00.01 Creación de un JAR ejecutable con Java con Apache Maven

* [1. Introducción](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/00configjava/01jarwithmaven/#1-introducci%C3%B3n)
* [2. Configuración manual](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/00configjava/01jarwithmaven/#2-configuraci%C3%B3n-manual)
  + [2.1. Configuración del archivo pom.xml](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/00configjava/01jarwithmaven/#21-configuraci%C3%B3n-del-archivo-pomxml)
  + [Añadir bibliotecas con las dependencias](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/00configjava/01jarwithmaven/#a%C3%B1adir-bibliotecas-con-las-dependencias)
  + [Creación de un jar ejecutable](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/00configjava/01jarwithmaven/#creaci%C3%B3n-de-un-jar-ejecutable)
  + [2.2. Incorporación de las dependencias dentro del JAR: maven-assembly-plugin](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/00configjava/01jarwithmaven/#22-incorporaci%C3%B3n-de-las-dependencias-dentro-del-jar-maven-assembly-plugin)
  + [2.3. Plugin Maven Shade: maven-shade-plugin](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/00configjava/01jarwithmaven/#23-plugin-maven-shade-maven-shade-plugin)
  + [2.4. Plugin One Jar Maven: onejar-maven-plugin](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/00configjava/01jarwithmaven/#24-plugin-one-jar-maven-onejar-maven-plugin)
  + [2.5. Plugin Spring Boot Maven: spring-boot-maven-plugin](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/00configjava/01jarwithmaven/#25-plugin-spring-boot-maven-spring-boot-maven-plugin)
  + [2.6. Aplicación Web ejecutable Tomcat: tomcat7-maven-plugin](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/00configjava/01jarwithmaven/#26-aplicaci%C3%B3n-web-ejecutable-tomcat-tomcat7-maven-plugin)

# 1. Introducción

Este documento explicaremos los distintos modos de empaquetar un proyecto Maven en un archivo Jar ejecutable.

Cuando creamos un archivo jar, generalmente queremos ejecutarlo fácilmente, sin utilizar el IDE. Muchas veces nos vemos sorprendidos al comprobar que el archivo de manifiesto no incorpora referencias al main o no so incluyen las bibliotecas.

Veremos varias configuraciones y pros/contras de cada uno de estos enfoques para crear un JAR ejecutable.

En este artículo, describimos muchas formas de crear un jar ejecutable con varios complementos de Maven.

# 2. Configuración manual

Este modo de hacerlo nos da flexiblidad, pues sólo se requiere de un proyecto maven y añadir los elementos necesarios al fichero de configuración de maven, pom.xml.

**No necesitamos ninguna dependencia adicional para crear un archivo jar ejecutable**. sólo necesitamos crear un proyecto Java Maven y tener al menos una clase con el método main(…), la entrada al programa.

En nuestro ejemplo, creamos una clase Java llamada AppExemplo.

También debemos asegurarnos de que nuestro pom.xml contenga estos elementos:

<modelVersion>4.0.0</modelVersion>

<groupId>com.pepinho</groupId>

<artifactId>core-java</artifactId>

<version>0.1.0-SNAPSHOT</version>

<packaging>jar</packaging>

Lo más importante aquí es el tipo de empaquetamiento, para crear un jar ejecutable el packaging tipo de jar.

Ahora podemos comenzar a usar las diversas soluciones.

## 2.1. Configuración del archivo pom.xml

### Añadir bibliotecas con las dependencias

Comencemos con un enfoque manual con la ayuda del [maven-dependency-plugin](https://maven.apache.org/plugins/maven-dependency-plugin/) y [copy-dependencies](https://maven.apache.org/plugins/maven-dependency-plugin/copy-dependencies-mojo.html), que copia las dependencias del proyecto desde el repositorio a una ubicación definida, en este caso el directorio libs:

<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>

Hay dos aspectos importantes a tener en cuenta.

Primero, especificamos la meta copy-dependencies, que le dice a Maven que copie estas dependencias en el directorio de salida especificado. En nuestro caso, crearemos una carpeta llamada libs dentro del directorio de despliegue del proyecto (que suele llamarse target).

### Creación de un jar ejecutable

En segundo lugar, vamos a [**crear un jar ejecutable**](https://maven.apache.org/plugins/maven-jar-plugin/) indicándoles la ruta a las biblitecas anteriores en el classpath, dentro del [plugin que permite crear jar](https://maven.apache.org/plugins/maven-jar-plugin/plugin-info.html) :

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-jar-plugin</artifactId>

<configuration>

<archive>

<manifest>

<addClasspath>true</addClasspath>

<classpathPrefix>libs/</classpathPrefix>

<mainClass>

com.pepinho.executable.AppExemplo

</mainClass>

</manifest>

</archive>

</configuration>

</plugin>

La parte más importante de esto es la configuración del archivo que se produce dentro del  [archivo MANIFEST.MF](https://www.baeldung.com/java-jar-manifest)  del JAR

<https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html>

Con esta configuración de Maven **se añaden las rutas a las bibliotecas del proyecto** dentro de la carpeta libs/ y se indica **qué clase tiene el main**.

El nombre de la clase con el main tiene que estar completamente calificado, incluytendo el nombre del paquete.

Las ventajas y desventajas de este enfoque son:

* **Ventajas:** Proceso transparente, donde podemos especificar cada paso
* **Inconvenientes**: se trata de un proceso manual, en el que las **dependencias están fuera del jar final**, lo que significa que nuestro jar ejecutable únicamente se ejecutará si la carpeta libs es accesible y visible para un jar.

## 2.2. Incorporación de las dependencias dentro del JAR: maven-assembly-plugin

El [Plugin de ensamblado de Apache Maven](https://maven.apache.org/plugins/maven-assembly-plugin/) maven-assembly-plugin permite agregar al paquete de salida del proyecto (jar en este caso), módulos, documentación del sitio y otros archivos en un único paquete ejecutable ("*…permite a los desarrolladores combinar los resultados del proyecto en un único archivo distribuible que también contiene dependencias, módulos, documentación del sitio y otros archivos*").

El objetivo principal (y ahora único) en el plugin de ensamblaje es el **crear un archivo único**, que se utiliza para crear todos los ensamblajes.

Puedes consultar el uso de este plugin en la [página oficial de Apeche Maven](https://maven.apache.org/plugins/maven-assembly-plugin/usage.html)

La configuración del **pom.xml** debe ser algi similar a lo siguiente:

<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.pepinho.executable.AppExemplo

</mainClass>

</manifest>

</archive>

<descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

</configuration>

</execution>

</executions>

</plugin>

 Clase con el main

Del mismo modo que añadiendo las bibliotecas a un directorio lib, De manera similar al enfoque manual, necesitamos **indicar el nombre de la clase con el main**. La diferencia es que el Plugin de ensamblaje de Maven **copiará automáticamente todas las dependencias necesarias dentro del mismo archivo jar**.

En la parte descriptorRefs del código de configuración, se indica el nombre que se agregará al nombre del proyecto (puede cambiarse)

La salida en nuestro ejemplo se llamará core-java-jar-with-dependencies.jar: “[*Nota*](https://maven.apache.org/plugins/maven-assembly-plugin/usage.html): … Tenga en cuenta que el complemento de ensamblaje le permite especificar varios descriptorRefsa la vez para producir múltiples tipos de ensamblajes en una sola invocación.”

* **Ventajas**: las dependencias se añaden dentro de un único archivo jar, dándole portabilidad “total”.
* **Desventajas:** no podemos reubicar las calses del proyecto. A veces, el tamaño puede aumentar considerablemente y sólo precisamos distribuir nuestras clases.

## 2.3. Plugin Maven Shade: maven-shade-plugin

El [plugin de sombreado de Apache Maven](https://maven.apache.org/plugins/maven-shade-plugin/) proporciona la capacidad de empaquetar el artefacto en un uber-jar, “*..proporciona la capacidad de empaquetar el artefacto en un uber-jar, incluidas sus dependencias y sombrear (es decir,****cambiar el nombre) los paquetes de algunas de las dependencias****.*”

Esto es, una vez creado el JAR permite cambiar las dependencias si se precisa en algún momento (versiones, etc.). Uso del plugin: <https://maven.apache.org/plugins/maven-shade-plugin/usage.html>

Configuración del pom.xml:

<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.pepinho.executable.AppExemplo</mainClass>

</transformer>

</transformers>

</configuration>

</execution>

</executions>

</plugin>

El archivo de configuración tiene 3 partes principales:

* Necesitamos especificar la clase principal de la aplicación: **com.pepinho.executable.AppExemplo**
* **<shadedArtifactAttached>** indicas las dependencias que deben ser empaquetadas en el jar.
* Debe indicarse la implementación del transformador, **transformer implementation=**, que en el ejemplo se emplea el estándar que añade las entradas al archiv MANIFEST.

 Implementaciones del transformer del plugin

Transformadores del plugin de org.apache.maven.plugins.shade.resourceson:

|  |  |
| --- | --- |
| Transformer | Descripción |
| ApacheLicenseResourceTransformer | Evita la duplicación de licencias |
| ApacheNoticeResourceTransformer | Prepara el NOTICE combinado |
| AppendingTransformer | Agrega contenido a un recurso |
| ComponentsXmlResourceTransformer | Agrega el archivo components.xml de Plexus |
| DontIncludeResourceTransformer | Evita la inclusión de recursos coincidentes |
| IncludeResourceTransformer | Agrega archivos del proyecto |
| ManifestResourceTransformer | Establece entradas en el MANIFEST |
| ServicesResourceTransformer | Fusiona los recursos META-INF/services |
| XmlAppendingTransformer | Agrega contenido XML a un recurso XML |

El archivo de salida se llamará core-java-0.1.0-SNAPSHOT-shaded.jar, donde core-java es el nombre de nuestro proyecto seguido por la versión de snapshot y el nombre del plugin.

* **Ventajas**: dependencias dentro del archivo jar, control avanzado del empaquetado del proyecto, con sombreado y reubicación de clases.
* **Deventajas:** configuración compleja (especialmente si queremos usar funciones avanzadas).

## 2.4. Plugin One Jar Maven: onejar-maven-plugin

Otra opción, **poco recomendable (como curiosidad)**, menos interesante y comercial para crear un jar ejecutable es el [proyecto **One Jar**](https://github.com/jolira/onejar-maven-plugin), que proporciona un **loader de clases personalizado que sabe cómo cargar clases y recursos desde archivos jar dentro de un archivo**, en lugar de desde archivos jar en el sistema de archivos. [Requiere dependencias](https://central.sonatype.com/artifact/com.jolira/onejar-maven-plugin).

Configuración del **pom.xml**:

<plugin>

<groupId>com.jolira</groupId>

<artifactId>onejar-maven-plugin</artifactId>

<executions>

<execution>

<configuration>

<mainClass>com.pepinho.executable.AppExemplo</mainClass>

<attachToBuild>true</attachToBuild>

<filename>

${project.build.finalName}.${project.packaging}

</filename>

</configuration>

<goals>

<goal>one-jar</goal>

</goals>

</execution>

</executions>

</plugin>

* Se debe especificar la clase principal y adjuntar todas las dependencias a la construcción, utilizando attachToBuild = true.
* Se debe proporcionar el nombre de archivo de salida.
* El objetivo para Maven es **one-jar**.

**One Jar es una solución comercial** que hará que las dependencias no se expandan en el sistema de archivos en tiempo de ejecución.

* **Ventajas:** modelo de delegación limpio, permite que las clases estén en el nivel superior de One Jar, admite archivos jar externos y puede admitir bibliotecas nativas
* **Desventajas:** no es compatible desde 2012

## 2.5. Plugin Spring Boot Maven: spring-boot-maven-plugin

Otra opción interesante es el [Plugin Spring Boot Maven](https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle/), que permite **empaquetar archivos jar o war ejecutables y ejecutar una aplicación “in situ”**.

Para usarlo, **necesitamos usar al menos la versión 3.2 de Maven**. La [descripción detallada está disponible esta disponible en la página de Spring](https://docs.spring.io/spring-boot/docs/1.4.1.RELEASE/maven-plugin/).

Configuración del **pom.xml**:

<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.pepinho.executable.AppExemplo

</mainClass>

</configuration>

</execution>

</executions>

</plugin>

Hay dos diferencias entre el plugin Spring y los demás:

El objetivo (goal) de la ejecución se llama repackage, y el clasificador (classifier) se llama spring-boot.

NO NECESITAMOS tener una aplicación Spring Boot para usar este plugin.

* **Ventajas**: dependencias dentro de un archivo jar, podemos ejecutarlo en cualquier ubicación accesible, control avanzado del empaquetado del proyecto, excluyendo dependencias del archivo jar, etc., empaquetado de otros tipos de archivo como war
* **Desventajas**: añade clases innecesarias de Spring y Spring Boot, pues no requiere usar Spring Boot para emplear este plugin.

## 2.6. Aplicación Web ejecutable Tomcat: tomcat7-maven-plugin

Por último, si queremos hacer una aplicación web independiente que esté empaquetada dentro de un archivo jar.

Necesitamos usar un plugin diferente, diseñado para crear archivos jar ejecutables:

Configuración del **pom.xml**:

<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>

El **goal** está configurado como **exec-war-only**, la ruta al servidor se especifica dentro de la etiqueta de **configuration**, con propiedades adicionales, como **finalName, charset**, etc.

Para construir un jar, ejecutamos **mvn package**, lo que dará como resultado la creación de **webapp.jar** en el directorio **target**.

Para ejecutar la aplicación, simplemente **escribimos java -jar target/webapp.jar en la consola y tratamos de probarlo especificando el localhost:8080/ en un navegador**. Ya tenemos nuestra aplicación Web ejecutándose desde línea de órdenes un archivo JAR ;-)

* Ventajas: tener un único archivo, fácil de implementar y ejecutar
* **Desventajas**: el tamaño del archivo es mucho mayor, debido al empaquetado de la distribución integrada de Tomcat dentro de un archivo jar.

 Implementaciones del transformer del plugin

Ten en cuenta que esta es la última versión de este plugin, que **admite el servidor Tomcat7**. Para evitar errores, podemos verificar que la dependencia para Servlets **tenga el ámbito configurado como provided**, de lo contrario, habría un conflicto en el tiempo de ejecución del jar ejecutable:

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<scope>provided</scope>

</dependency>