Spring Boot Gradle Plugin Reference Guide

Andy Wilkinson

Table of Contents

1. Introduction . . . 1
2. Getting started . . . 2
3. Managing dependencies . . . 3
   3.1. Customizing managed versions . . . 3
   3.2. Using Spring Boot's dependency management in isolation . . . 3
   3.3. Learning more . . . 4
4. Packaging executable archives . . . 5
   4.1. Packaging executable jars . . . 5
   4.2. Packaging executable wars . . . 5
      4.2.1. Packaging executable and deployable wars . . . 5
   4.3. Packaging executable and normal archives . . . 5
   4.4. Configuring executable archive packaging . . . 6
      4.4.1. Configuring the main class . . . 6
      4.4.2. Excluding Devtools . . . 7
      4.4.3. Configuring libraries that require unpacking . . . 7
      4.4.4. Making an archive fully executable . . . 7
      4.4.5. Using the `PropertiesLauncher` . . . 8
5. Publishing your application . . . 9
   5.1. Publishing with the `maven` plugin . . . 9
   5.2. Publishing with the `maven-publish` plugin . . . 9
   5.3. Distributing with the `application` plugin . . . 9
6. Running your application with Gradle . . . 10
   6.1. Passing arguments to your application . . . 10
   6.2. Reloading resources . . . 11
7. Integrating with Actuator . . . 12
   7.1. Generating build information . . . 12
8. Reacting to other plugins . . . 14
   8.1. Reacting to the Java plugin . . . 14
   8.2. Reacting to the Kotlin plugin . . . 14
   8.3. Reacting to the war plugin . . . 14
   8.4. Reacting to the dependency management plugin . . . 14
   8.5. Reacting to the application plugin . . . 15
   8.6. Reacting to the Maven plugin . . . 15

Chapter 1. Introduction

The Spring Boot Gradle Plugin provides Spring Boot support in Gradle, allowing you to package executable jar or war archives, run Spring Boot applications, and use the dependency management provided by `spring-boot-dependencies`. Spring Boot's Gradle plugin requires Gradle 4.4 or later.

In addition to this user guide, API documentation is also available.

Chapter 2. Getting started

To get started with the plugin it needs to be applied to your project.

```
buildscript {
    repositories {
        maven { url 'https://repo.spring.io/libs-milestone' }
    }

    dependencies {
        classpath 'org.springframework.boot:spring-boot-gradle-plugin:2.1.0.M3'
    }
}

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

Applied in isolation the plugin makes few changes to a project. Instead, the plugin detects when certain other plugins are applied and reacts accordingly. For example, when the `java` plugin is applied a task for building an executable jar is automatically configured.

A typical Spring Boot project will apply the `groovy`, `java`, or `org.jetbrains.kotlin.jvm` plugin and the `io.spring.dependency-management` plugin as a minimum. For example:

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

To learn more about how the Spring Boot plugin behaves when other plugins are applied please see the section on reacting to other plugins.

Chapter 3. Managing dependencies

When you apply the `io.spring.dependency-management` plugin, Spring Boot's plugin will automatically import the `spring-boot-dependencies` bom from the version of Spring Boot that you are using. This provides a similar dependency management experience to the one that's enjoyed by Maven users. For example, it allows you to omit version numbers when declaring dependencies that are managed in the bom. To make use of this functionality, simply declare dependencies in the usual way but omit the version number:

```
dependencies {
    compile 'org.springframework.boot:spring-boot-starter-web'
    compile 'org.springframework.boot:spring-boot-starter-data-jpa'
}
```

3.1. Customizing managed versions

The `spring-boot-dependencies` bom that is automatically imported when the dependency management plugin is applied uses properties to control the versions of the dependencies that it manages. Please refer to the bom for a complete list of these properties.

To customize a managed version you set its corresponding property. For example, to customize the version of SLF4J which is controlled by the `slf4j.version` property:

```
ext['slf4j.version'] = '1.7.20'
```

Each Spring Boot release is designed and tested against a specific set of third-party dependencies. Overriding versions may cause compatibility issues and should be done with care.

3.2. Using Spring Boot's dependency management in isolation

Spring Boot's dependency management can be used in a project without applying Spring Boot's plugin to that project. The `SpringBootPlugin` class provides a `BOM_COORDINATES` constant that can be used to import the bom without having to know its group ID, artifact ID, or version.

First, configure the project to depend on the Spring Boot plugin but do not apply it:

```
buildscript {
    repositories {
        maven { url 'https://repo.spring.io/libs-milestone' }
    }

    dependencies {
        classpath 'org.springframework.boot:spring-boot-gradle-plugin:2.1.0.M3'
    }
}
```

The Spring Boot plugin's dependency on the dependency management plugin means that you can use the dependency management plugin without having to declare a dependency on it. This also means that you will automatically use the same version of the dependency management plugin as Spring Boot uses.

Apply the dependency management plugin and then configure it to import Spring Boot's bom:

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

dependencyManagement {
    imports {
        mavenBom
org.springframework.boot.gradle.plugin.SpringBootPlugin.BOM_COORDINATES
    }
}
```

3.3. Learning more

To learn more about the capabilities of the dependency management plugin, please refer to its documentation.

Chapter 4. Packaging executable archives

The plugin can create executable archives (jar files and war files) that contain all of an application's dependencies and can then be run with `java -jar`.

4.1. Packaging executable jars

Executable jars can be built using the `bootJar` task. The task is automatically created when the `java` plugin is applied and is an instance of `BootJar`. The `assemble` task is automatically configured to depend upon the `bootJar` task so running `assemble` (or `build`) will also run the `bootJar` task.

4.2. Packaging executable wars

Executable wars can be built using the `bootWar` task. The task is automatically created when the `war` plugin is applied and is an instance of `BootWar`. The `assemble` task is automatically configured to depend upon the `bootWar` task so running `assemble` (or `build`) will also run the `bootWar` task.

4.2.1. Packaging executable and deployable wars

A war file can be packaged such that it can be executed using `java -jar` and deployed to an external container. To do so, the embedded servlet container dependencies should be added to the `providedRuntime` configuration, for example:

```
dependencies {
    compile 'org.springframework.boot:spring-boot-starter-web'
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
}
```

This ensures that they are package in the war file's `WEB-INF/lib-provided` directory from where they will not conflict with the external container's own classes.

> `providedRuntime` is preferred to Gradle's `compileOnly` configuration as, among other limitations, `compileOnly` dependencies are not on the test classpath so any web-based integration tests will fail.

4.3. Packaging executable and normal archives

By default, when the `bootJar` or `bootWar` tasks are configured, the `jar` or `war` tasks are disabled. A project can be configured to build both an executable archive and a normal archive at the same time by enabling the `jar` or `war` task:

```
jar {
    enabled = true
}
```

To avoid the executable archive and the normal archive from being written to the same location, one or the other should be configured to use a different location. One way to do so is by configuring a classifier:

```
bootJar {
    classifier = 'boot'
}
```

4.4. Configuring executable archive packaging

The `BootJar` and `BootWar` tasks are subclasses of Gradle's `Jar` and `War` tasks respectively. As a result, all of the standard configuration options that are available when packaging a jar or war are also available when packaging an executable jar or war. A number of configuration options that are specific to executable jars and wars are also provided.

4.4.1. Configuring the main class

By default, the executable archive's main class will be configured automatically by looking for a class with a `public static void main(String[])` method in directories on the task's classpath.

The main class can also be configured explicitly using the task's `mainClassName` property:

```
bootJar {
    mainClassName = 'com.example.ExampleApplication'
}
```

Alternatively, the main class name can be configured project-wide using the `mainClassName` property of the Spring Boot DSL:

```
springBoot {
    mainClassName = 'com.example.ExampleApplication'
}
```

If the `application` plugin has been applied its `mainClassName` project property can be used for the same purpose:

```
mainClassName = 'com.example.ExampleApplication'
```

Lastly, the `Start-Class` attribute can be configured on the task's manifest:

```
bootJar {
    manifest {
        attributes 'Start-Class': 'com.example.ExampleApplication'
    }
}
```

4.4.2. Excluding Devtools

By default, Spring Boot's Devtools module, `org.springframework.boot:spring-boot-devtools`, will be excluded from an executable jar or war. If you want to include Devtools in your archive set the `excludeDevtools` property to `false`:

```
bootWar {
    excludeDevtools = false
}
```

4.4.3. Configuring libraries that require unpacking

Most libraries can be used directly when nested in an executable archive, however certain libraries can have problems. For example, JRuby includes its own nested jar support which assumes that `jruby-complete.jar` is always directly available on the file system.

To deal with any problematic libraries, an executable archive can be configured to unpack specific nested jars to a temporary folder when the executable archive is run. Libraries can be identified as requiring unpacking using Ant-style patterns that match against the absolute path of the source jar file:

```
bootJar {
    requiresUnpack '**/jruby-complete-*.jar'
}
```

For more control a closure can also be used. The closure is passed a `FileTreeElement` and should return a `boolean` indicating whether or not unpacking is required.

4.4.4. Making an archive fully executable

Spring Boot provides support for fully executable archives. An archive is made fully executable by prepending a shell script that knows how to launch the application. On Unix-like platforms, this launch script allows the archive to be run directly like any other executable or to be installed as a service.

To use this feature, the inclusion of the launch script must be enabled:

```
bootJar {
    launchScript()
}
```

This will add Spring Boot's default launch script to the archive. The default launch script includes several properties with sensible default values. The values can be customized using the `properties` property:

```
bootJar {
    launchScript {
        properties 'logFilename': 'example-app.log'
    }
}
```

If the default launch script does not meet your needs, the `script` property can be used to provide a custom launch script:

```
bootJar {
    launchScript {
        script = file('src/custom.script')
    }
}
```

4.4.5. Using the `PropertiesLauncher`

To use the `PropertiesLauncher` to launch an executable jar or war, configure the task's manifest to set the `Main-Class` attribute:

```
bootWar {
    manifest {
        attributes 'Main-Class': 'org.springframework.boot.loader.PropertiesLauncher'
    }
}
```

Chapter 5. Publishing your application

5.1. Publishing with the `maven` plugin

When the `maven` plugin is applied, an `Upload` task for the `bootArchives` configuration named `uploadBootArchives` is automatically created. By default, the `bootArchives` configuration contains the archive produced by the `bootJar` or `bootWar` task. The `uploadBootArchives` task can be configured to publish the archive to a Maven repository:

```
uploadBootArchives {
    repositories {
        mavenDeployer {
            repository url: 'https://repo.example.com'
        }
    }
}
```

5.2. Publishing with the `maven-publish` plugin

To publish your Spring Boot jar or war, add it to the publication using the `artifact` method on `MavenPublication`. Pass the task that produces that artifact that you wish to publish to the `artifact` method. For example, to publish the artifact produced by the default `bootJar` task:

```
publishing {
    publications {
        bootJava(MavenPublication) {
            artifact bootJar
        }
    }
    repositories {
        maven {
            url 'https://repo.example.com'
        }
    }
}
```

5.3. Distributing with the `application` plugin

When the `application` plugin is applied a distribution named `boot` is created. This distribution contains the archive produced by the `bootJar` or `bootWar` task and scripts to launch it on Unix-like platforms and Windows. Zip and tar distributions can be built by the `bootDistZip` and `bootDistTar` tasks respectively.

Chapter 6. Running your application with Gradle

To run your application without first building an archive use the `bootRun` task:

```
$ ./gradlew bootRun
```

The `bootRun` task is an instance of `BootRun` which is a `JavaExec` subclass. As such, all of the usual configuration options for executing a Java process in Gradle are available to you. The task is automatically configured to use the runtime classpath of the main source set.

By default, the main class will be configured automatically by looking for a class with a `public static void main(String[])` method in directories on the task's classpath.

The main class can also be configured explicitly using the task's `main` property:

```
bootRun {
    main = 'com.example.ExampleApplication'
}
```

Alternatively, the main class name can be configured project-wide using the `mainClassName` property of the Spring Boot DSL:

```
springBoot {
    mainClassName = 'com.example.ExampleApplication'
}
```

If the `application` plugin has been applied, its `mainClassName` project property can be used for the same purpose:

```
mainClassName = 'com.example.ExampleApplication'
```

6.1. Passing arguments to your application

Like all `JavaExec` tasks, arguments can be passed into `bootRun` from the command line using `--args='<arguments>'` when using Gradle 4.9 or later. For example, to run your application with a profile named `dev` active the following command can be used:

```
$ ./gradlew bootRun --args='--spring.profiles.active=dev'
```

See the javadoc for `JavaExec.setArgsString` for further details.

6.2. Reloading resources

If devtools has been added to your project it will automatically monitor your application for changes. Alternatively, you can configure `bootRun` such that your application's static resources are loaded from their source location:

```
bootRun {
    sourceResources sourceSets.main
}
```

This makes them reloadable in the live application which can be helpful at development time.

Chapter 7. Integrating with Actuator

7.1. Generating build information

Spring Boot Actuator's `info` endpoint automatically publishes information about your build in the presence of a `META-INF/build-info.properties` file. A `BuildInfo` task is provided to generate this file. The easiest way to use the task is via the plugin's DSL:

```
springBoot {
    buildInfo()
}
```

This will configure a `BuildInfo` task named `bootBuildInfo` and, if it exists, make the Java plugin's `classes` task depend upon it. The task's destination directory will be `META-INF` in the output directory of the main source set's resources (typically `build/resources/main`).

By default, the generated build information is derived from the project:

Property	Default value
`build.artifact`	The base name of the `bootJar` or `bootWar` task, or `unspecified` if no such task exists
`build.group`	The group of the project
`build.name`	The name of the project
`build.version`	The version of the project
`build.time`	The time at which the project is being built

The properties can be customized using the DSL:

```
springBoot {
    buildInfo {
        properties {
            artifact = 'example-app'
            version = '1.2.3'
            group = 'com.example'
            name = 'Example application'
        }
    }
}
```

The default value for `build.time` is the instant at which the project is being built. A side-effect of this is that the task will never be up-to-date. As a result, builds will take longer as more tasks, including the project's tests, will have to be executed. Another side-effect is that the task's output will always change and, therefore, the build will not be truly repeatable. If you value build performance or

repeatability more highly than the accuracy of the `build.time` property, set `time` to `null` or a fixed value.

Additional properties can also be added to the build information:

```
springBoot {
    buildInfo {
        properties {
            additional = [
                'a': 'alpha',
                'b': 'bravo'
            ]
        }
    }
}
```

Chapter 8. Reacting to other plugins

When another plugin is applied the Spring Boot plugin reacts by making various changes to the project's configuration. This section describes those changes.

8.1. Reacting to the Java plugin

When Gradle's `java` plugin is applied to a project, the Spring Boot plugin:

1. Creates a `BootJar` task named `bootJar` that will create an executable, fat jar for the project. The jar will contain everything on the runtime classpath of the main source set; classes are packaged in `BOOT-INF/classes` and jars are packaged in `BOOT-INF/lib`
2. Configures the `assemble` task to depend on the `bootJar` task.
3. Disables the `jar` task.
4. Creates a `BootRun` task named `bootRun` that can be used to run your application.
5. Creates a configuration named `bootArchives` that contains the artifact produced by the `bootJar` task.
6. Configures any `JavaCompile` tasks with no configured encoding to use `UTF-8`.
7. Configures any `JavaCompile` tasks to use the `-parameters` compiler argument.

8.2. Reacting to the Kotlin plugin

When Kotlin's Gradle plugin is applied to a project, the Spring Boot plugin:

1. Aligns the Kotlin version used in Spring Boot's dependency management with the version of the plugin. This is achieved by setting the `kotlin.version` property with a value that matches the version of the Kotlin plugin.
2. Configures any `KotlinCompile` tasks to use the `-java-parameters` compiler argument.

8.3. Reacting to the war plugin

When Gradle's `war` plugin is applied to a project, the Spring Boot plugin:

1. Creates a `BootWar` task named `bootWar` that will create an executable, fat war for the project. In addition to the standard packaging, everything in the `providedRuntime` configuration will be packaged in `WEB-INF/lib-provided`.
2. Configures the `assemble` task to depend on the `bootWar` task.
3. Disables the `war` task.
4. Configures the `bootArchives` configuration to contain the artifact produced by the `bootWar` task.

8.4. Reacting to the dependency management plugin

When the `io.spring.dependency-management` plugin is applied to a project, the Spring Boot plugin will

automatically import the `spring-boot-dependencies` bom.

8.5. Reacting to the application plugin

When Gradle's `application` plugin is applied to a project, the Spring Boot plugin:

1. Creates a `CreateStartScripts` task named `bootStartScripts` that will create scripts that launch the artifact in the `bootArchives` configuration using `java -jar`. The task is configured to use the `applicationDefaultJvmArgs` property as a convention for its `defaultJvmOpts` property.
2. Creates a new distribution named `boot` and configures it to contain the artifact in the `bootArchives` configuration in its `lib` directory and the start scripts in its `bin` directory.
3. Configures the `bootRun` task to use the `mainClassName` property as a convention for its `main` property.
4. Configures the `bootRun` task to use the `applicationDefaultJvmArgs` property as a convention for its `jvmArgs` property.
5. Configures the `bootJar` task to use the `mainClassName` property as a convention for the `Start-Class` entry in its manifest.
6. Configures the `bootWar` task to use the `mainClassName` property as a convention for the `Start-Class` entry in its manifest.

8.6. Reacting to the Maven plugin

When Gradle's `maven` plugin is applied to a project, the Spring Boot plugin will configure the `uploadBootArchives Upload` task to ensure that no dependencies are declared in the pom that it generates.