How to Create an Executable JAR with Maven

Last modified: December 24, 2021

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

Maven (https://www.baeldung.com/category/maven)

1. Overview

In this quick tutorial, we'll focus on **packaging a Maven project into an executable Jar file.**

When creating a *jar* file, we usually want to run it easily, without using the IDE. To that end, we'll discuss the configuration and pros/cons of using each of these approaches for creating the executable.

Further reading:

Apache Maven Tutorial (/maven)

A quick and practical guide to building and managing Java projects using Apache Maven.

Read more (/maven) →

Where is the Maven Local Repository? (/maven-local-repository)

Quick tutorial showing you where Maven stores its local repo and how to change that.

Read more (/maven-local-repository) →

Spring with Maven BOM (/spring-maven-bom)

Learn how to use a BOM, Bill of Materials, in your Spring Maven project.

Read more (/spring-maven-bom) →

2. Configuration

We don't need any additional dependencies to create an executable *jar*. We just need to create a Maven Java project and have at least one class with the *main(…)* method.

In our example, we created Java class named *ExecutableMavenJar*.

We also need to make sure that our *pom.xml* contains these elements:

```
<modelVersion>4.0.0</modelVersion>
<groupId>com.baeldung</groupId>
<artifactId>core-java</artifactId>
<version>0.1.0-SNAPSHOT</version>
<packaging>jar</packaging>
```

The most important aspect here is the type — to create an executable *jar*, double-check the configuration uses a *jar* type.

Now we can start using the various solutions.

2.1. Manual Configuration

Let's start with a manual approach with the help of the *maven-dependency-plugin*.

We'll begin by copying all required dependencies into the folder that we'll specify:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
        <execution>
            <id>copy-dependencies</id>
            <phase>prepare-package</phase>
            <goals>
                <goal>copy-dependencies</goal>
            </goals>
            <configuration>
                <outputDirectory>
                    ${project.build.directory}/libs
                </outputDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>
```

There are two important aspects to notice.

First, we specify the goal *copy-dependencies*, which tells Maven to copy these dependencies into the specified *outputDirectory*. In our case, we'll create a folder named *libs* inside the project build directory (which is usually the *target* folder).

Second, we are going to create executable and classpath-aware *jar*, with the link to the dependencies copied in the first step:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <configuration>
        <archive>
            <manifest>
                <addClasspath>true</addClasspath>
                <classpathPrefix>libs/</classpathPrefix>
                <mainClass>
                    com.baeldung.executable.ExecutableMavenJar
                </mainClass>
            </manifest>
        </archive>
    </configuration>
</plugin>
```

The most important part of this is the *manifest* configuration. We add a classpath, with all dependencies (folder *libs/*), and provide the information about the main class.

Please note that we need to provide a fully qualified name of the class, which means it will include package name.

The advantages and disadvantages of this approach are:

- **pros** – transparent process, where we can specify each step
- **cons** – manual; dependencies are out of the final *jar*, which means that our executable *jar* will only run if the *libs* folder will be accessible and visible for a *jar*

2.2. Apache Maven Assembly Plugin

The Apache Maven Assembly Plugin allows users to aggregate the project output along with its dependencies, modules, site documentation, and other files into a single, runnable package.

The main goal in the assembly plugin is the *single (https://maven.apache.org/plugins/maven-assembly-plugin/single-mojo.html)* goal, which is used to create all assemblies (all other goals are deprecated and will be removed in a future release).

Let's take a look at the configuration in *pom.xml*:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
            <configuration>
                <archive>
                <manifest>
                    <mainClass>
                        com.baeldung.executable.ExecutableMavenJar
                    </mainClass>
                </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Similarly to the manual approach, we need to provide the information about the main class. The difference is that the Maven Assembly Plugin will automatically copy all required dependencies into a *jar* file.

In the *descriptorRefs* part of the configuration code, we provided the name that will be added to the project name.

Output in our example will be named as *core-java-jar-with-dependencies.jar*.

- **pros –** dependencies inside the *jar* file, one file only
- **cons –** basic control of packaging our artifact, for example, there is no class relocation support

2.3. Apache Maven Shade Plugin

Apache Maven Shade Plugin provides the capability to package the artifact in an *uber-jar*, which consists of all dependencies required to run the project. Moreover, it supports shading — i.e. renaming — the packages of some of the dependencies.

Let's take a look at the configuration:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>shade</goal>
            </goals>
            <configuration>
                <shadedArtifactAttached>true</shadedArtifactAttached>
                <transformers>
                    <transformer implementation=

"org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">

<mainClass>com.baeldung.executable.ExecutableMavenJar</mainClass>
                </transformer>
            </transformers>
        </configuration>
        </execution>
    </executions>
</plugin>
```

There are three main parts to this configuration.

First, *<shadedArtifactAttached>* marks all dependencies to be packaged into the *jar*.

Second, we need to specify the transformer implementation (https://maven.apache.org/plugins/maven-shade-plugin/usage.html); we used the standard one in our example.

Finally, we need to specify the main class of our application.

The output file will be named *core-java-0.1.0-SNAPSHOT-shaded.jar*, where *core-java* is our project name followed by snapshot version and plugin name.

- **pros –** dependencies inside the *jar* file, advanced control of packaging our artifact, with shading and class relocation
- **cons** – complex configuration (especially if we want to use advanced features)

2.4. One Jar Maven Plugin

Another option to create an executable *jar* is the One Jar project.

This provides a custom class loader that knows how to load classes and resources from jars inside an archive, instead of from *jars* in the filesystem.

Let's take a look at the configuration:

```
<plugin>
    <groupId>com.jolira</groupId>
    <artifactId>onejar-maven-plugin</artifactId>
    <executions>
        <execution>
            <configuration>
                <mainClass>org.baeldung.executable.
                  ExecutableMavenJar</mainClass>
                <attachToBuild>true</attachToBuild>
                <filename>
                  ${project.build.finalName}.${project.packaging}
                </filename>
            </configuration>
            <goals>
                <goal>one-jar</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

As shown in the configuration, we need to specify the main class and attach all dependencies to build, by using *attachToBuild = true*.

Also, we should provide the output filename. Moreover, the goal for Maven is *one-jar*. Please note that One Jar is a commercial solution that will make dependency *jars* not expanded into the filesystem at runtime.

- **pros –** clean delegation model, allows classes to be at the top level of the One Jar, supports external *jars* and can support Native libraries
- **cons –** not actively supported since 2012

2.5. Spring Boot Maven Plugin

Finally, the last solution we'll look at is the Spring Boot Maven Plugin.

This allows for packaging executable *jar* or *war* archives and run an application "in place."

To use it, we need to use at least Maven version 3.2. The detailed description is available here (https://docs.spring.io/spring-boot/docs/1.4.1.RELEASE/maven-plugin/).

Let's have a look at the config:

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>repackage</goal>
            </goals>
            <configuration>
                <classifier>spring-boot</classifier>
                <mainClass>
                  com.baeldung.executable.ExecutableMavenJar
                </mainClass>
            </configuration>
        </execution>
    </executions>
</plugin>
```

There are two differences between Spring plugin and the others: The goal of the execution is called *repackage*, and the classifier is named *spring-boot*.

Note that we don't need to have Spring Boot application in order to use this plugin.

- **pros –** dependencies inside a *jar* file, we can run it in every accessible location, advanced control of packaging our artifact, with excluding dependencies from the *jar* file etc., packaging of *war* files as well
- **cons –** adds potentially unnecessary Spring and Spring Boot-related classes

2.6. Web Application With Executable Tomcat

In the last part, we want to cover a standalone web application that is packed inside a *jar* file.

In order to do that, we need to use different plugin, designed for creating executable jar files:

```
<plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <version>2.0</version>
    <executions>
        <execution>
            <id>tomcat-run</id>
            <goals>
                <goal>exec-war-only</goal>
            </goals>
            <phase>package</phase>
            <configuration>
                <path>/</path>
                <enableNaming>false</enableNaming>
                <finalName>webapp.jar</finalName>
                <charset>utf-8</charset>
            </configuration>
        </execution>
    </executions>
</plugin>
```

The *goal* is set as *exec-war-only*, *path* to our server is specified inside *configuration* tag, with additional properties, like *finalName*, *charset* etc.

To build a *jar*, we run *man package*, which will result in creating *webapp.jar* in our *target* directory.

To run the application, we just write *java -jar target/webapp.jar* in our console and try to test it by specifying the *localhost:8080/* in a browser.

- **pros** – having one file, easy to deploy and run
- **cons** – a size of the file is much larger, due to packing Tomcat embedded distribution inside a w*ar* file

Note that this is the latest version of this plugin, which supports Tomcat7 server. To avoid errors, we can check that our dependency for Servlets has *scope* set as *provided*, otherwise, there will be a conflict at the *runtime* of executable *jar*.

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <scope>provided</scope>
</dependency>
```

3. Conclusion

In this article, we described many ways of creating an executable *jar* with various Maven plugins.

The full implementation of this tutorial can be found in these GitHub projects: executable jar (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-jar) and executable war (https://github.com/eugenp/tutorials/tree/master/spring-web-modules/spring-thymeleaf-5).

How to test? In order to compile the project into an executable *jar*, please run Maven with *mvn clean package* command.

This article hopefully provided some more insights and will help you find your preferred approach depending on your needs.

One quick final note: We want to make sure the licenses of the jars we're bundling don't prohibit this kind of operation. That generally won't be the case, but it's worth considering.

Comments are closed on this article!

COURSES

ALL COURSES (/ALL-COURSES)
ALL BULK COURSES (/ALL-BULK-COURSES)
ALL BULK TEAM COURSES (/ALL-BULK-TEAM-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
JOBS (/TAG/ACTIVE-JOB/)
OUR PARTNERS (/PARTNERS)
PARTNER WITH BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)