< ALL GUIDES

Building Java Projects with Gradle

This guide walks you through using Gradle to build a simple Java project.

What you'll build

You'll create a simple app and then build it using Gradle.

What you'll need

- About 15 minutes
- A favorite text editor or IDE
- JDK 6 or later

How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and complete each step or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Set up the project.

To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git: `git clone https://github.com/spring-guides/gs-gradle.git`
- cd into `gs-gradle/initial`
- Jump ahead to Install Gradle.

When you finish, you can check your results against the code in `gs-gradle/complete`.

Set up the project

First you set up a Java project for Gradle to build. To keep the focus on Gradle, make the project as simple as possible for now.

Create the directory structure

In a project directory of your choosing, create the following subdirectory structure; for example, with `mkdir -p src/main/java/hello` on *nix systems:

```
└── src
    └── main
        └── java
            └── hello
```

Within the `src/main/java/hello` directory, you can create any Java classes you want. For simplicity's sake and for consistency with the rest of this guide, Spring recommends that you create two classes: `HelloWorld.java` and `Greeter.java`.

`src/main/java/hello/HelloWorld.java`

```
package hello;

public class HelloWorld {
  public static void main(String[] args) {
  Greeter greeter = new Greeter();
  System.out.println(greeter.sayHello());
  }
}
```

COPY

`src/main/java/hello/Greeter.java`

```
package hello;

public class Greeter {
  public String sayHello() {
  return "Hello world!";
```

COPY

```
    }
}
```

Install Gradle

Now that you have a project that you can build with Gradle, you can install Gradle.

It's highly recommended to use an installer:

- SDKMAN
- Homebrew (brew install gradle)

As a last resort, if neither of these tools suit your needs, you can download the binaries from https://www.gradle.org/downloads. Only the binaries are required, so look for the link to gradle-*version*-bin.zip. (You can also choose gradle-*version*-all.zip to get the sources and documentation as well as the binaries.)

Unzip the file to your computer, and add the bin folder to your path.

To test the Gradle installation, run Gradle from the command-line:

```
gradle
```

If all goes well, you see a welcome message:

```
:help

Welcome to Gradle 6.0.1.

To run a build, run gradle <task> ...

To see a list of available tasks, run gradle tasks

To see a list of command-line options, run gradle --help

To see more detail about a task, run gradle help --task <task>

For troubleshooting, visit https://help.gradle.org

Deprecated Gradle features were used in this build, making it incom
Use '--warning-mode all' to show the individual deprecation warning
See https://docs.gradle.org/6.0.1/userguide/command_line_interface.

BUILD SUCCESSFUL in 455ms
1 actionable task: 1 executed
```

You now have Gradle installed.

Find out what Gradle can do

Now that Gradle is installed, see what it can do. Before you even create a build.gradle file for the project, you can ask it what tasks are available:

```
gradle tasks
```

You should see a list of available tasks. Assuming you run Gradle in a folder that doesn't already have a *build.gradle* file, you'll see some very elementary tasks such as this:

```
:tasks

------------------------------------------------------------
Tasks runnable from root project
------------------------------------------------------------

Build Setup tasks
-----------------
init - Initializes a new Gradle build.
wrapper - Generates Gradle wrapper files.

Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared i
components - Displays the components produced by root project 'gs-g
dependencies - Displays all dependencies declared in root project '
dependencyInsight - Displays the insight into a specific dependency
dependentComponents - Displays the dependent components of componen
help - Displays a help message.
model - Displays the configuration model of root project 'gs-gradle
outgoingVariants - Displays the outgoing variants of root project '
projects - Displays the sub-projects of root project 'gs-gradle'.
properties - Displays the properties of root project 'gs-gradle'.
tasks - Displays the tasks runnable from root project 'gs-gradle'.

To see all tasks and more detail, run gradle tasks --all

To see more detail about a task, run gradle help --task <task>

Deprecated Gradle features were used in this build, making it incom
Use '--warning-mode all' to show the individual deprecation warning
See https://docs.gradle.org/6.0.1/userguide/command_line_interface.
```

```
BUILD SUCCESSFUL in 477ms
1 actionable task: 1 executed
```

Even though these tasks are available, they don't offer much value without a project build configuration. As you flesh out the `build.gradle` file, some tasks will be more useful. The list of tasks will grow as you add plugins to `build.gradle`, so you'll occasionally want to run **tasks** again to see what tasks are available.

Speaking of adding plugins, next you add a plugin that enables basic Java build functionality.

Build Java code

Starting simple, create a very basic `build.gradle` file in the <project folder> you created at the beginning of this guide. Give it just just one line:

```
apply plugin: 'java'
```
COPY

This single line in the build configuration brings a significant amount of power. Run **gradle tasks** again, and you see new tasks added to the list, including tasks for building the project, creating JavaDoc, and running tests.

You'll use the **gradle build** task frequently. This task compiles, tests, and assembles the code into a JAR file. You can run it like this:

```
gradle build
```

After a few seconds, "BUILD SUCCESSFUL" indicates that the build has completed.

To see the results of the build effort, take a look in the *build* folder. Therein you'll find several directories, including these three notable folders:

- *classes*. The project's compiled .class files.
- *reports*. Reports produced by the build (such as test reports).
- *libs*. Assembled project libraries (usually JAR and/or WAR files).

The classes folder has .class files that are generated from compiling the Java code. Specifically, you should find HelloWorld.class and Greeter.class.

At this point, the project doesn't have any library dependencies, so there's nothing in the **dependency_cache** folder.

The reports folder should contain a report of running unit tests on the project. Because the project doesn't yet have any unit tests, that report will be uninteresting.

The libs folder should contain a JAR file that is named after the project's folder. Further down, you'll see how you can specify the name of the JAR and its version.

Declare dependencies

The simple Hello World sample is completely self-contained and does not depend on any additional libraries. Most applications, however, depend on external libraries to handle common and/or complex functionality.

For example, suppose that in addition to saying "Hello World!", you want the application to print the current date and time. You could use the date and time facilities in the native Java libraries, but you can make things more interesting by using the Joda Time libraries.

First, change HelloWorld.java to look like this:

COPY

```
package hello;

import org.joda.time.LocalTime;

public class HelloWorld {
  public static void main(String[] args) {
    LocalTime currentTime = new LocalTime();
    System.out.println("The current local time is: " +
currentTime);

    Greeter greeter = new Greeter();
    System.out.println(greeter.sayHello());
  }
}
```

Here `HelloWorld` uses Joda Time's `LocalTime` class to get and print the current time.

If you ran `gradle build` to build the project now, the build would fail because you have not declared Joda Time as a compile dependency in the build.

For starters, you need to add a source for 3rd party libraries.

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

COPY

The `repositories` block indicates that the build should resolve its dependencies from the Maven Central repository. Gradle leans heavily on many conventions and facilities established by the Maven build tool, including the option of using Maven Central as a source of library dependencies.

Now that we're ready for 3rd party libraries, let's declare some.

```
sourceCompatibility = 1.8
targetCompatibility = 1.8

dependencies {
    implementation "joda-time:joda-time:2.2"
    testImplementation "junit:junit:4.12"
}
```

COPY

With the `dependencies` block, you declare a single dependency for Joda Time. Specifically, you're asking for (reading right to left) version 2.2 of the joda-time library, in the joda-time group.

Another thing to note about this dependency is that it is a `compile` dependency, indicating that it should be available during compile-time (and if you were building a WAR file, included in the /WEB-INF/libs folder of the WAR). Other notable types of dependencies include:

- `implementation`. Required dependencies for compiling the project code, but that will be provided at runtime by a container running the code (for example, the Java Servlet API).

- `testImplementation`. Dependencies used for compiling and running tests, but not required for building or running the project's runtime code.

Finally, let's specify the name for our JAR artifact.

```
jar {
    archiveBaseName = 'gs-gradle'
    archiveVersion =  '0.1.0'
}
```

COPY

The `jar` block specifies how the JAR file will be named. In this case, it will render `gs-gradle-0.1.0.jar`.

Now if you run `gradle build`, Gradle should resolve the Joda Time dependency from the Maven Central repository and the build will succeed.

Build your project with Gradle Wrapper

The Gradle Wrapper is the preferred way of starting a Gradle build. It consists of a batch script for Windows and a shell script for OS X and Linux. These scripts allow you to run a Gradle build without requiring that Gradle be installed on your system. This used to be something added to your build file, but it's been folded into Gradle, so there is no longer any need. Instead, you simply use the following command.

```
$ gradle wrapper --gradle-version 6.0.1
```

After this task completes, you will notice a few new files. The two scripts are in the root of the folder, while the wrapper jar and properties files have been added to a new `gradle/wrapper` folder.

```
└── <project folder>
    └── gradlew
    └── gradlew.bat
    └── gradle
        └── wrapper
            └── gradle-wrapper.jar
            └── gradle-wrapper.properties
```

The Gradle Wrapper is now available for building your project. Add it to your version control system, and everyone that clones your project can build it just the same. It can be used in the exact same way as an installed version of Gradle. Run the wrapper script to perform the build task, just like you did previously:

```
./gradlew build
```

The first time you run the wrapper for a specified version of Gradle, it downloads and caches the Gradle binaries for that version. The Gradle Wrapper files are designed to be committed to source control so that anyone can build the project without having to first install and configure a specific version of Gradle.

At this stage, you will have built your code. You can see the results here:

```
build
├── classes
│   └── main
│       └── hello
│           ├── Greeter.class
│           └── HelloWorld.class
├── dependency-cache
├── libs
│   └── gs-gradle-0.1.0.jar
└── tmp
    └── jar
        └── MANIFEST.MF
```

Included are the two expected class files for `Greeter` and `HelloWorld`, as well as a JAR file. Take a quick peek:

```
$ jar tvf build/libs/gs-gradle-0.1.0.jar
  0 Fri May 30 16:02:32 CDT 2014 META-INF/
 25 Fri May 30 16:02:32 CDT 2014 META-INF/MANIFEST.MF
  0 Fri May 30 16:02:32 CDT 2014 hello/
369 Fri May 30 16:02:32 CDT 2014 hello/Greeter.class
988 Fri May 30 16:02:32 CDT 2014 hello/HelloWorld.class
```

The class files are bundled up. It's important to note, that even though you declared joda-time as a dependency, the library isn't included here. And the JAR file isn't runnable either.

To make this code runnable, we can use gradle's `application` plugin. Add this to your `build.gradle` file.

```
apply plugin: 'application'

mainClassName = 'hello.HelloWorld'
```

Then you can run the app!

```
$ ./gradlew run
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:run
The current local time is: 16:16:20.544
Hello world!

BUILD SUCCESSFUL

Total time: 3.798 secs
```

To bundle up dependencies requires more thought. For example, if we were building a WAR file, a format commonly associated with packing in 3rd party dependencies, we could use gradle's WAR plugin. If you are using Spring Boot and want a runnable JAR file, the spring-boot-gradle-plugin is quite handy. At this stage, gradle doesn't know enough about your system to make a choice. But for now, this should be enough to get started using gradle.

To wrap things up for this guide, here is the completed `build.gradle` file:

`build.gradle`

```
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'application'

mainClassName = 'hello.HelloWorld'

// tag::repositories[]
repositories {
    mavenCentral()
}
// end::repositories[]

// tag::jar[]
jar {
```

COPY

```
    archiveBaseName = 'gs-gradle'
    archiveVersion =  '0.1.0'
}
// end::jar[]

// tag::dependencies[]
sourceCompatibility = 1.8
targetCompatibility = 1.8

dependencies {
    implementation "joda-time:joda-time:2.2"
    testImplementation "junit:junit:4.12"
}
// end::dependencies[]

// tag::wrapper[]
// end::wrapper[]
```

There are many start/end comments embedded here. This makes it possible to extract bits of the build file into this guide for the detailed explanations above. You don't need them in your production build file.

Summary

Congratulations! You have now created a simple yet effective Gradle build file for building Java projects.

See Also

The following guide may also be helpful:

- Building Java Projects with Maven

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.