# **Building Java Applications Sample**

Version 8.2.1

#### **Contents**

What you'll build

What you'll need

Create a project folder

Run the init task

Review the project files

Run the application

Bundle the application

Publish a Build Scan

Summary

Next steps

Kotlin DSL **Groovy DSL** 

> You can open this sample inside an IDE using the IntelliJ native importer or Eclipse Buildship.

This guide demonstrates how to create a Java application with Gradle using gradle init. You can follow the guide step-by-step to create a new project from scratch or download the complete sample project using the links above.

# What you'll build

You'll generate a Java application that follows Gradle's conventions.

# What you'll need

- A text editor or IDE for example IntelliJ IDEA
- A Java Development Kit (JDK), version 8 or higher for example AdoptOpenJDK
- The latest Gradle distribution

# Create a project folder

Gradle comes with a built-in task, called init, that initializes a new Gradle project in an empty folder. The init task uses the (also built-in) wrapper task to create a Gradle wrapper script, gradlew.

The first step is to create a folder for the new project and change directory into it.

```
$ mkdir demo
```

\$ cd demo

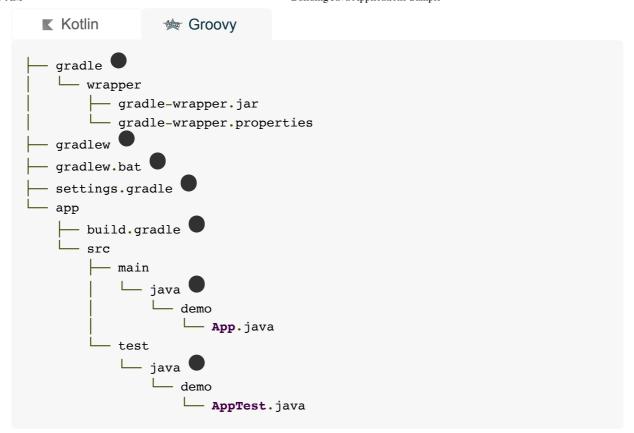
#### Run the init task

From inside the new project directory, run the init task using the following command in a terminal: gradle init. When prompted, select the 2: application project type and 3: Java as implementation language. Next you can choose the DSL for writing buildscripts - 1 : Groovy or 2: Kotlin. For the other questions, press enter to use the default values.

The output will look like this:

```
$ gradle init
Select type of project to generate:
 1: basic
 2: application
  3: library
  4: Gradle plugin
Enter selection (default: basic) [1..4] 2
Select implementation language:
 1: C++
  2: Groovy
 3: Java
  4: Kotlin
  5: Scala
  6: Swift
Enter selection (default: Java) [1..6] 3
Select build script DSL:
 1: Groovy
  2: Kotlin
Enter selection (default: Groovy) [1..2] 1
Select test framework:
  1: JUnit 4
 2: TestNG
  3: Spock
  4: JUnit Jupiter
Enter selection (default: JUnit 4) [1..4]
Project name (default: demo):
Source package (default: demo):
BUILD SUCCESSFUL
2 actionable tasks: 2 executed
```

The init task generates the new project with the following structure:



- Generated folder for wrapper files
- Gradle wrapper start scripts
- Settings file to define build name and subprojects
- Build script of app project
- Default Java source folder
- Default Java test source folder

You now have the project setup to build a Java application.

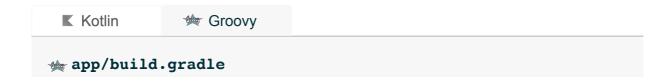
### Review the project files

The settings.gradle(.kts) file has two interesting lines:

```
settings.gradle
rootProject.name = 'demo'
include('app')
```

- rootProject.name assigns a name to the build, which overrides the default behavior of naming the build after the directory it's in. It's recommended to set a fixed name as the folder might change if the project is shared - e.g. as root of a Git repository.
- include("app") defines that the build consists of one subproject called app that contains the actual code and build logic. More subprojects can be added by additional include(...) statements.

Our build contains one subproject called app that represents the Java application we are building. It is configured in the app/build.gradle(.kts) file:



```
plugins {
    id 'application'
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.9.2'

    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'

    implementation 'com.google.guava:guava:31.1-jre'
}

application {
    mainClass = 'demo.App'
}

tasks.named('test') {
    useJUnitPlatform()
}
```

- Apply the application plugin to add support for building a CLI application in Java.
- Use Maven Central for resolving dependencies.
- Use JUnit Jupiter for testing.
- This dependency is used by the application.
- Define the main class for the application.
- Use JUnit Platform for unit tests.

The file src/main/java/demo/App.java is shown here:

#### Generated src/main/java/demo/App.java

```
* This Java source file was generated by the Gradle 'init' task.
package demo;
public class App {
    public String getGreeting() {
        return "Hello World!";
    public static void main(String[] args) {
        System.out.println(new App().getGreeting());
    }
}
```

The generated test, src/test/java/demo/App.java is shown next:

#### Generated src/test/java/demo/AppTest.java

```
* This Java source file was generated by the Gradle 'init' task.
package demo;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
class AppTest {
    @Test void appHasAGreeting() {
        App classUnderTest = new App();
        assertNotNull(classUnderTest.getGreeting(), "app should have a greeti
}
```

The generated test class has a single JUnit Jupiter test. The test instantiates the App class, invokes a method on it, and checks that it returns the expected value.

## Run the application

Thanks to the application plugin, you can run the application directly from the command line. The run task tells Gradle to execute the main method in the class assigned to the mainClass property.

```
$ ./gradlew run
> Task :app:run
Hello world!
BUILD SUCCESSFUL
2 actionable tasks: 2 executed
```

The first time you run the wrapper script, gradlew, there may be a delay while that version of gradle is downloaded and stored locally in your ~/.gradle/wrapper/dists folder.

## Bundle the application

The application plugin also bundles the application, with all its dependencies, for you. The archive will also contain a script to start the application with a single command.

```
$ ./gradlew build
BUILD SUCCESSFUL in 0s
7 actionable tasks: 7 executed
```

If you run a full build as shown above, Gradle will have produced the archive in two formats for

you: app/build/distributions/app.tar and app/build/distributions/app

#### Publish a Build Scan

The best way to learn more about what your build is doing behind the scenes, is to publish a build scan. To do so, just run Gradle with the --scan flag.

```
$ ./gradlew build --scan
BUILD SUCCESSFUL in 0s
7 actionable tasks: 7 executed
Publishing a build scan to scans.gradle.com requires accepting the Gradle Ter
Do you accept these terms? [yes, no] yes
Gradle Terms of Service accepted.
Publishing build scan...
https://gradle.com/s/5u4w3gxeurtd2
```

Click the link and explore which tasks where executed, which dependencies where downloaded and many more details!

## **Summary**

That's it! You've now successfully configured and built a Java application project with Gradle. You've learned how to:

- Initialize a project that produces a Java application
- Run the build and view the test report
- Execute a Java application using the run task from the application plugin
- Bundle the application in an archive

# **Next steps**

To learn more about how you can further customize Java application projects, check out the following user manual chapters:

- Building Java & JVM projects
- Java Application Plugin documentation