Usage

The plugin provides several goals to work with a Spring Boot application:

- `repackage` : create a jar or war file that is auto-executable. It can replace the regular artifact or can be attached to the build lifecycle with a separate **classifier**.
- `run` : run your Spring Boot application with several options to pass parameters to it.
- `start` and `stop` : integrate your Spring Boot application to the `integration-test` phase so that the application starts before it.
- `build-info` : generate a build information that can be used by the Actuator.

Each goal is further described below.

Repackaging an application

In order to repackage your application, you simply need to add a reference to the plugin in your `pom.xml` :

```
<build>
  ...
  <plugins>
    ...
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>2.1.8.RELEASE</version>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    ...
  </plugins>
  ...
</build>
```

The example above repackages a jar or war 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 Exclude a dependency (./examples/exclude-dependency.html) for more details. Please note that 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 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 plugin rewrites your manifest, and in particular it manages the **Main-Class** and **Start-Class** entries, so if the defaults don't work you have to configure those there (not in the jar plugin). The **Main-Class** in the manifest is actually controlled by the **layout** property of the boot plugin, e.g.

```
<build>
  ...
  <plugins>
    ...
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>2.1.8.RELEASE</version>
      <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 guess based on 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.

For more detailed examples of how to configure this goal see:

- Custom repackage classifier (./examples/repackage-classifier.html)
- Exclude a dependency (./examples/exclude-dependency.html)

Running the application

The plugin includes a run goal which can be used to launch your application from the command line:

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

By default the application is executed directly from the Maven JVM. If you need to run in a forked process you can use the 'fork' option. Forking will also occur if the 'jvmArguments', 'systemPropertyVariables', 'environmentVariables' or 'agent' options are specified, or if devtools is present.

If you need to specify some JVM arguments (i.e. for debugging purposes), you can use the `jvmArguments` parameter, see Debug the application (./examples/run-debug.html) for more details. There is also explicit support for system properties (./examples/run-system-properties.html) and environment variables (./examples/run-env-variables.html).

As a convenience, the profiles to enable are handled by a specific property (`profiles`), see Specify active profiles (./examples/run-profiles.html).

Spring Boot 1.3 has introduced `devtools` , 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>
    <version>2.1.8.RELEASE</version>
    <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>
      <version>2.1.8.RELEASE</version>
      <configuration>
        <addResources>true</addResources>
      </configuration>
    </plugin>
    ...
  </plugins>
  ...
</build>
```

When `addResources` is enabled, any **src/main/resources** folder 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.

Note that 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. See Exclude a dependency (./examples/exclude-dependency.html) for more details.

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

Working with 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 described below:

```
<build>
  ...
  <plugins>
    ...
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>2.1.8.RELEASE</version>
      <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 (https://maven.apache.org/surefire/maven-failsafe-plugin/) to run your integration tests as you would expect.

You could also configure a more advanced setup to skip the integration tests when a specific property has been set:

```
<properties>
  <it.skip>false</it.skip>
</properties>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
      <configuration>
        <skip>${it.skip}</skip>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>2.1.8.RELEASE</version>
      <executions>
        <execution>
          <id>pre-integration-test</id>
          <goals>
            <goal>start</goal>
          </goals>
          <configuration>
            <skip>${it.skip}</skip>
          </configuration>
        </execution>
        <execution>
          <id>post-integration-test</id>
          <goals>
            <goal>stop</goal>
          </goals>
          <configuration>
            <skip>${it.skip}</skip>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

If you run `mvn verify -Dit.skip=true` your integration tests will be skipped altogether.

For more detailed examples of how to configure this goal see:

- Random port for integration tests (./examples/it-random-port.html)