Spring Boot Gradle Plugin

Last modified: July 20, 2020

by baeldung (https://www.baeldung.com/author/baeldung/)

- DevOps (https://www.baeldung.com/category/devops/)
- Gradle (https://www.baeldung.com/category/gradle/)
- Spring Boot (https://www.baeldung.com/category/spring/spring-boot/)

1. Overview

The Spring Boot Gradle plugin helps us manage Spring Boot dependencies, as well as package and run our application when using Gradle as a build tool.

In this tutorial, we'll discuss how we can add and configure the plugin, and then we'll see how to build and run a Spring Boot project.

2. Build File Configuration

First, **we need to add the Spring Boot plugin to our *build.gradle*** file by including it in our *plugins* section:

```
plugins {
    id "org.springframework.boot" version "2.0.1.RELEASE"
}
```

If we're using a Gradle version earlier than 2.1 or we need dynamic configuration, we can add it like this instead:

```
buildscript {
    ext {
        springBootVersion = '2.0.1.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath(
          "org.springframework.boot:spring-boot-gradle-plugin:${springBoot
    }
}

apply plugin: 'org.springframework.boot'
```

3. Packaging Our Application

We can package our application to an executable archive (jar or war file) by building it using the *build* command:

```
./gradlew build
```

As a result, the generated executable archive will be placed in the *build/libs* directory.

If we want to generate an executable *jar* file, then we also need to apply the *java* plugin:

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

On the other hand, if we need a *war* file, we'll apply the *war* plugin:

```
apply plugin: 'war'
```

Building the application will generate executable archives for both Spring Boot 1.x and 2.x. However, for each version, Gradle triggers different tasks.

Next, let's have a closer look at the build process for each Boot version.

3.1. Spring Boot 2.x

In Boot 2.x, the *bootJar* and *bootWar* tasks are responsible for packaging the application.

The *bootJar* task is responsible for creating the executable *jar* file. This is created automatically once the *java* plugin is applied.

Let's see how we can execute the *bootJar* task directly:

```
./gradlew bootJar
```

Similarly, *bootWar* generates an executable war file and gets created once the *war* plugin is applied.

We can execute the *bootWar* task using:

```
./gradlew bootWar
```

Note that for Spring Boot 2.x, we need to use Gradle 4.0 or later.

We can also configure both tasks. For example, let's set the main class by using the *mainClassName* property:

```
bootJar {
    mainClassName = 'com.baeldung.Application'
}
```

Alternatively, we can use use the same property from the Spring Boot DSL:

```
springBoot {
    mainClassName = 'com.baeldung.Application'
}
```

3.2. Spring Boot 1.x

With Spring Boot 1.x, *bootRepackage* is responsible for creating the executable archive *(jar* or *war* file depending on the configuration.

We can execute the *bootRepackage* task directly using:

```
./gradlew bootRepackage
```

Similar to the Boot 2.x version, we can add configurations to the *bootRepackage* task in our *build.gradle:*

```
bootRepackage {
    mainClass = 'com.example.demo.Application'
}
```

We can also disable the *bootRepackage* task by setting the *enabled* option to *false:*

```
bootRepackage {
    enabled = false
}
```

4. Running Our Application

After building the application, **we can just run it by using the *java -jar* command** on the generated executable jar file:

```
java -jar build/libs/demo.jar
```

Spring Boot Gradle plugin also provides us with the *bootRun* task which enables us to run the application without the need to build it first:

```
./gradlew bootRun
```

The *bootRun* task can be simply configured in *build.gradle.*

For example, we can define the main class:

```
bootRun {
    main = 'com.example.demo.Application'
}
```

5. Relation With Other Plugins

5.1. Dependency Management Plugin

For Spring Boot 1.x, it used to apply the dependency management plugin automatically. This would import the Spring Boot dependencies BOM and act similar to dependency management for Maven.

But since Spring Boot 2.x, we need to apply it explicitly in our *build.gradle* if we need this functionality:

```
apply plugin: 'io.spring.dependency-management'
```

5.2. Java Plugin

When we apply the *java* plugin, the Spring Boot Gradle plugin takes multiple actions like:

- creating *a bootJar* task, which we can use to generate an executable jar file
- creating *a bootRun* task, which we can use to run our application directly

- disabling *jar* task

5.3. War Plugin

Similarly, when we apply the *war* plugin, that results in:

- creating the *bootWar* task, which we can use to generate an executable war file
- disabling the *war* task

6. Conclusion

In this quick tutorial, we learned about the Spring Boot Gradle Plugin and its different tasks.

Also, we discussed how it interacts with other plugins.