your project produces a Java library, you need to build a jar, and maybe a source jar and some documentation, and publish them somewhere.

These outgoing files form the publications of the project. Gradle also takes care of this important work for you. You declare the publications of your project, and Gradle take care of building them and publishing them somewhere. Exactly what “publishing” means depends on what you want to do. You might want to copy the files to a local directory, or upload them to a remote Maven or Ivy repository. Or you might use the files in another project in the same multi-project build. We call this process *publication*.

7.2. Declaring your dependencies

Let's look at some dependency declarations. Here's a basic build script:

Example 7.1. Declaring dependencies

build.gradle

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

What's going on here? This build script says a few things about the project. Firstly, it states that Hibernate core 3.6.7.Final is required to compile the project's production source. By implication, Hibernate core and its dependencies are also required at runtime. The build script also states that any junit >= 4.0 is required to compile the project's tests. It also tells Gradle to look in the Maven central repository for any dependencies that are required. The following sections go into the details.

7.3. Dependency configurations

In Gradle dependencies are grouped into *configurations*. A configuration is simply a named set of dependencies. We will refer to them as *dependency configurations*. You can use them to declare the external dependencies of your project. As we will see later, they are also used to declare the publications of your project.

The Java plugin defines a number of standard configurations. These configurations represent the classpaths that the Java plugin uses. Some are listed below, and you can find more details in Table 47.5, “Java plugin - dependency configurations”.
compile: The dependencies required to compile the production source of the project.

runtime: The dependencies required by the production classes at runtime. By default, also includes the compile time dependencies.

testCompile: The dependencies required to compile the test source of the project. By default, also includes the compiled production classes and the compile time dependencies.

testRuntime: The dependencies required to run the tests. By default, also includes the compile, runtime and test compile dependencies.

Various plugins add further standard configurations. You can also define your own custom configurations to use in your build. Please see Section 25.3, “Dependency configurations” for the details of defining and customizing dependency configurations.

7.4. External dependencies

There are various types of dependencies that you can declare. One such type is an *external dependency*. This is a dependency on some files built outside the current build, and stored in a repository of some kind, such as Maven central, or a corporate Maven or Ivy repository, or a directory in the local file system.

To define an external dependency, you add it to a dependency configuration:

Example 7.2. Definition of an external dependency

`build.gradle`

```
dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
}
```

An external dependency is identified using `group`, `name` and `version` attributes. Depending on which kind of repository you are using, `group` and `version` may be optional.

The shortcut form for declaring external dependencies looks like "*`group`*`:`*`name`*`:`*`version`*".

Example 7.3. Shortcut definition of an external dependency

`build.gradle`

```
dependencies {
    compile 'org.hibernate:hibernate-core:3.6.7.Final'
}
```

To find out more about defining and working with dependencies, have a look at Section 25.4, "How to declare your dependencies".

7.5. Repositories

How does Gradle find the files for external dependencies? Gradle looks for them in a *repository*. A repository is really just a collection of files, organized by `group`, `name` and `version`. Gradle understands several different repository formats, such as Maven and Ivy, and several different ways of accessing the repository, such as using the local file system or HTTP.

By default, Gradle does not define any repositories. You need to define at least one before you can use external dependencies. One option is use the Maven central repository:

Example 7.4. Usage of Maven central repository

`build.gradle`

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

Or Bintray's JCenter:

Example 7.5. Usage of JCenter repository

`build.gradle`

```
repositories {
    jcenter()
}
```

Or a any other remote Maven repository:

Example 7.6. Usage of a remote Maven repository

`build.gradle`

```
repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
    }
}
```

Or a remote Ivy repository:

Example 7.7. Usage of a remote Ivy directory

`build.gradle`

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
    }
}
```

You can also have repositories on the local file system. This works for both Maven and Ivy repositories.

Example 7.8. Usage of a local Ivy directory

`build.gradle`

```
repositories {
    ivy {
        // URL can refer to a local directory
        url "../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.