The Executable Jar Format

Back to index

The `spring-boot-loader` modules lets Spring Boot support executable jar and war files. If you use the Maven plugin or the Gradle plugin, executable jars are automatically generated, and you generally do not need to know the details of how they work.

If you need to create executable jars from a different build system or if you are just curious about the underlying technology, this appendix provides some background.

1. Nested JARs

Java does not provide any standard way to load nested jar files (that is, jar files that are themselves contained within a jar). This can be problematic if you need to distribute a self-contained application that can be run from the command line without unpacking.

To solve this problem, many developers use “shaded” jars. A shaded jar packages all classes, from all jars, into a single “uber jar”. The problem with shaded jars is that it becomes hard to see which libraries are actually in your application. It can also be problematic if the same filename is used (but with different content) in multiple jars. Spring Boot takes a different approach and lets you actually nest jars directly.

1.1. The Executable Jar File Structure

Spring Boot Loader-compatible jar files should be structured in the following way:

```
example.jar
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |      +-boot
 |          +-loader
 |              +-<spring boot loader classes>
 +-BOOT-INF
    +-classes
    |  +-mycompany
    |      +-project
    |          +-YourClasses.class
    +-lib
       +-dependency1.jar
       +-dependency2.jar
```

Application classes should be placed in a nested `BOOT-INF/classes` directory.

Dependencies should be placed in a nested `BOOT-INF/lib` directory.

1.2. The Executable War File Structure

Spring Boot Loader-compatible war files should be structured in the following way:

```
example.war
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |      +-boot
 |          +-loader
 |              +-<spring boot loader classes>
 +-WEB-INF
    +-classes
    |  +-com
    |      +-mycompany
    |          +-project
    |              +-YourClasses.class
    +-lib
    |  +-dependency1.jar
    |  +-dependency2.jar
    +-lib-provided
        +-servlet-api.jar
        +-dependency3.jar
```

Dependencies should be placed in a nested WEB-INF/lib directory. Any dependencies that are required when running embedded but are not required when deploying to a traditional web container should be placed in WEB-INF/lib-provided.

1.3. Index Files

Spring Boot Loader-compatible jar and war archives can include additional index files under the BOOT-INF/ directory. A classpath.idx file can be provided for both jars and wars, and it provides the ordering that jars should be added to the classpath. The layers.idx file can be used only for jars, and it allows a jar to be split into logical layers for Docker/OCI image creation.

Index files follow a YAML compatible syntax so that they can be easily parsed by third-party tools. These files, however, are *not* parsed internally as YAML and they must be written in exactly the formats described below in order to be used.

1.4. Classpath Index

The classpath index file can be provided in BOOT-INF/classpath.idx. Typically, it is generated automatically by Spring Boot's Maven and Gradle build plugins. It provides a list of jar names (including the directory) in the order that they should be added to the classpath. When generated by the build plugins, this classpath ordering matches that used by the build system for running and testing the application. Each line must start with dash space ("-·") and names must be in double quotes.

For example, given the following jar:

```
example.jar
 |
 +-META-INF
 |   +-...
 +-BOOT-INF
     +-classes
     |   +...
     +-lib
         +-dependency1.jar
         +-dependency2.jar
```

The index file would look like this:

```
- "BOOT-INF/lib/dependency2.jar"
- "BOOT-INF/lib/dependency1.jar"
```

1.5. Layer Index

The layers index file can be provided in `BOOT-INF/layers.idx`. It provides a list of layers and the parts of the jar that should be contained within them. Layers are written in the order that they should be added to the Docker/OCI image. Layers names are written as quoted strings prefixed with dash space ("-·") and with a colon (":") suffix. Layer content is either a file or directory name written as a quoted string prefixed by space space dash space ("··-·"). A directory name ends with /, a file name does not. When a directory name is used it means that all files inside that directory are in the same layer.

A typical example of a layers index would be:

```
- "dependencies":
  - "BOOT-INF/lib/dependency1.jar"
  - "BOOT-INF/lib/dependency2.jar"
- "application":
  - "BOOT-INF/classes/"
  - "META-INF/"
```

2. Spring Boot’s “NestedJarFile” Class

The core class used to support loading nested jars is `org.springframework.boot.loader.jar.NestedJarFile`. It lets you load jar content from nested child jar data. When first loaded, the location of each JarEntry is mapped to a physical file offset of the outer jar, as shown in the following example:

```
myapp.jar
+-------------------+-------------------------+
| /BOOT-INF/classes | /BOOT-INF/lib/mylib.jar |
|+-----------------+||+-----------+----------+|
||     A.class      |||  B.class  |  C.class ||
|+-----------------+||+-----------+----------+|
+-------------------+-------------------------+
 ^                    ^           ^
 0063                 3452        3980
```

The preceding example shows how A.class can be found in /BOOT-INF/classes in myapp.jar at position 0063. B.class from the nested jar can actually be found in myapp.jar at position 3452, and C.class is at position 3980. Armed with this information, we can load specific nested entries by seeking to the appropriate part of the outer jar. We do not need to unpack the archive, and we do not need to read all entry data into memory.

2.1. Compatibility With the Standard Java “JarFile”

Spring Boot Loader strives to remain compatible with existing code and libraries. org.springframework.boot.loader.jar.NestedJarFile extends from java.util.jar.JarFile and should work as a drop-in replacement.

Nested JAR URLs of the form jar:nested:/path/myjar.jar/!BOOT-INF/lib/mylib.jar!/B.class are supported and open a connection compatible with java.net.JarURLConnection. These can be used with Java’s URLClassLoader.

3. Launching Executable Jars

The org.springframework.boot.loader.launch.Launcher class is a special bootstrap class that is used as an executable jar's main entry point. It is the actual Main-Class in your jar file, and it is used to setup an appropriate ClassLoader and ultimately call your main() method.

There are three launcher subclasses (JarLauncher, WarLauncher, and PropertiesLauncher). Their purpose is to load resources (.class files and so on) from nested jar files or war files in directories (as opposed to those explicitly on the classpath). In the case of JarLauncher and WarLauncher, the nested paths are fixed. JarLauncher looks in BOOT-INF/lib/, and WarLauncher looks in WEB-INF/lib/ and WEB-INF/lib-provided/. You can add extra jars in those locations if you want more.

The PropertiesLauncher looks in BOOT-INF/lib/ in your application archive by default. You can add additional locations by setting an environment variable called LOADER_PATH or loader.path in loader.properties (which is a comma-separated list of directories, archives, or directories within archives).

3.1. Launcher Manifest

You need to specify an appropriate Launcher as the Main-Class attribute of META-INF/MANIFEST.MF. The actual class that you want to launch (that is, the class that contains a main method) should be specified in the Start-Class attribute.

The following example shows a typical MANIFEST.MF for an executable jar file:

```
Main-Class: org.springframework.boot.loader.launch.JarLauncher
Start-Class: com.mycompany.project.MyApplication
```

For a war file, it would be as follows:

```
Main-Class: org.springframework.boot.loader.launch.WarLauncher
Start-Class: com.mycompany.project.MyApplication
```

You need not specify Class-Path entries in your manifest file. The classpath is deduced from the nested jars.

4. PropertiesLauncher Features

PropertiesLauncher has a few special features that can be enabled with external properties (System properties, environment variables, manifest entries, or loader.properties). The following table describes these properties:

Key	Purpose
loader.path	Comma-separated Classpath, such as lib,${HOME}/app/lib. Earlier entries take precedence, like a regular -classpath on the javac command line.
loader.home	Used to resolve relative paths in loader.path. For example, given loader.path=lib, then ${loader.home}/lib is a classpath location (along with all jar files in that directory). This property is also used to locate a loader.properties file, as in the following example /opt/app It defaults to ${user.dir}.
loader.args	Default arguments for the main method (space separated).
loader.main	Name of main class to launch (for example, com.app.Application).
loader.config.name	Name of properties file (for example, launcher). It defaults to loader.

Key	Purpose
loader.config.location	Path to properties file (for example, classpath:loader.properties). It defaults to loader.properties.
loader.system	Boolean flag to indicate that all properties should be added to System properties. It defaults to false.

When specified as environment variables or manifest entries, the following names should be used:

Key	Manifest entry	Environment variable
loader.path	Loader-Path	LOADER_PATH
loader.home	Loader-Home	LOADER_HOME
loader.args	Loader-Args	LOADER_ARGS
loader.main	Start-Class	LOADER_MAIN
loader.config.location	Loader-Config-Location	LOADER_CONFIG_LOCATION
loader.system	Loader-System	LOADER_SYSTEM

Build plugins automatically move the Main-Class attribute to Start-Class when the uber jar is built. If you use that, specify the name of the class to launch by using the Main-Class attribute and leaving out Start-Class.

The following rules apply to working with PropertiesLauncher:

- loader.properties is searched for in loader.home, then in the root of the classpath, and then in classpath:/BOOT-INF/classes. The first location where a file with that name exists is used.
- loader.home is the directory location of an additional properties file (overriding the default) only when loader.config.location is not specified.

- loader.path can contain directories (which are scanned recursively for jar and zip files), archive paths, a directory within an archive that is scanned for jar files (for example, dependencies.jar!/lib), or wildcard patterns (for the default JVM behavior). Archive paths can be relative to loader.home or anywhere in the file system with a jar:file: prefix.
- loader.path (if empty) defaults to BOOT-INF/lib (meaning a local directory or a nested one if running from an archive). Because of this, PropertiesLauncher behaves the same as JarLauncher when no additional configuration is provided.
- loader.path can not be used to configure the location of loader.properties (the classpath used to search for the latter is the JVM classpath when PropertiesLauncher is launched).
- Placeholder replacement is done from System and environment variables plus the properties file itself on all values before use.
- The search order for properties (where it makes sense to look in more than one place) is environment variables, system properties, loader.properties, the exploded archive manifest, and the archive manifest.

5. Executable Jar Restrictions

You need to consider the following restrictions when working with a Spring Boot Loader packaged application:

- Zip entry compression: The ZipEntry for a nested jar must be saved by using the ZipEntry.STORED method. This is required so that we can seek directly to individual content within the nested jar. The content of the nested jar file itself can still be compressed, as can any other entry in the outer jar.
- System classLoader: Launched applications should use Thread.getContextClassLoader() when loading classes (most libraries and frameworks do so by default). Trying to load nested jar classes with ClassLoader.getSystemClassLoader() fails. java.util.Logging always uses the system classloader. For this reason, you should consider a different logging implementation.

6. Alternative Single Jar Solutions

If the preceding restrictions mean that you cannot use Spring Boot Loader, consider the following alternatives:

- Maven Shade Plugin
- JarClassLoader
- OneJar
- Gradle Shadow Plugin