


Building apps using Gradle

Theory

Practice

 100% completed, 0 problems solved ▾

Theory

🕒 15 minutes reading

Unskip this topic

Start practicing

We hope you've already got a basic understanding of what Gradle is and how to use it. In this topic, we will consider the basic structure of the `build.gradle` file, and then build and run a small application. The knowledge you obtain here can be used for any JVM-based programming language supported by Gradle (e.g. Java or Kotlin).

This article was written using **Gradle 6.8.1**. There may be some differences for other versions of Gradle. If you have trouble with this article, you can read the comments or just follow the [official Gradle doc](#) instead of this.

§1. Initializing an application


We assume that you already have some experience with the terminal of your operating system and will interact with Gradle using it. First of all, create a new empty folder named what you want (e.g., `demo`). In this folder, you need to invoke the `gradle init` command to start initializing a new Gradle-based project. This command will show you a dialogue form to set up the project you need.

In this form, choose `application` as the type of the project; **Java** or **Kotlin** as the implementation language; and `org.hyperskill.gradleapp` as the project name if you would like to precisely follow our example (but it isn't required). For all other questions, you can choose their default options, since it doesn't matter now.

Below is an example of choosing options.


```
1  Select type of project to generate:
2    1: basic
3    2: application
4    3: library
5    4: Gradle plugin
6  Enter selection (default: basic) [1..4] 2
7
8  Select implementation language:
9    1: C++
10
11   2: Groovy
12
13   3: Java
14
15   4: Kotlin
16
17   5: Scala
18
19   6: Swift
20
21 Enter selection (default: Java) [1..6] 3
22
23 Split functionality across multiple subprojects?:
24
25   1: no - only one application project
```

1 required topic



✓ [Basic project with Gradle](#) ▾

1 dependent topic



✓ [Dependency management](#) ▾

Table of contents:

- [1 Building apps using Gradle](#)
- [§1. Initializing an application](#)
- [§2. Running the application](#)
- [§3. Plugins](#)
- [§4. Repositories and dependencies](#)
- [§5. Configurations for the application plugin](#)
- [§6. Generating and running Jar archive](#)
- [§7. Building the application](#)
- [§8. Conclusion](#)
- [Discussion](#)

```
1
9   2: yes - application and library projects
2
0   Enter selection (default: no - only one application project) [1..2]
2
1
2
2   Select build script DSL:
2
3   1: Groovy
2
4   2: Kotlin
2
5   Enter selection (default: Groovy) [1..2]
2
6
2
7   Select test framework:
2
8   1: JUnit 4
2
9   2: TestNG
3
0   3: Spock
3
1   4: JUnit Jupiter
3
2   Enter selection (default: JUnit 4) [1..4]
3
3
3
4   Project name (default: demo): org.hyperskill.gradleapp
3
5   Source package (default: org.hyperskill.gradleapp):
```

After the initialization is completed, the project structure will be the following:

```
1  .
2  ├── app
3  │   ├── build.gradle
4  │   └── src
5  │       ├── main
6  │       │   ├── java
7  │       │   │   ├── org
8  │       │   │   └── hyperskill
9  │       │       └── gradleapp
10 │       │           └── App.java
11 │       └── resources
12 │           └── test
13 │               ├── java
14 │               │   └── org
15 │               └── hyperskill
16 │                   └── gradleapp
17 │                       └── AppTest.java
18 │                           └── resources
19 └── gradle
20     └── wrapper
21         └── gradle-wrapper.jar
```

```
2 |           └─ gradle-wrapper.properties
2 |
3 | └─ gradlew
2 |
4 | └─ gradlew.bat
2 |
5 | └─ settings.gradle
```

This structure includes a lot of files we have already considered (`settings.gradle` , wrapper files, etc). The most important file called `build.gradle` , which contains tasks and external libraries, is located within the `app` directory. This folder exists because we've chosen `application` as the type of the project and the folder represents our application.

There is also the `src` directory inside `app` . It contains two sub-directories `main` and `test` . This is a quite standard project structure when using Gradle. In our case, the package `org.hyperskill.gradleapp` has some Java source code (`App.java`).

If you chose Kotlin as the implementation language, the project structure will be the same except for Kotlin source code files (`.kt` instead of `.java`) and `kotlin` folders instead of `java` ones.

Please note, it is a good practice for Java and Kotlin projects to include the name of your organization in the path to your source code files as a package name like `org.hyperskill` . We follow this recommendation too.

§2. Running the application

If you look at the list of available tasks for managing the project using the command `gradle tasks --all` , you will see that the list is fairly long. Here is a shortened version of it:

```
1 Application tasks
2 -----
3 run - Runs this project as a JVM application
4
5 Build tasks
6 -----
7 assemble - Assembles the outputs of this project.
8 build - Assembles and tests this project.
9 ...
```

You can use the `run` command to start the application. To do it, just invoke the `gradle run` command or use a Gradle wrapper script for your OS. This command will build and run the application. Here is an output example:

```
1 > Task :app:run
2 Hello World!
3
4 BUILD SUCCESSFUL in 623ms
5 2 actionable tasks: 1 executed, 1 up-to-date
```

As you can see, the autogenerated application can already display a welcome string. If you get a similar result, it means that everything is OK: your application works and Gradle can manage it!

If you look at the project structure again, you will see that it has some new files, including files with bytecode (`App.class` , `AppTest.class`). Actually, Gradle built and started the `App.class` file when we invoked the `run` command.

Now, let's consider the build file (`build.gradle` for Groovy DSL or `build.gradle.kts` for Kotlin DSL) thanks to which we can build our application

successfully and run it using Gradle. This file specifies the project structure and adds some tasks and external libraries to the project. We will not present the entire file here, only its main parts.

§3. Plugins

The `plugins` section adds some plugins to extend the capabilities of the project: e.g., to add new tasks or properties.

```
1  plugins {  
2      // Apply the application plugin to add support for building a  
CLI application  
3      id("application")  
4  
5      // Apply the plugin which adds support for Java  
6      id("java")  
7  
8      // Apply the plugin which adds support for Kotlin/JVM  
9      id("org.jetbrains.kotlin.jvm")  
1  
0  }
```

Here, `id` is a globally unique identifier, or name, for plugins. Core Gradle plugins are special in that they provide short names, such as `"java"` or `"application"`.

Basically, plugins for Kotlin and Java know how to build, package, and run tests on the project. The `application` plugin facilitates creating an executable JVM application.

There is an alternative way to use a plugin in the project. It's more like a legacy way of applying plugins which is not widely used now, but just in case you see it somewhere, here it is:

```
1  apply plugin: "application" // for Groovy DSL  
2  apply(plugin = "application") // for Kotlin DSL
```

There are many other plugins already available for you, and you can find them on the [official Gradle Plugins page](#). A large project can use dozens and hundreds of them. Gradle does not limit the maximum number of plugins used in a project.

§4. Repositories and dependencies

Usually, you don't need to write your program from scratch – you use already written pieces of code, either yours or other developers'. This is where the dependency system comes in handy.

The `repositories` section declares locations from which dependencies will be downloaded and added to the project.

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

There are plenty of public repositories: **JCenter**, **Maven Central**, **Google**, and others. Usually, a description of a dependency says which repository contains it.

The `dependencies` section is used to add external libraries to the project. Gradle will automatically download them from the repositories and put them in the archive with the application. Right now your `dependencies` section should contain at least a testing library like `JUnit` or something else, depending on your choice when the project was initialized.

```
1 dependencies {
2     // Use JUnit test framework.
3     testImplementation 'junit:junit:4.13'
4
5     // This dependency is used by the application.
6     implementation 'com.google.guava:guava:29.0-jre'
7 }
```

We will take a closer look at repositories and dependencies in the next topics.

This is a standard Gradle build structure. You apply some plugins and specify dependencies for your project. This structure will be the same for any project managed by Gradle.

§5. Configurations for the application plugin

The auto-generated `build.gradle(.kts)` file has a section that configures the `application` plugin thanks to which the application runs with the `gradle run` command as mentioned above.

```
1 application {
2     // Defines the main class for the application
3     mainClassName = "org.hyperskill.gradleapp.App"
4 }
```

The `mainClassName` property defines a class with the entry point of the application. It allows us to run the application invoking the `gradle run` command.

§6. Generating and running Jar archive

The classic way to run a JVM-based application is to use the `java -jar` command. This command can be run without Gradle, you only need to have a JAR beforehand.

So let's build the JAR file for our application:

```
1 gradle jar
2
3 BUILD SUCCESSFUL in 748ms
4 2 actionable tasks: 2 executed
```

Now, the JAR file is in the `app/build/libs` directory.

If you want to clean the project folder from all generated artifacts, just run the `gradle clean` command.

However, if you now try to run our generated application using the classic approach, there will be a problem:

```
1 java -jar app/build/libs/app.jar
2 no main manifest attribute, in app/build/libs/app.jar
```

The thing is that the application does not contain the `Main-Class` attribute in the `MANIFEST.MF` file. So, the JVM does not know the path to the entry point of the application.

To fix this we need to add the required attribute when generating an archive for the application. Just add the following declaration to the `build.gradle(.kts)` file:

```
1 jar {
2     manifest {
3         attributes("Main-Class": "org.hyperskill.gradleapp.App")
4     }
5 }
```

```
// for Groovy DSL
4     attributes("Main-Class" to "org.hyperskill.gradleapp.App")
// for Kotlin DSL
5     }
6 }
```

This code adds the `Main-Class` attribute to the manifest property of the jar task. See the manifest as a map of properties where we put our pair `Main-Class -> Main`.

So, now when we execute `gradle jar` followed by `java -jar app/build/libs/app.jar`, everything should work as planned and you will see the output line `Hello world!`. What is good about this way of running applications is that `java -jar` command can be run without Gradle, you only need to have a JAR beforehand.

§7. Building the application

If you would like to generate a bundle of the application with all its dependencies and a script to start the application, use the `gradle build` command.

```
1 BUILD SUCCESSFUL in 1s
2 7 actionable tasks: 7 executed
```

If everything is OK, Gradle will have produced the archive in two formats for you: `app/build/distributions/app.tar` and `app/build/distributions/app.zip`. Now, you can distribute your application!

§8. Conclusion

In this topic, you've learned how to generate Gradle-based applications with source code in Java or Kotlin as well as how to run this application using both `gradle run` and `java -jar` commands. You have also become familiar with the basic structure of the `build.gradle(.kts)` file and got initial information about plugins, repositories, and dependencies. You will learn more about these things further on.

 Report a typo

414 users liked this piece of theory. 91 didn't like it. What about you?



Start practicing

Unskip this topic

Comments (97)

Useful links (2)

Show discussion

 JetBrains Academy

Tracks

About

 Become beta tester

Pricing

Contribute

Be the first to see what's new

For organizations

Careers

Terms Support  

Made with  by Hyperskill and JetBrains