Spring Boot Maven Plugin Documentation

Stephane Nicoll, Andy Wilkinson, Scott Frederick

Table of Contents

Chapter 1. Introduction

The Spring Boot Maven Plugin provides Spring Boot support in Apache Maven. It allows you to package executable jar or war archives, run Spring Boot applications, generate build information and start your Spring Boot application prior to running integration tests.

Chapter 2. Getting Started

To use the Spring Boot Maven Plugin, include the appropriate XML in the `plugins` section of your `pom.xml`, as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <!-- ... -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

If you use a milestone or snapshot release, you also need to add the appropriate `pluginRepository` elements, as shown in the following listing:

```
<pluginRepositories>
    <pluginRepository>
        <id>spring-snapshots</id>
        <url>https://repo.spring.io/snapshot</url>
    </pluginRepository>
    <pluginRepository>
        <id>spring-milestones</id>
        <url>https://repo.spring.io/milestone</url>
    </pluginRepository>
</pluginRepositories>
```

Chapter 3. Using the Plugin

Maven users can inherit from the `spring-boot-starter-parent` project to obtain sensible defaults. The parent project provides the following features:

- Java 1.8 as the default compiler level.
- UTF-8 source encoding.
- A dependency management section, inherited from the `spring-boot-dependencies` POM, that manages the versions of common dependencies. This dependency management lets you omit `<version>` tags for those dependencies when used in your own POM.
- An execution of the `repackage` goal with a `repackage` execution id.
- Sensible resource filtering.
- Sensible plugin configuration (Git commit ID, and shade).
- Sensible resource filtering for `application.properties` and `application.yml` including profile-specific files (for example, `application-dev.properties` and `application-dev.yml`)

Since the `application.properties` and `application.yml` files accept Spring style placeholders (`${⋯}`), the Maven filtering is changed to use `@..@` placeholders. (You can override that by setting a Maven property called `resource.delimiter`.)

3.1. Inheriting the Starter Parent POM

To configure your project to inherit from the `spring-boot-starter-parent`, set the `parent` as follows:

```
<!-- Inherit defaults from Spring Boot -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.7</version>
</parent>
```

You should need to specify only the Spring Boot version number on this dependency. If you import additional starters, you can safely omit the version number.

With that setup, you can also override individual dependencies by overriding a property in your own project. For instance, to use a different version of the SLF4J library and the Spring Data release train, you would add the following to your `pom.xml`:

```
<properties>
    <slf4j.version>1.7.30</slf4j.version>
    <spring-data-releasetrain.version>Moore-SR6</spring-data-releasetrain.version>
</properties>
```

Browse the `Dependency versions Appendix` in the Spring Boot reference for a complete list of dependency version properties.

3.2. Using Spring Boot without the Parent POM

There may be reasons for you not to inherit from the `spring-boot-starter-parent` POM. You may have your own corporate standard parent that you need to use or you may prefer to explicitly declare all your Maven configuration.

If you do not want to use the `spring-boot-starter-parent`, you can still keep the benefit of the dependency management (but not the plugin management) by using an `import` scoped dependency, as follows:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <!-- Import dependency management from Spring Boot -->
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>2.4.7</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

The preceding sample setup does not let you override individual dependencies by using properties, as explained above. To achieve the same result, you need to add entries in the `dependencyManagement` section of your project **before** the `spring-boot-dependencies` entry. For instance, to use a different version of the SLF4J library and the Spring Data release train, you could add the following elements to your `pom.xml`:

```
<dependencyManagement>
    <dependencies>
        <!-- Override SLF4J provided by Spring Boot -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.30</version>
        </dependency>
        <!-- Override Spring Data release train provided by Spring Boot -->
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-releasetrain</artifactId>
            <version>2020.0.0-SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>2.4.7</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

3.3. Overriding settings on the command-line

The plugin offers a number of user properties, starting with `spring-boot`, to let you customize the configuration from the command-line.

For instance, you could tune the profiles to enable when running the application as follows:

```
$ mvn spring-boot:run -Dspring-boot.run.profiles=dev,local
```

If you want to both have a default while allowing it to be overridden on the command-line, you should use a combination of a user-provided project property and MOJO configuration.

```
<project>
    <properties>
        <app.profiles>local,dev</app.profiles>
    </properties>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <profiles>${app.profiles}</profiles>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

The above makes sure that `local` and `dev` are enabled by default. Now a dedicated property has been exposed, this can be overridden on the command-line as well:

```
$ mvn spring-boot:run -Dapp.profiles=test
```

Chapter 4. Goals

The Spring Boot Plugin has the following goals:

Goal	Description
spring-boot:build-image	Package an application into a OCI image using a buildpack.
spring-boot:build-info	Generate a `build-info.properties` file based on the content of the current `MavenProject`.
spring-boot:help	Display help information on spring-boot-maven-plugin. Call `mvn spring-boot:help -Ddetail=true -Dgoal=<goal-name>` to display parameter details.
spring-boot:repackage	Repackage existing JAR and WAR archives so that they can be executed from the command line using `java -jar`. With `layout=NONE` can also be used simply to package a JAR with nested dependencies (and no main class, so not executable).
spring-boot:run	Run an application in place.
spring-boot:start	Start a spring application. Contrary to the `run` goal, this does not block and allows other goals to operate on the application. This goal is typically used in integration test scenario where the application is started before a test suite and stopped after.
spring-boot:stop	Stop an application that has been started by the "start" goal. Typically invoked once a test suite has completed.

Chapter 5. 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`.

Packaging an executable archive is performed by the `repackage` goal, as shown in the following example:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <executions>
                <execution>
                    <goals>
                        <goal>repackage</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

If you are using `spring-boot-starter-parent`, such execution is already pre-configured with a `repackage` execution ID so that only the plugin definition should be added.

The example above repackages a `jar` or `war` archive that is built during the package phase of the Maven lifecycle, including any `provided` dependencies that are defined in the project. If some of these dependencies need to be excluded, you can use one of the `exclude` options; see the dependency exclusion for more details.

The original (i.e. non-executable) artifact is renamed to `.original` by default but it is also possible to keep the original artifact using a custom classifier.

The `outputFileNameMapping` feature of the `maven-war-plugin` is currently not supported.

Devtools is automatically excluded by default (you can control that using the `excludeDevtools` property). In order to make that work with `war` packaging, the `spring-boot-devtools` dependency must be set as `optional` or with the `provided` scope.

The plugin rewrites your manifest, and in particular it manages the `Main-Class` and `Start-Class` entries. If the defaults don't work you have to configure the values in the Spring Boot plugin, not in the jar plugin. The `Main-Class` in the manifest is controlled by the `layout` property of the Spring Boot plugin, as shown in the following example:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <mainClass>${start.class}</mainClass>
                <layout>ZIP</layout>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>repackage</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

The `layout` property defaults to a value determined by the archive type (`jar` or `war`). The following layouts are available:

- `JAR`: regular executable JAR layout.
- `WAR`: executable WAR layout. `provided` dependencies are placed in `WEB-INF/lib-provided` to avoid any clash when the `war` is deployed in a servlet container.
- `ZIP` (alias to `DIR`): similar to the `JAR` layout using `PropertiesLauncher`.
- `NONE`: Bundle all dependencies and project resources. Does not bundle a bootstrap loader.

5.1. Layered Jars

A repackaged jar contains the application's classes and dependencies in `BOOT-INF/classes` and `BOOT-INF/lib` respectively. For cases where a docker image needs to be built from the contents of the jar, it's useful to be able to separate these directories further so that they can be written into distinct layers.

Layered jars use the same layout as regular repackaged jars, but include an additional meta-data file that describes each layer.

By default, the following layers are defined:

- `dependencies` for any dependency whose version does not contain `SNAPSHOT`.
- `spring-boot-loader` for the jar loader classes.
- `snapshot-dependencies` for any dependency whose version contains `SNAPSHOT`.
- `application` for local module dependencies, application classes, and resources.

Module dependencies are identified by looking at all of the modules that are part of the current build. If a module dependency can only be resolved because it has been installed into Maven's local cache and it is not part of the current build, it will be identified as regular dependency.

The layers order is important as it determines how likely previous layers can be cached when part of the application changes. The default order is `dependencies`, `spring-boot-loader`, `snapshot-dependencies`, `application`. Content that is least likely to change should be added first, followed by layers that are more likely to change.

The repackaged jar includes the `layers.idx` file by default. To disable this feature, you can do so in the following manner:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <layers>
                        <enabled>false</enabled>
                    </layers>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

5.1.1. Custom Layers Configuration

Depending on your application, you may want to tune how layers are created and add new ones. This can be done using a separate configuration file that should be registered as shown below:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <layers>
                        <enabled>true</enabled>

<configuration>${project.basedir}/src/layers.xml</configuration>
                    </layers>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

The configuration file describes how the jar can be separated into layers, and the order of those layers. The following example shows how the default ordering described above can be defined explicitly:

```
<layers xmlns="http://www.springframework.org/schema/boot/layers"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/boot/layers
                      https://www.springframework.org/schema/boot/layers/layers-
2.4.xsd">
    <application>
        <into layer="spring-boot-loader">
            <include>org/springframework/boot/loader/**</include>
        </into>
        <into layer="application" />
    </application>
    <dependencies>
        <into layer="application">
            <includeModuleDependencies />
        </into>
        <into layer="snapshot-dependencies">
            <include>*:*:*SNAPSHOT</include>
        </into>
        <into layer="dependencies" />
    </dependencies>
    <layerOrder>
        <layer>dependencies</layer>
        <layer>spring-boot-loader</layer>
        <layer>snapshot-dependencies</layer>
        <layer>application</layer>
    </layerOrder>
</layers>
```

The `layers` XML format is defined in three sections:

- The `<application>` block defines how the application classes and resources should be layered.
- The `<dependencies>` block defines how dependencies should be layered.
- The `<layerOrder>` block defines the order that the layers should be written.

Nested `<into>` blocks are used within `<application>` and `<dependencies>` sections to claim content for a layer. The blocks are evaluated in the order that they are defined, from top to bottom. Any content not claimed by an earlier block remains available for subsequent blocks to consider.

The `<into>` block claims content using nested `<include>` and `<exclude>` elements. The `<application>` section uses Ant-style patch matching for include/exclude expressions. The `<dependencies>` section uses `group:artifact[:version]` patterns. It also provides `<includeModuleDependencies />` and `<excludeModuleDependencies />` elements that can be used to include or exclude local module dependencies.

If no `<include>` is defined, then all content (not claimed by an earlier block) is considered.

If no `<exclude>` is defined, then no exclusions are applied.

Looking at the `<dependencies>` example above, we can see that the first `<into>` will claim all module dependencies for the `application.layer`. The next `<into>` will claim all SNAPSHOT dependencies for the `snapshot-dependencies` layer. The final `<into>` will claim anything left (in this case, any dependency that is not a SNAPSHOT) for the `dependencies` layer.

The `<application>` block has similar rules. First claiming `org/springframework/boot/loader/**` content for the `spring-boot-loader` layer. Then claiming any remaining classes and resources for the `application` layer.

The order that `<into>` blocks are defined is often different from the order that the layers are written. For this reason the `<layerOrder>` element must always be included and cover all layers referenced by the `<into>` blocks.

5.2. `spring-boot:repackage`

`org.springframework.boot:spring-boot-maven-plugin:2.4.7`

Repackage existing JAR and WAR archives so that they can be executed from the command line using `java -jar`. With `layout=NONE` can also be used simply to package a JAR with nested dependencies (and no main class, so not executable).

5.2.1. Required parameters

Name	Type	Default
outputDirectory	`File`	`${project.build.directory}`

5.2.2. Optional parameters

Name	Type	Default
attach	`boolean`	`true`
classifier	`String`	
embeddedLaunchScript	`File`	
embeddedLaunchScriptProperties	`Properties`	
excludeDevtools	`boolean`	`true`
excludeGroupIds	`String`	
excludes	`List`	
executable	`boolean`	`false`
includeSystemScope	`boolean`	`false`
includes	`List`	
layers	`Layers`	
layout	`LayoutType`	
layoutFactory	`LayoutFactory`	

Name	Type	Default
mainClass	String	
outputTimestamp	String	${project.build.outputTimestamp}
requiresUnpack	List	
skip	boolean	false

5.2.3. Parameter details

`attach`

Attach the repackaged archive to be installed into your local Maven repository or deployed to a remote repository. If no classifier has been configured, it will replace the normal jar. If a `classifier` has been configured such that the normal jar and the repackaged jar are different, it will be attached alongside the normal jar. When the property is set to `false`, the repackaged archive will not be installed or deployed.

Name	attach
Type	boolean
Default value	true
User property	
Since	1.4.0

`classifier`

Classifier to add to the repackaged archive. If not given, the main artifact will be replaced by the repackaged archive. If given, the classifier will also be used to determine the source archive to repackage: if an artifact with that classifier already exists, it will be used as source and replaced. If no such artifact exists, the main artifact will be used as source and the repackaged archive will be attached as a supplemental artifact with that classifier. Attaching the artifact allows to deploy it alongside to the original one, see $1[$2].

Name	classifier
Type	java.lang.String
Default value	
User property	
Since	1.0.0

embeddedLaunchScript

The embedded launch script to prepend to the front of the jar if it is fully executable. If not specified the 'Spring Boot' default script will be used.

Name	embeddedLaunchScript
Type	java.io.File
Default value	
User property	
Since	1.3.0

embeddedLaunchScriptProperties

Properties that should be expanded in the embedded launch script.

Name	embeddedLaunchScriptProperties
Type	java.util.Properties
Default value	
User property	
Since	1.3.0

excludeDevtools

Exclude Spring Boot devtools from the repackaged archive.

Name	excludeDevtools
Type	boolean
Default value	true
User property	spring-boot.repackage.excludeDevtools
Since	1.3.0

excludeGroupIds

Comma separated list of groupId names to exclude (exact match).

Name	excludeGroupIds

Type	`java.lang.String`
Default value	
User property	`spring-boot.excludeGroupIds`
Since	`1.1.0`

`excludes`

Collection of artifact definitions to exclude. The `Exclude` element defines mandatory `groupId` and `artifactId` properties and an optional `classifier` property.

Name	`excludes`
Type	`java.util.List`
Default value	
User property	`spring-boot.excludes`
Since	`1.1.0`

`executable`

Make a fully executable jar for *nix machines by prepending a launch script to the jar. <p> Currently, some tools do not accept this format so you may not always be able to use this technique. For example, `jar -xf` may silently fail to extract a jar or war that has been made fully-executable. It is recommended that you only enable this option if you intend to execute it directly, rather than running it with `java -jar` or deploying it to a servlet container.

Name	`executable`
Type	`boolean`
Default value	`false`
User property	
Since	`1.3.0`

`includeSystemScope`

Include system scoped dependencies.

Name	`includeSystemScope`

Type	`boolean`
Default value	`false`
User property	
Since	`1.4.0`

`includes`

Collection of artifact definitions to include. The `Include` element defines mandatory `groupId` and `artifactId` properties and an optional mandatory `groupId` and `artifactId` properties and an optional `classifier` property.

Name	`includes`
Type	`java.util.List`
Default value	
User property	`spring-boot.includes`
Since	`1.2.0`

`layers`

Layer configuration with options to disable layer creation, exclude layer tools jar, and provide a custom layers configuration file.

Name	`layers`
Type	`org.springframework.boot.maven.Layers`
Default value	
User property	
Since	`2.3.0`

`layout`

The type of archive (which corresponds to how the dependencies are laid out inside it). Possible values are `JAR`, `WAR`, `ZIP`, `DIR`, `NONE`. Defaults to a guess based on the archive type.

Name	`layout`
Type	`org.springframework.boot.maven.AbstractPackagerMojo$LayoutType`

Default value	
User property	`spring-boot.repackage.layout`
Since	`1.0.0`

`layoutFactory`

The layout factory that will be used to create the executable archive if no explicit layout is set. Alternative layouts implementations can be provided by 3rd parties.

Name	`layoutFactory`
Type	`org.springframework.boot.loader.tools.LayoutFactory`
Default value	
User property	
Since	`1.5.0`

`mainClass`

The name of the main class. If not specified the first compiled class found that contains a `main` method will be used.

Name	`mainClass`
Type	`java.lang.String`
Default value	
User property	
Since	`1.0.0`

`outputDirectory`

Directory containing the generated archive.

Name	`outputDirectory`
Type	`java.io.File`
Default value	`${project.build.directory}`

User property	
Since	1.0.0

`outputTimestamp`

Timestamp for reproducible output archive entries, either formatted as ISO 8601 (`yyyy-MM-dd'T'HH:mm:ssXXX`) or an `int` representing seconds since the epoch. Not supported with war packaging.

Name	`outputTimestamp`
Type	`java.lang.String`
Default value	`${project.build.outputTimestamp}`
User property	
Since	2.3.0

`requiresUnpack`

A list of the libraries that must be unpacked from fat jars in order to run. Specify each library as a `<dependency>` with a `<groupId>` and a `<artifactId>` and they will be unpacked at runtime.

Name	`requiresUnpack`
Type	`java.util.List`
Default value	
User property	
Since	1.1.0

`skip`

Skip the execution.

Name	`skip`
Type	`boolean`
Default value	`false`

User property	`spring-boot.repackage.skip`
Since	`1.2.0`

5.3. Examples

5.3.1. Custom Classifier

By default, the `repackage` goal replaces the original artifact with the repackaged one. That is a sane behavior for modules that represent an application but if your module is used as a dependency of another module, you need to provide a classifier for the repackaged one. The reason for that is that application classes are packaged in `BOOT-INF/classes` so that the dependent module cannot load a repackaged jar's classes.

If that is the case or if you prefer to keep the original artifact and attach the repackaged one with a different classifier, configure the plugin as shown in the following example:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                        <configuration>
                            <classifier>exec</classifier>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

If you are using `spring-boot-starter-parent`, the `repackage` goal is executed automatically in an execution with id `repackage`. In that setup, only the configuration should be specified, as shown in the following example:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <configuration>
                            <classifier>exec</classifier>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

This configuration will generate two artifacts: the original one and the repackaged counter part produced by the repackage goal. Both will be installed/deployed transparently.

You can also use the same configuration if you want to repackage a secondary artifact the same way the main artifact is replaced. The following configuration installs/deploys a single `task` classified artifact with the repackaged application:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <executions>
                    <execution>
                        <goals>
                            <goal>jar</goal>
                        </goals>
                        <phase>package</phase>
                        <configuration>
                            <classifier>task</classifier>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                        <configuration>
                            <classifier>task</classifier>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

As both the `maven-jar-plugin` and the `spring-boot-maven-plugin` runs at the same phase, it is important that the jar plugin is defined first (so that it runs before the repackage goal). Again, if you are using `spring-boot-starter-parent`, this can be simplified as follows:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <executions>
                    <execution>
                        <id>default-jar</id>
                        <configuration>
                            <classifier>task</classifier>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <configuration>
                            <classifier>task</classifier>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

5.3.2. Custom Name

If you need the repackaged jar to have a different local name than the one defined by the `artifactId` attribute of the project, use the standard `finalName`, as shown in the following example:

```
<project>
    <build>
        <finalName>my-app</finalName>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

This configuration will generate the repackaged artifact in `target/my-app.jar`.

5.3.3. Local Repackaged Artifact

By default, the `repackage` goal replaces the original artifact with the executable one. If you need to only deploy the original jar and yet be able to run your app with the regular file name, configure the plugin as follows:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                        <configuration>
                            <attach>false</attach>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

This configuration generates two artifacts: the original one and the executable counter part produced by the `repackage` goal. Only the original one will be installed/deployed.

5.3.4. Custom Layout

Spring Boot repackages the jar file for this project using a custom layout factory defined in the additional jar file, provided as a dependency to the build plugin:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                        <configuration>
                            <layoutFactory
implementation="com.example.CustomLayoutFactory">
                                <customProperty>value</customProperty>
                            </layoutFactory>
                        </configuration>
                    </execution>
                </executions>
                <dependencies>
                    <dependency>
                        <groupId>com.example</groupId>
                        <artifactId>custom-layout</artifactId>
                        <version>0.0.1.BUILD-SNAPSHOT</version>
                    </dependency>
                </dependencies>
            </plugin>
        </plugins>
    </build>
</project>
```

The layout factory is provided as an implementation of `LayoutFactory` (from `spring-boot-loader-tools`) explicitly specified in the pom. If there is only one custom `LayoutFactory` on the plugin classpath and it is listed in `META-INF/spring.factories` then it is unnecessary to explicitly set it in the plugin configuration.

Layout factories are always ignored if an explicit layout is set.

5.3.5. Dependency Exclusion

By default, both the `repackage` and the `run` goals will include any `provided` dependencies that are defined in the project. A Spring Boot project should consider `provided` dependencies as "container" dependencies that are required to run the application.

Some of these dependencies may not be required at all and should be excluded from the executable jar. For consistency, they should not be present either when running the application.

There are two ways one can exclude a dependency from being packaged/used at runtime:

- Exclude a specific artifact identified by `groupId` and `artifactId`, optionally with a `classifier` if needed.
- Exclude any artifact belonging to a given `groupId`.

The following example excludes `com.foo:bar`, and only that artifact:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>com.foo</groupId>
                            <artifactId>bar</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

This example excludes any artifact belonging to the `com.foo` group:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludeGroupIds>com.foo</excludeGroupIds>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

5.3.6. Layered Jar Tools

When a layered jar is created, the `spring-boot-jarmode-layertools` jar will be added as a dependency to your jar. With this jar on the classpath, you can launch your application in a special mode which allows the bootstrap code to run something entirely different from your application, for example, something that extracts the layers. If you wish to exclude this dependency, you can do so in the following manner:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <layers>
                        <includeLayerTools>false</includeLayerTools>
                    </layers>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

5.3.7. Custom Layers Configuration

The default setup splits dependencies into snapshot and non-snapshot, however, you may have more complex rules. For example, you may want to isolate company-specific dependencies of your project in a dedicated layer. The following `layers.xml` configuration shown one such setup:

```
<layers xmlns="http://www.springframework.org/schema/boot/layers"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/boot/layers
                      https://www.springframework.org/schema/boot/layers/layers-
2.4.xsd">
    <application>
        <into layer="spring-boot-loader">
            <include>org/springframework/boot/loader/**</include>
        </into>
        <into layer="application" />
    </application>
    <dependencies>
        <into layer="snapshot-dependencies">
            <include>*:*:*SNAPSHOT</include>
        </into>
        <into layer="company-dependencies">
            <include>com.acme:*</include>
        </into>
        <into layer="dependencies"/>
    </dependencies>
    <layerOrder>
        <layer>dependencies</layer>
        <layer>spring-boot-loader</layer>
        <layer>snapshot-dependencies</layer>
        <layer>company-dependencies</layer>
        <layer>application</layer>
    </layerOrder>
</layers>
```

The configuration above creates an additional `company-dependencies` layer with all libraries with the `com.acme` groupId.

Chapter 6. Packaging OCI Images

The plugin can create an OCI image from an executable jar file using Cloud Native Buildpacks (CNB). Images can be built using the `build-image` goal.

For security reasons, images build and run as non-root users. See the CNB specification for more details.

The `build-image` goal is not supported with projects using war packaging.

The easiest way to get started is to invoke `mvn spring-boot:build-image` on a project. It is possible to automate the creation of an image whenever the `package` phase is invoked, as shown in the following example:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <executions>
                <execution>
                    <goals>
                        <goal>build-image</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

While the buildpack runs from an executable archive, it is not necessary to execute the `repackage` goal first as the executable archive is created automatically if necessary. When the `build-image` repackages the application, it applies the same settings as the `repackage` goal would, i.e. dependencies can be excluded using one of the exclude options, and Devtools is automatically excluded by default (you can control that using the `excludeDevtools` property).

6.1. Docker Daemon

The `build-image` goal requires access to a Docker daemon. By default, it will communicate with a Docker daemon over a local connection. This works with Docker Engine on all supported platforms without configuration.

Environment variables can be set to configure the `build-image` goal to use the Docker daemon provided by minikube. The following table shows the environment variables and their values:

Environment variable	Description
DOCKER_HOST	URL containing the host and port for the Docker daemon - e.g. `tcp://192.168.99.100:2376`
DOCKER_TLS_VERIFY	Enable secure HTTPS protocol when set to `1` (optional)
DOCKER_CERT_PATH	Path to certificate and key files for HTTPS (required if `DOCKER_TLS_VERIFY=1`, ignored otherwise)

On Linux and macOS, these environment variables can be set using the command `eval $(minikube docker-env)` after minikube has been started.

Docker daemon connection information can also be provided using `docker` parameters in the plugin configuration. The following table summarizes the available parameters:

Parameter	Description
`host`	URL containing the host and port for the Docker daemon - e.g. `tcp://192.168.99.100:2376`
`tlsVerify`	Enable secure HTTPS protocol when set to `true` (optional)
`certPath`	Path to certificate and key files for HTTPS (required if `tlsVerify` is `true`, ignored otherwise)

For more details, see also examples.

6.2. Docker Registry

If the Docker images specified by the `builder` or `runImage` parameters are stored in a private Docker image registry that requires authentication, the authentication credentials can be provided using `docker.builderRegistry` parameters.

If the generated Docker image is to be published to a Docker image registry, the authentication credentials can be provided using `docker.publishRegistry` parameters.

Parameters are provided for user authentication or identity token authentication. Consult the documentation for the Docker registry being used to store images for further information on supported authentication methods.

The following table summarizes the available parameters for `docker.builderRegistry` and `docker.publishRegistry`:

Parameter	Description
`username`	Username for the Docker image registry user. Required for user authentication.

Parameter	Description
password	Password for the Docker image registry user. Required for user authentication.
url	Address of the Docker image registry. Optional for user authentication.
email	E-mail address for the Docker image registry user. Optional for user authentication.
token	Identity token for the Docker image registry user. Required for token authentication.

For more details, see also examples.

6.3. Image Customizations

The plugin invokes a builder to orchestrate the generation of an image. The builder includes multiple buildpacks that can inspect the application to influence the generated image. By default, the plugin chooses a builder image. The name of the generated image is deduced from project properties.

The image parameter allows configuration of the builder and how it should operate on the project. The following table summarizes the available parameters and their default values:

Parameter	Description	User property	Default value
builder	Name of the Builder image to use.	spring-boot.build-image.builder	paketobuildpacks/builder:base
runImage	Name of the run image to use.	spring-boot.build-image.runImage	No default value, indicating the run image specified in Builder metadata should be used.
name	Image name for the generated image.	spring-boot.build-image.imageName	docker.io/library/${project.artifactId}:${project.version}
pullPolicy	Policy used to determine when to pull the builder and run images from the registry. Acceptable values are ALWAYS, NEVER, and IF_NOT_PRESENT.	spring-boot.build-image.pullPolicy	ALWAYS
env	Environment variables that should be passed to the builder.		

Parameter	Description	User property	Default value
`cleanCache`	Whether to clean the cache before building.	`spring-boot.build-image.cleanCache`	`false`
`verboseLogging`	Enables verbose logging of builder operations.		`false`
`publish`	Whether to publish the generated image to a Docker registry.	`spring-boot.build-image.publish`	`false`

The plugin detects the target Java compatibility of the project using the compiler's plugin configuration or the `maven.compiler.target` property. When using the default Paketo builder and buildpacks, the plugin instructs the buildpacks to install the same Java version. You can override this behaviour as shown in the builder configuration examples.

For more details, see also examples.

6.4. `spring-boot:build-image`

`org.springframework.boot:spring-boot-maven-plugin:2.4.7`

Package an application into a OCI image using a buildpack.

6.4.1. Required parameters

Name	Type	Default
sourceDirectory	`File`	`${project.build.directory}`

6.4.2. Optional parameters

Name	Type	Default
classifier	`String`	
docker	`Docker`	
excludeDevtools	`boolean`	`true`
excludeGroupIds	`String`	
excludes	`List`	
image	`Image`	
includeSystemScope	`boolean`	`false`
includes	`List`	
layers	`Layers`	

Name	Type	Default
layout	LayoutType	
layoutFactory	LayoutFactory	
mainClass	String	
skip	boolean	false

6.4.3. Parameter details

`classifier`

Classifier used when finding the source archive.

Name	classifier
Type	java.lang.String
Default value	
User property	
Since	2.3.0

`docker`

Docker configuration options.

Name	docker
Type	org.springframework.boot.maven.Docker
Default value	
User property	
Since	2.4.0

`excludeDevtools`

Exclude Spring Boot devtools from the repackaged archive.

Name	excludeDevtools
Type	boolean
Default value	true

User property	spring-boot.repackage.excludeDevtools
Since	1.3.0

excludeGroupIds

Comma separated list of groupId names to exclude (exact match).

Name	excludeGroupIds
Type	java.lang.String
Default value	
User property	spring-boot.excludeGroupIds
Since	1.1.0

excludes

Collection of artifact definitions to exclude. The `Exclude` element defines mandatory `groupId` and `artifactId` properties and an optional `classifier` property.

Name	excludes
Type	java.util.List
Default value	
User property	spring-boot.excludes
Since	1.1.0

image

Image configuration, with `builder`, `runImage`, `name`, `env`, `cleanCache`, `verboseLogging`, `pullPolicy`, and `publish` options.

Name	image
Type	org.springframework.boot.maven.Image
Default value	
User property	

Since	2.3.0

includeSystemScope

Include system scoped dependencies.

Name	includeSystemScope
Type	boolean
Default value	false
User property	
Since	1.4.0

includes

Collection of artifact definitions to include. The Include element defines mandatory groupId and artifactId properties and an optional mandatory groupId and artifactId properties and an optional classifier property.

Name	includes
Type	java.util.List
Default value	
User property	spring-boot.includes
Since	1.2.0

layers

Layer configuration with options to disable layer creation, exclude layer tools jar, and provide a custom layers configuration file.

Name	layers
Type	org.springframework.boot.maven.Layers
Default value	
User property	
Since	2.3.0

`layout`

The type of archive (which corresponds to how the dependencies are laid out inside it). Possible values are `JAR`, `WAR`, `ZIP`, `DIR`, `NONE`. Defaults to a guess based on the archive type.

Name	`layout`
Type	`org.springframework.boot.maven.AbstractPackagerMojo$LayoutType`
Default value	
User property	
Since	`2.3.11`

`layoutFactory`

The layout factory that will be used to create the executable archive if no explicit layout is set. Alternative layouts implementations can be provided by 3rd parties.

Name	`layoutFactory`
Type	`org.springframework.boot.loader.tools.LayoutFactory`
Default value	
User property	
Since	`2.3.11`

`mainClass`

The name of the main class. If not specified the first compiled class found that contains a `main` method will be used.

Name	`mainClass`
Type	`java.lang.String`
Default value	
User property	
Since	`1.0.0`

`skip`

Skip the execution.

Name	`skip`
Type	`boolean`
Default value	`false`
User property	`spring-boot.build-image.skip`
Since	`2.3.0`

`sourceDirectory`

Directory containing the source archive.

Name	`sourceDirectory`
Type	`java.io.File`
Default value	`${project.build.directory}`
User property	
Since	`2.3.0`

6.5. Examples

6.5.1. Custom Image Builder

If you need to customize the builder used to create the image or the run image used to launch the built image, configure the plugin as shown in the following example:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <image>
                        <builder>mine/java-cnb-builder</builder>
                        <runImage>mine/java-cnb-run</runImage>
                    </image>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

This configuration will use a builder image with the name `mine/java-cnb-builder` and the tag `latest`, and the run image named `mine/java-cnb-run` and the tag `latest`.

The builder and run image can be specified on the command line as well, as shown in this example:

```
$ mvn spring-boot:build-image -Dspring-boot.build-image.builder=mine/java-cnb-builder
-Dspring-boot.build-image.runImage=mine/java-cnb-run
```

6.5.2. Builder Configuration

If the builder exposes configuration options using environment variables, those can be set using the `env` attributes.

The following is an example of configuring the JVM version used by the Paketo Java buildpacks at build time:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <image>
                        <env>
                            <BP_JVM_VERSION>8.*</BP_JVM_VERSION>
                        </env>
                    </image>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

If there is a network proxy between the Docker daemon the builder runs in and network locations that buildpacks download artifacts from, you will need to configure the builder to use the proxy. When using the Paketo builder, this can be accomplished by setting the `HTTPS_PROXY` and/or `HTTP_PROXY` environment variables as show in the following example:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <image>
                        <env>
                            <HTTP_PROXY>http://proxy.example.com</HTTP_PROXY>
                            <HTTPS_PROXY>https://proxy.example.com</HTTPS_PROXY>
                        </env>
                    </image>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

6.5.3. Runtime JVM Configuration

Paketo Java buildpacks configure the JVM runtime environment by setting the `JAVA_TOOL_OPTIONS` environment variable. The buildpack-provided `JAVA_TOOL_OPTIONS` value can be modified to customize JVM runtime behavior when the application image is launched in a container.

Environment variable modifications that should be stored in the image and applied to every deployment can be set as described in the Paketo documentation and shown in the following example:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <image>
                        <env>
                            <BPE_DELIM_JAVA_TOOL_OPTIONS xml:space="preserve">
</BPE_DELIM_JAVA_TOOL_OPTIONS>
                            <BPE_APPEND_JAVA_TOOL_OPTIONS>-
XX:+HeapDumpOnOutOfMemoryError</BPE_APPEND_JAVA_TOOL_OPTIONS>
                        </env>
                    </image>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

6.5.4. Custom Image Name

By default, the image name is inferred from the `artifactId` and the `version` of the project, something like `docker.io/library/${project.artifactId}:${project.version}`. You can take control over the name, as shown in the following example:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <image>
                        <name>example.com/library/${project.artifactId}</name>
                    </image>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

This configuration does not provide an explicit tag so `latest` is used. It is possible to specify a tag as well, either using `${project.version}`, any property available in the build or a hardcoded version.

The image name can be specified on the command line as well, as shown in this example:

```
$ mvn spring-boot:build-image -Dspring-boot.build
-image.imageName=example.com/library/my-app:v1
```

6.5.5. Image Publishing

The generated image can be published to a Docker registry by enabling a `publish` option and configuring authentication for the registry using `docker.publishRegistry` parameters.

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <image>
                        <name>docker.example.com/library/${project.artifactId}</name>
                        <publish>true</publish>
                    </image>
                    <docker>
                        <publishRegistry>
                            <username>user</username>
                            <password>secret</password>
                            <url>https://docker.example.com/v1/</url>
                            <email>user@example.com</email>
                        </publishRegistry>
                    </docker>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

The `publish` option can be specified on the command line as well, as shown in this example:

```
$ mvn spring-boot:build-image -Dspring-boot.build
-image.imageName=docker.example.com/library/my-app:v1 -Dspring-boot.build
-image.publish=true
```

6.5.6. Docker Configuration

If you need the plugin to communicate with the Docker daemon using a remote connection instead of the default local connection, the connection details can be provided using `docker` parameters as shown in the following example:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <docker>
                        <host>tcp://192.168.99.100:2376</host>
                        <tlsVerify>true</tlsVerify>
                        <certPath>/home/user/.minikube/certs</certPath>
                    </docker>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

If the builder or run image are stored in a private Docker registry that supports user authentication, authentication details can be provided using `docker.builderRegistry` parameters as shown in the following example:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <docker>
                        <builderRegistry>
                            <username>user</username>
                            <password>secret</password>
                            <url>https://docker.example.com/v1/</url>
                            <email>user@example.com</email>
                        </builderRegistry>
                    </docker>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

If the builder or run image is stored in a private Docker registry that supports token authentication, the token value can be provided using `docker.builderRegistry` parameters as shown in the following example:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <docker>
                        <builderRegistry>
                            <token>9cbaf023786cd7...</token>
                        </builderRegistry>
                    </docker>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Chapter 7. Running your Application with Maven

The plugin includes a run goal which can be used to launch your application from the command line, as shown in the following example:

```
$ mvn spring-boot:run
```

Application arguments can be specified using the `arguments` parameter, see using application arguments for more details.

By default the application is executed in a forked process and setting properties on the command-line will not affect the application. If you need to specify some JVM arguments (i.e. for debugging purposes), you can use the `jvmArguments` parameter, see Debug the application for more details. There is also explicit support for system properties and environment variables.

As enabling a profile is quite common, there is dedicated `profiles` property that offers a shortcut for `-Dspring-boot.run.jvmArguments="-Dspring.profiles.active=dev"`, see Specify active profiles.

Although this is not recommended, it is possible to execute the application directly from the Maven JVM by disabling the `fork` property. Doing so means that the `jvmArguments`, `systemPropertyVariables`, `environmentVariables` and `agents` options are ignored.

Spring Boot `devtools` is a module to improve the development-time experience when working on Spring Boot applications. To enable it, just add the following dependency to your project:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

When `devtools` is running, it detects change when you recompile your application and automatically refreshes it. This works for not only resources but code as well. It also provides a LiveReload server so that it can automatically trigger a browser refresh whenever things change.

Devtools can also be configured to only refresh the browser whenever a static resource has changed (and ignore any change in the code). Just include the following property in your project:

```
spring.devtools.remote.restart.enabled=false
```

Prior to `devtools`, the plugin supported hot refreshing of resources by default which has now be disabled in favour of the solution described above. You can restore it at any time by configuring

your project:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <addResources>true</addResources>
            </configuration>
        </plugin>
    </plugins>
</build>
```

When `addResources` is enabled, any `src/main/resources` directory will be added to the application classpath when you run the application and any duplicate found in `target/classes` will be removed. This allows hot refreshing of resources which can be very useful when developing web applications. For example, you can work on HTML, CSS or JavaScript files and see your changes immediately without recompiling your application. It is also a helpful way of allowing your front end developers to work without needing to download and install a Java IDE.

A side effect of using this feature is that filtering of resources at build time will not work.

In order to be consistent with the `repackage` goal, the `run` goal builds the classpath in such a way that any dependency that is excluded in the plugin's configuration gets excluded from the classpath as well. For more details, see the dedicated example.

Sometimes it is useful to include test dependencies when running the application. For example, if you want to run your application in a test mode that uses stub classes. If you wish to do this, you can set the `useTestClasspath` parameter to true.

This is only applied when you run an application: the `repackage` goal will not add test dependencies to the resulting JAR/WAR.

7.1. `spring-boot:run`

`org.springframework.boot:spring-boot-maven-plugin:2.4.7`

Run an application in place.

7.1.1. Required parameters

Name	Type	Default
classesDirectory	`File`	`${project.build.outputDirectory}`

7.1.2. Optional parameters

Name	Type	Default
addResources	`boolean`	`false`
agents	`File[]`	
arguments	`String[]`	
commandlineArguments	`String`	
directories	`String[]`	
environmentVariables	`Map`	
excludeGroupIds	`String`	
excludes	`List`	
folders	`String[]`	
fork	`boolean`	`true`
includes	`List`	
jvmArguments	`String`	
mainClass	`String`	
noverify	`boolean`	
optimizedLaunch	`boolean`	`true`
profiles	`String[]`	
skip	`boolean`	`false`
systemPropertyVariables	`Map`	
useTestClasspath	`Boolean`	`false`
workingDirectory	`File`	

7.1.3. Parameter details

`addResources`

Add maven resources to the classpath directly, this allows live in-place editing of resources. Duplicate resources are removed from `target/classes` to prevent them to appear twice if `ClassLoader.getResources()` is called. Please consider adding `spring-boot-devtools` to your project instead as it provides this feature and many more.

Name	`addResources`
Type	`boolean`
Default value	`false`

User property	`spring-boot.run.addResources`
Since	`1.0.0`

`agents`

Path to agent jars. NOTE: a forked process is required to use this feature.

Name	`agents`
Type	`java.io.File[]`
Default value	
User property	`spring-boot.run.agents`
Since	`2.2.0`

`arguments`

Arguments that should be passed to the application.

Name	`arguments`
Type	`java.lang.String[]`
Default value	
User property	
Since	`1.0.0`

`classesDirectory`

Directory containing the classes and resource files that should be packaged into the archive.

Name	`classesDirectory`
Type	`java.io.File`
Default value	`${project.build.outputDirectory}`
User property	
Since	`1.0.0`

commandlineArguments

Arguments from the command line that should be passed to the application. Use spaces to separate multiple arguments and make sure to wrap multiple values between quotes. When specified, takes precedence over #arguments.

Name	commandlineArguments
Type	java.lang.String
Default value	
User property	spring-boot.run.arguments
Since	2.2.3

directories

Additional directories besides the classes directory that should be added to the classpath.

Name	directories
Type	java.lang.String[]
Default value	
User property	spring-boot.run.directories
Since	1.0.0

environmentVariables

List of Environment variables that should be associated with the forked process used to run the application. NOTE: a forked process is required to use this feature.

Name	environmentVariables
Type	java.util.Map
Default value	
User property	
Since	2.1.0

excludeGroupIds

Comma separated list of groupId names to exclude (exact match).

Name	excludeGroupIds
Type	java.lang.String
Default value	
User property	spring-boot.excludeGroupIds
Since	1.1.0

excludes

Collection of artifact definitions to exclude. The `Exclude` element defines mandatory `groupId` and `artifactId` properties and an optional `classifier` property.

Name	excludes
Type	java.util.List
Default value	
User property	spring-boot.excludes
Since	1.1.0

folders

Additional directories besides the classes directory that should be added to the classpath.

Name	folders
Type	java.lang.String[]
Default value	
User property	spring-boot.run.folders
Since	1.0.0

fork

Flag to indicate if the run processes should be forked. Disabling forking will disable some features such as an agent, custom JVM arguments, devtools or specifying the working directory to use.

Name	fork
Type	boolean

Default value	`true`
User property	`spring-boot.run.fork`
Since	`1.2.0`

`includes`

Collection of artifact definitions to include. The `Include` element defines mandatory `groupId` and `artifactId` properties and an optional mandatory `groupId` and `artifactId` properties and an optional `classifier` property.

Name	`includes`
Type	`java.util.List`
Default value	
User property	`spring-boot.includes`
Since	`1.2.0`

`jvmArguments`

JVM arguments that should be associated with the forked process used to run the application. On command line, make sure to wrap multiple values between quotes. NOTE: a forked process is required to use this feature.

Name	`jvmArguments`
Type	`java.lang.String`
Default value	
User property	`spring-boot.run.jvmArguments`
Since	`1.1.0`

`mainClass`

The name of the main class. If not specified the first compiled class found that contains a 'main' method will be used.

Name	`mainClass`
Type	`java.lang.String`

Default value	
User property	`spring-boot.run.main-class`
Since	`1.0.0`

`noverify`

Flag to say that the agent requires -noverify.

Name	`noverify`
Type	`boolean`
Default value	
User property	`spring-boot.run.noverify`
Since	`1.0.0`

`optimizedLaunch`

Whether the JVM's launch should be optimized.

Name	`optimizedLaunch`
Type	`boolean`
Default value	`true`
User property	`spring-boot.run.optimizedLaunch`
Since	`2.2.0`

`profiles`

The spring profiles to activate. Convenience shortcut of specifying the 'spring.profiles.active' argument. On command line use commas to separate multiple profiles.

Name	`profiles`
Type	`java.lang.String[]`
Default value	

User property	`spring-boot.run.profiles`
Since	`1.3.0`

`skip`

Skip the execution.

Name	`skip`
Type	`boolean`
Default value	`false`
User property	`spring-boot.run.skip`
Since	`1.3.2`

`systemPropertyVariables`

List of JVM system properties to pass to the process. NOTE: a forked process is required to use this feature.

Name	`systemPropertyVariables`
Type	`java.util.Map`
Default value	
User property	
Since	`2.1.0`

`useTestClasspath`

Flag to include the test classpath when running.

Name	`useTestClasspath`
Type	`java.lang.Boolean`
Default value	`false`
User property	`spring-boot.run.useTestClasspath`
Since	`1.3.0`

`workingDirectory`

Current working directory to use for the application. If not specified, basedir will be used. NOTE: a forked process is required to use this feature.

Name	`workingDirectory`
Type	`java.io.File`
Default value	
User property	`spring-boot.run.workingDirectory`
Since	`1.5.0`

7.2. Examples

7.2.1. Debug the Application

By default, the `run` goal runs your application in a forked process. If you need to debug it, you should add the necessary JVM arguments to enable remote debugging. The following configuration suspend the process until a debugger has joined on port 5005:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <jvmArguments>
                        -Xdebug
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005
                    </jvmArguments>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

These arguments can be specified on the command line as well, make sure to wrap that properly, that is:

```
$ mvn spring-boot:run -Dspring-boot.run.jvmArguments="-Xdebug
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005"
```

7.2.2. Using System Properties

System properties can be specified using the `systemPropertyVariables` attribute. The following example sets `property1` to `test` and `property2` to 42:

```
<project>
    <build>
        <properties>
            <my.value>42</my.value>
        </properties>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <systemPropertyVariables>
                        <property1>test</property1>
                        <property2>${my.value}</property2>
                    </systemPropertyVariables>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

If the value is empty or not defined (i.e. `<my-property/>`), the system property is set with an empty String as the value. Maven trims values specified in the pom so it is not possible to specify a System property which needs to start or end with a space via this mechanism: consider using `jvmArguments` instead.

Any String typed Maven variable can be passed as system properties. Any attempt to pass any other Maven variable type (e.g. a `List` or a `URL` variable) will cause the variable expression to be passed literally (unevaluated).

The `jvmArguments` parameter takes precedence over system properties defined with the mechanism above. In the following example, the value for `property1` is `overridden`:

```
$ mvn spring-boot:run -Dspring-boot.run.jvmArguments="-Dproperty1=overridden"
```

7.2.3. Using Environment Variables

Environment variables can be specified using the `environmentVariables` attribute. The following example sets the 'ENV1', 'ENV2', 'ENV3', 'ENV4' env variables:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <environmentVariables>
                        <ENV1>5000</ENV1>
                        <ENV2>Some Text</ENV2>
                        <ENV3/>
                        <ENV4></ENV4>
                    </environmentVariables>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

If the value is empty or not defined (i.e. `<MY_ENV/>`), the env variable is set with an empty String as the value. Maven trims values specified in the pom so it is not possible to specify an env variable which needs to start or end with a space.

Any String typed Maven variable can be passed as system properties. Any attempt to pass any other Maven variable type (e.g. a `List` or a `URL` variable) will cause the variable expression to be passed literally (unevaluated).

Environment variables defined this way take precedence over existing values.

7.2.4. Using Application Arguments

Application arguments can be specified using the `arguments` attribute. The following example sets two arguments: `property1` and `property2=42`:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <arguments>
                        <argument>property1</argument>
                        <argument>property2=${my.value}</argument>
                    </arguments>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

On the command-line, arguments are separated by a space the same way `jvmArguments` are. If an argument contains a space, make sure to quote it. In the following example, two arguments are available: `property1` and `property2=Hello World`:

```
$ mvn spring-boot:run -Dspring-boot.run.arguments="property1 'property2=Hello World'"
```

7.2.5. Specify Active Profiles

The active profiles to use for a particular application can be specified using the `profiles` argument.

The following configuration enables the `foo` and `bar` profiles:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <profiles>
                        <profile>foo</profile>
                        <profile>bar</profile>
                    </profiles>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

The profiles to enable can be specified on the command line as well, make sure to separate them

with a comma, as shown in the following example:

```
$ mvn spring-boot:run -Dspring-boot.run.profiles=foo,bar
```

Chapter 8. Running Integration Tests

While you may start your Spring Boot application very easily from your test (or test suite) itself, it may be desirable to handle that in the build itself. To make sure that the lifecycle of your Spring Boot application is properly managed around your integration tests, you can use the `start` and `stop` goals, as shown in the following example:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <executions>
                <execution>
                    <id>pre-integration-test</id>
                    <goals>
                        <goal>start</goal>
                    </goals>
                </execution>
                <execution>
                    <id>post-integration-test</id>
                    <goals>
                        <goal>stop</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Such setup can now use the failsafe-plugin to run your integration tests as you would expect.

By default, the application is started in a separate process and JMX is used to communicate with the application. If you need to configure the JMX port, see the dedicated example.

You could also configure a more advanced setup to skip the integration tests when a specific property has been set, see the dedicated example.

8.1. Using Failsafe Without Spring Boot's Parent POM

Spring Boot's Parent POM, `spring-boot-starter-parent`, configures Failsafe's `<classesDirectory>` to be `${project.build.outputDirectory}`. Without this configuration, which causes Failsafe to use the compiled classes rather than the repackaged jar, Failsafe cannot load your application's classes. If you are not using the parent POM, you should configure Failsafe in the same way, as shown in the following example:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <configuration>
        <classesDirectory>${project.build.outputDirectory}</classesDirectory>
    </configuration>
</plugin>
```

8.2. spring-boot:start

org.springframework.boot:spring-boot-maven-plugin:2.4.7

Start a spring application. Contrary to the `run` goal, this does not block and allows other goals to operate on the application. This goal is typically used in integration test scenario where the application is started before a test suite and stopped after.

8.2.1. Required parameters

Name	Type	Default
classesDirectory	File	${project.build.outputDirectory}

8.2.2. Optional parameters

Name	Type	Default
addResources	boolean	false
agents	File[]	
arguments	String[]	
commandlineArguments	String	
directories	String[]	
environmentVariables	Map	
excludeGroupIds	String	
excludes	List	
folders	String[]	
fork	boolean	true
includes	List	
jmxName	String	
jmxPort	int	
jvmArguments	String	
mainClass	String	
maxAttempts	int	

Name	Type	Default
noverify	boolean	
profiles	String[]	
skip	boolean	false
systemPropertyVariables	Map	
useTestClasspath	Boolean	false
wait	long	
workingDirectory	File	

8.2.3. Parameter details

`addResources`

Add maven resources to the classpath directly, this allows live in-place editing of resources. Duplicate resources are removed from `target/classes` to prevent them to appear twice if `ClassLoader.getResources()` is called. Please consider adding `spring-boot-devtools` to your project instead as it provides this feature and many more.

Name	addResources
Type	boolean
Default value	false
User property	spring-boot.run.addResources
Since	1.0.0

`agents`

Path to agent jars. NOTE: a forked process is required to use this feature.

Name	agents
Type	java.io.File[]
Default value	
User property	spring-boot.run.agents
Since	2.2.0

`arguments`

Arguments that should be passed to the application.

Name	`arguments`
Type	`java.lang.String[]`
Default value	
User property	
Since	`1.0.0`

`classesDirectory`

Directory containing the classes and resource files that should be packaged into the archive.

Name	`classesDirectory`
Type	`java.io.File`
Default value	`${project.build.outputDirectory}`
User property	
Since	`1.0.0`

`commandlineArguments`

Arguments from the command line that should be passed to the application. Use spaces to separate multiple arguments and make sure to wrap multiple values between quotes. When specified, takes precedence over `#arguments`.

Name	`commandlineArguments`
Type	`java.lang.String`
Default value	
User property	`spring-boot.run.arguments`
Since	`2.2.3`

`directories`

Additional directories besides the classes directory that should be added to the classpath.

Name	directories
Type	java.lang.String[]
Default value	
User property	spring-boot.run.directories
Since	1.0.0

environmentVariables

List of Environment variables that should be associated with the forked process used to run the application. NOTE: a forked process is required to use this feature.

Name	environmentVariables
Type	java.util.Map
Default value	
User property	
Since	2.1.0

excludeGroupIds

Comma separated list of groupId names to exclude (exact match).

Name	excludeGroupIds
Type	java.lang.String
Default value	
User property	spring-boot.excludeGroupIds
Since	1.1.0

excludes

Collection of artifact definitions to exclude. The `Exclude` element defines mandatory `groupId` and `artifactId` properties and an optional `classifier` property.

Name	excludes
Type	java.util.List

Default value	
User property	`spring-boot.excludes`
Since	`1.1.0`

`folders`

Additional directories besides the classes directory that should be added to the classpath.

Name	`folders`
Type	`java.lang.String[]`
Default value	
User property	`spring-boot.run.folders`
Since	`1.0.0`

`fork`

Flag to indicate if the run processes should be forked. Disabling forking will disable some features such as an agent, custom JVM arguments, devtools or specifying the working directory to use.

Name	`fork`
Type	`boolean`
Default value	`true`
User property	`spring-boot.run.fork`
Since	`1.2.0`

`includes`

Collection of artifact definitions to include. The `Include` element defines mandatory `groupId` and `artifactId` properties and an optional mandatory `groupId` and `artifactId` properties and an optional `classifier` property.

Name	`includes`
Type	`java.util.List`
Default value	

User property	`spring-boot.includes`
Since	`1.2.0`

`jmxName`

The JMX name of the automatically deployed MBean managing the lifecycle of the spring application.

Name	`jmxName`
Type	`java.lang.String`
Default value	
User property	
Since	

`jmxPort`

The port to use to expose the platform MBeanServer if the application is forked.

Name	`jmxPort`
Type	`int`
Default value	
User property	
Since	

`jvmArguments`

JVM arguments that should be associated with the forked process used to run the application. On command line, make sure to wrap multiple values between quotes. NOTE: a forked process is required to use this feature.

Name	`jvmArguments`
Type	`java.lang.String`
Default value	

User property	`spring-boot.run.jvmArguments`
Since	`1.1.0`

`mainClass`

The name of the main class. If not specified the first compiled class found that contains a 'main' method will be used.

Name	`mainClass`
Type	`java.lang.String`
Default value	
User property	`spring-boot.run.main-class`
Since	`1.0.0`

`maxAttempts`

The maximum number of attempts to check if the spring application is ready. Combined with the "wait" argument, this gives a global timeout value (30 sec by default)

Name	`maxAttempts`
Type	`int`
Default value	
User property	
Since	

`noverify`

Flag to say that the agent requires -noverify.

Name	`noverify`
Type	`boolean`
Default value	
User property	`spring-boot.run.noverify`

Since	1.0.0

profiles

The spring profiles to activate. Convenience shortcut of specifying the 'spring.profiles.active' argument. On command line use commas to separate multiple profiles.

Name	profiles
Type	java.lang.String[]
Default value	
User property	spring-boot.run.profiles
Since	1.3.0

skip

Skip the execution.

Name	skip
Type	boolean
Default value	false
User property	spring-boot.run.skip
Since	1.3.2

systemPropertyVariables

List of JVM system properties to pass to the process. NOTE: a forked process is required to use this feature.

Name	systemPropertyVariables
Type	java.util.Map
Default value	
User property	
Since	2.1.0

`useTestClasspath`

Flag to include the test classpath when running.

Name	`useTestClasspath`
Type	`java.lang.Boolean`
Default value	`false`
User property	`spring-boot.run.useTestClasspath`
Since	`1.3.0`

`wait`

The number of milli-seconds to wait between each attempt to check if the spring application is ready.

Name	`wait`
Type	`long`
Default value	
User property	
Since	

`workingDirectory`

Current working directory to use for the application. If not specified, basedir will be used. NOTE: a forked process is required to use this feature.

Name	`workingDirectory`
Type	`java.io.File`
Default value	
User property	`spring-boot.run.workingDirectory`
Since	`1.5.0`

8.3. `spring-boot:stop`

`org.springframework.boot:spring-boot-maven-plugin:2.4.7`

Stop an application that has been started by the "start" goal. Typically invoked once a test suite has completed.

8.3.1. Optional parameters

Name	Type	Default
fork	`Boolean`	
jmxName	`String`	
jmxPort	`int`	
skip	`boolean`	`false`

8.3.2. Parameter details

`fork`

Flag to indicate if the process to stop was forked. By default, the value is inherited from the `MavenProject` with a fallback on the default fork value (`true`). If it is set, it must match the value used to `StartMojo start` the process.

Name	`fork`
Type	`java.lang.Boolean`
Default value	
User property	`spring-boot.stop.fork`
Since	`1.3.0`

`jmxName`

The JMX name of the automatically deployed MBean managing the lifecycle of the application.

Name	`jmxName`
Type	`java.lang.String`
Default value	
User property	
Since	

`jmxPort`

The port to use to lookup the platform MBeanServer if the application has been forked.

Name	`jmxPort`
Type	`int`
Default value	
User property	
Since	

`skip`

Skip the execution.

Name	`skip`
Type	`boolean`
Default value	`false`
User property	`spring-boot.stop.skip`
Since	`1.3.2`

8.4. Examples

8.4.1. Random Port for Integration Tests

One nice feature of the Spring Boot test integration is that it can allocate a free port for the web application. When the `start` goal of the plugin is used, the Spring Boot application is started separately, making it difficult to pass the actual port to the integration test itself.

The example below showcases how you could achieve the same feature using the Build Helper Maven Plugin:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>build-helper-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>reserve-tomcat-port</id>
                        <goals>
                            <goal>reserve-network-port</goal>
                        </goals>
```

```
                        <phase>process-resources</phase>
                        <configuration>
                            <portNames>
                                <portName>tomcat.http.port</portName>
                            </portNames>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>pre-integration-test</id>
                        <goals>
                            <goal>start</goal>
                        </goals>
                        <configuration>
                            <arguments>
                                <argument>--server.port=${tomcat.http.port}</argument>
                            </arguments>
                        </configuration>
                    </execution>
                    <execution>
                        <id>post-integration-test</id>
                        <goals>
                            <goal>stop</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-failsafe-plugin</artifactId>
                <configuration>
                    <systemPropertyVariables>
                        <test.server.port>${tomcat.http.port}</test.server.port>
                    </systemPropertyVariables>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

You can now retrieve the `test.server.port` system property in any of your integration test to create a proper `URL` to the server.

8.4.2. Customize JMX port

The `jmxPort` property allows to customize the port the plugin uses to communicate with the Spring Boot application.

This example shows how you can customize the port in case `9001` is already used:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                   <configuration>
                    <jmxPort>9009</jmxPort>
                </configuration>
                <executions>
                    <execution>
                        <id>pre-integration-test</id>
                        <goals>
                            <goal>start</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>post-integration-test</id>
                        <goals>
                            <goal>stop</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

If you need to configure the JMX port, make sure to do so in the global configuration as shown above so that it is shared by both goals.

8.4.3. Skip Integration Tests

The `skip` property allows to skip the execution of the Spring Boot maven plugin altogether.

This example shows how you can skip integration tests with a command-line property and still make sure that the `repackage` goal runs:

```
<project>
    <properties>
        <skip.it>false</skip.it>
    </properties>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>pre-integration-test</id>
                        <goals>
                            <goal>start</goal>
                        </goals>
                        <configuration>
                            <skip>${skip.it}</skip>
                        </configuration>
                    </execution>
                    <execution>
                        <id>post-integration-test</id>
                        <goals>
                            <goal>stop</goal>
                        </goals>
                        <configuration>
                            <skip>${skip.it}</skip>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-failsafe-plugin</artifactId>
                <configuration>
                    <skip>${skip.it}</skip>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

By default, the integration tests will run but this setup allows you to easily disable them on the command-line as follows:

```
$ mvn verify -Dskip.it=true
```

Chapter 9. Integrating with Actuator

Spring Boot Actuator displays build-related information if a `META-INF/build-info.properties` file is present. The `build-info` goal generates such file with the coordinates of the project and the build time. It also allows you to add an arbitrary number of additional properties, as shown in the following example:

```
<project>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <goals>
                            <goal>build-info</goal>
                        </goals>
                        <configuration>
                            <additionalProperties>
                                <encoding.source>UTF-8</encoding.source>
                                <encoding.reporting>UTF-8</encoding.reporting>
                                <java.source>${maven.compiler.source}</java.source>
                                <java.target>${maven.compiler.target}</java.target>
                            </additionalProperties>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

This configuration will generate a `build-info.properties` at the expected location with four additional keys.

`maven.compiler.source` and `maven.compiler.target` are expected to be regular properties available in the project. They will be interpolated as you would expect.

9.1. `spring-boot:build-info`

`org.springframework.boot:spring-boot-maven-plugin:2.4.7`

Generate a `build-info.properties` file based on the content of the current `MavenProject`.

9.1.1. Optional parameters

Name	Type	Default
additionalProperties	Map	
outputFile	File	${project.build.outputDirectory}/META-INF/build-info.properties
time	String	

9.1.2. Parameter details

`additionalProperties`

Additional properties to store in the `build-info.properties` file. Each entry is prefixed by `build.` in the generated `build-info.properties`.

Name	additionalProperties
Type	java.util.Map
Default value	
User property	
Since	

`outputFile`

The location of the generated `build-info.properties` file.

Name	outputFile
Type	java.io.File
Default value	${project.build.outputDirectory}/META-INF/build-info.properties
User property	
Since	

`time`

The value used for the `build.time` property in a form suitable for `Instant#parse(CharSequence)`. Defaults to `session.request.startTime`. To disable the `build.time` property entirely, use `'off'`.

Name	time
Type	java.lang.String
Default value	

User property	
Since	2.2.0

Chapter 10. Help Information

The `help` goal is a standard goal that displays information on the capabilities of the plugin.

10.1. `spring-boot:help`

`org.springframework.boot:spring-boot-maven-plugin:2.4.7`

Display help information on spring-boot-maven-plugin. Call `mvn spring-boot:help -Ddetail=true -Dgoal=<goal-name>` to display parameter details.

10.1.1. Optional parameters

Name	Type	Default
detail	`boolean`	`false`
goal	`String`	
indentSize	`int`	`2`
lineLength	`int`	`80`

10.1.2. Parameter details

`detail`

If `true`, display all settable properties for each goal.

Name	`detail`
Type	`boolean`
Default value	`false`
User property	`detail`
Since	

`goal`

The name of the goal for which to show help. If unspecified, all goals will be displayed.

Name	`goal`
Type	`java.lang.String`
Default value	

User property	`goal`
Since	

`indentSize`

The number of spaces per indentation level, should be positive.

Name	`indentSize`
Type	`int`
Default value	`2`
User property	`indentSize`
Since	

`lineLength`

The maximum length of a display line, should be positive.

Name	`lineLength`
Type	`int`
Default value	`80`
User property	`lineLength`
Since	