CONTAINERS

Containerizing Apps with jlink

A JDK utility greatly facilitates containerizing your applications.

by Nicolas Fränkel

If you have tried working with Java modules, you might have realized that modularization is not easy. The first hurdle might be modularizing your own application, but many issues are due to the current state of modularization of third-party libraries. This is unfortunate, because once an application is modularized, it can be distributed as a standalone executable with a trimmed-down version of the JDK. In this era of containerization, that means small Docker images.

In this article, I explain how to use jlink, which is a command-line utility available since Java 9, to create easy-to-containerize Java executables. I'll start with a quick overview of modules. Then I'll demonstrate how to use jlink to create standalone executables and how beneficial jlink can be when used with Docker containers.

The complete source code and files for this article are available on GitHub. The source code and configuration files for the larger project in this article can also be found on GitHub.

Modularization and jlink

A class A may make use of other classes such as `java.util.List` at compile time (the compiler checks the dependency is available on the compiling classpath) or at runtime (in which case the JRE tries to resolve the same dependency on the runtime classpath).

One issue that occurs is that the JRE delivers many classes; some of them might not be used by the application, but they are bundled anyway. For example, application servers that run in headless mode still bundle graphical packages such as `javax.swing`.

Another issue that arises from dependence on the JRE is how visibility is managed in Java. For class `A` in package `ch.frankel.a` to be visible from class `B` in package `ch.frankel.b`, class `A` must have public visibility. With that in mind, it's impossible for third-party JAR libraries to cleanly separate their API classes and their internal classes into different packages. Historically, packages that were meant to be internal relied upon implicit naming, such as `ch.frankel.c.internal`. However, there was no technical way to enforce this constraint.

This section has been updated by the author based on comments from Mikalai Zaikin with the help of Nicolai Parlog.

Java 9 tried to address this problem by providing another way to manage accessibility: modules. There are several kinds of modules:

- Java and JDK Modules: The former are associated with subpackages of `java`, while the latter are associated with subpackages of `jdk`. Both are provided by the JDK.

- Explicit application modules: A JAR file can be made into a modular JAR by providing a `module-info` class at its root. In the module system, these are referred to as "explicit application" modules: *explicit* because of the metadata definition class, and *application* because they don't come from the JDK.
- Automatic modules: A JAR that is not a modular JAR will be made into an automatic module if placed on the module path. You can add an `Automatic-Module-Name` entry in a JAR's `MANIFEST.MF` file: this information will be used for the module's name. This is the recommended practice. Barring that, by default, its module name will be derived from the JAR's name. If you maintain a library, please add this automatic module name to your manifest.
- Unnamed module: When using the classpath, every JAR that is not modularized will be part of the unnamed module.

A Java 9-modularized application makes use of a `module-info` class file located at each JAR's root. This file is a specific manifest—a module definition—that contains the module name and declares which module dependencies are required, as well as other metadata such as its exposed API, provided services, and so on. At runtime, the loader reads that manifest to load only the modules that are necessary.

To reduce the overall size of an image, you can take advantage of the module system and distribute only the required modules of the JDK.

With this design, it's possible to eliminate unnecessary modules that are part of the JDK, which is the mission of jlink. As the official documentation states, "You can use the jlink tool to assemble and optimize a set of modules and their dependencies into a custom runtime image."

jlink enables you to use the underlying module configuration of an application to deliver a custom JRE along with the application. Using the same mechanics, it also allows you to create an executable out of the application, so the deliverable is completely self-sufficient and doesn't rely on the target system having a compatible JRE.

Laying the Foundation

Let's examine jlink starting with the simplest possible application: Hello World. Here it is in all its glory:

```
//Main.java
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

It is hard to argue that Docker is not the most popular container distribution channel nowadays. To distribute this Hello World application, it would be a huge benefit to use Docker. Because my intention is to both create a single Dockerfile and keep the final image as small as possible, a multistage build is needed.

As a reminder, a multistage Docker build allows you to chain stages in such a way that a later stage can reuse build results from previous stages. In addition, each stage can inherit from different base images and you can name each stage, because it is easier to reference a stage by name than by index. The main benefit of multistage builds is the ability to use the most relevant image in each stage, so you can have the smallest resulting image at the end of the build process.

Here's an example Dockerfile showing how to create an image for Hello World by using Maven. I assume the project has a Maven-compatible structure:

```
Dockerfile
FROM maven:3.6-jdk-12-alpine as build

WORKDIR /app

COPY pom.xml .
COPY src src

RUN mvn package

FROM openjdk:12-alpine

COPY --from=build /app/target/jlink-ground-up-1.0-SI
                  /app/target/jlink-ground-up.jar

ENTRYPOINT ["java", "-jar"]
CMD ["/app/target/jlink-ground-up.jar"]
```

In this file, the second line identifies the first stage of the multistage build, which uses a Maven image and is labeled <`build`. The `mvn package` command generates the JAR, using the default name that is `jlink-ground-up-1.0-SNAPSHOT.jar`.

In the line that begins with `FROM`, you see the second and last stage of the build. That line uses one of the smallest images possible, an Alpine distribution of Linux. There's no Alpine image for Java 11, but there is one for Java 12. Unfortunately, no JRE is available, only a JDK. The `COPY` statement that's next reuses the JAR file that is the output of the first stage.

Then, you create the image:

```
$ docker build -t jlink:1.0 .
```

The main issue with this approach is that the Docker image is huge, because it embeds the whole JDK, even for a Hello World application. In fact, the size of the application is negligible compared to the size of OpenJDK 12:

```
$ docker images
REPOSITORY   TAG          IMAGE ID        CREATED
openjdk      12-alpine    8f180304fad9    7 days ag
jlink        1.0          7c612235f308    About a m
```

To reduce the overall size of an image, you can take advantage of the module system and distribute only the required modules of the JDK.

Distributing a Custom Launcher

The Hello World application is so simple that no module except `java.base` is required. This module is automatically required, just as `java.lang` packages are implicitly imported.

To distribute a custom, smaller executable, the first step is to migrate the application to the module system. As stated earlier, jlink can work only with modularized applications, because it relies on the `module-info` file.

Because this app requires only the `java.base` module, creating a `module-info.java` module descriptor is straightforward:

```
// module-info.java
module ch.frankel.jlink {
    exports ch.frankel.blog.jlink;
}
```

jlink's main feature is to optimize an application to keep only the modules that will be used. Furthermore, it can be used to create a standalone executable out of the optimized version of the application. Because our app is now taking advantage of the module system, it becomes possible to create this dedicated launcher.

However, jlink requires an existing JAR to work its magic:

```
$ mvn clean package
```

Everything is now set, so it's time to use `jlink`. Be aware that like the java or `javac` commands, jlink requires options to be specified. Here's the command to create a custom executable for Hello World:

```
$ jlink --add-modules ch.frankel.blog.jlink \
    --module-path ${JAVA_HOME}/jmods:target/jlink-gi
    --output target/jlink-image \
    --launcher hello=ch.frankel.jlink/ch.frankel.blo
```

Let's examine the options.

- The first line defines the modules that are required via their names. The module's name of our application should be set.
- The second line specifies the module path. While pre-Java 9 applications use the classpath, module-compatible applications use the module path. Just as with the classpath, the module path references path elements to search for dependent modules. For now, the path to every module including those provided by the JDK and the JAR file, must be referenced.
- The third line specifies the output folder.
- The final line specifies the entry point of the custom distribution. Its format consists of several parts: the final executable name, an = sign, the module name, and a / followed by the fully qualified class name of the `Main` class.

Once you have created the distribution, you can launch it using this command:

```
$ target/jlink-image/bin/hello
```

As expected, this command prints `Hello world` to standard out.

The good news is that there's an existing Maven plugin to help you declaratively manage the module path: the ModiTect plugin.

Just as before, the goal is to wrap this custom distribution in a Docker image. Let's adapt the Dockerfile accordingly, as follows:

```
FROM maven:3.6-jdk-12-alpine as build

WORKDIR /app
COPY pom.xml .
COPY src src
RUN mvn package && \
    jlink --add-modules ch.frankel.jlink \
     --module-path ${JAVA_HOME}/jmods:target/jlink-
     --output target/jlink-image \
     --launcher hello=ch.frankel.jlink/ch.frankel.bl

FROM alpine:3.8

COPY --from=build /app/target/jlink-image /app
ENTRYPOINT ["/app/bin/hello"]
```

The next question is whether this extra step had any effect on the size of the resulting Docker image. Let's compare it with the previous build that was created without jlink:

```
REPOSITORY     TAG    IMAGE ID         CREATED
jlink          1.0    7c612235f308     About a minut
jlink          1.1    e590fb6697e7     14 hours ago
```

Wow; that's a whopping 283 MB savings from the previous unmodularized application! If 53 MB seems a bit much for a Hello World application, remember that the distribution contains all the power of the JVM, including the just-in-time (JIT) compiler and garbage collection management.If necessary, pause to savor this victory, and when you are ready, proceed to the next section.

Industrializing the Generation of the Custom Launcher

The previous jlink command is verbose. Moreover, it will become more verbose when the number of module dependencies increases. The situation is quite similar to that of the Java compiler: In day-to-day life, developers rarely invoke the compiler directly; instead they favor the use of a build tool such as Maven. With Maven, dependencies are declared in the POM file, and the Java compiler plugin manages the classpath for you. It would be very awkward to manage the classpath manually at every compilation.

The good news is that there's an existing Maven plugin to help you declaratively manage the module path: the ModiTect plugin. I use this plugin in the rest of this article.

Among other features, the plugin offers a `create-runtime-image` goal that creates the launcher. Here is a POM snippet that creates the custom launcher as described earlier but in a repeatable way:

```
<plugin>
  <groupId>org.moditect</groupId>
  <artifactId>moditect-maven-plugin</artifactId>
  <version>1.0.0.Beta2</version>
  <executions>
    <execution>
      <id>create-runtime-image</id>
      <phase>package</phase>    <!-- comment 1 -->
      <goals>
        <goal>create-runtime-image</goal>  <!-- comm
      </goals>
      <configuration>
        <modulePath>            <!-- comment 3 -->
          <path>                <!-- comment 4 -->
            ${project.build.directory}/${project.art
            ${project.version}.${project.packaging}
          </path>
        </modulePath>
        <modules>
          <module>ch.frankel.jlink</module>   <!-- c
        </modules>
        <launcher>
          <name>hello</name>                  <!-- c
          <module>
            ch.frankel.jlink/ch.frankel.blog.jlink.M
          </module>                           <!-- c
        </launcher>
        <outputDirectory>
            ${project.build.directory}/jlink-image
        </outputDirectory>                    <!-- c
      </configuration>
    </execution>
  </executions>
</plugin>
```

The line that contains `comment 1` binds execution to the package phase. The line that contains `comment 2` calls the `create-runtime-image goal`. The lines that contain `comment 3` will

be translated to a jlink command-line option. And the line that contains `comment 4` is followed by two lines that you should enter as a single line (including the hyphen); the two lines are shown separately so they fit on the page.

Table 1 shows how the declarative configuration maps to the jlink command-line options:

XML	JLINK OPTION
<modulePath>	--module-path
<modules>	--add-modules
<name>	FIRST PART OF --launcher
<module>	SECOND PART OF --launcher
<outputDirectory>	--output

Table 1. How the configuration maps to the jlink options

With that information added to the POM, the Dockerfile can be simplified further:

```
Dockerfile
FROM maven:3.6-jdk-12-alpine as build

WORKDIR /app
COPY pom.xml .
COPY src src
RUN mvn package

FROM alpine:3.8

COPY --from=build /app/target/jlink-image /app
ENTRYPOINT ["/app/bin/hello"]
```

At this point, you have a repeatable build process for creating custom distributions.

Adding Module Dependencies

Let's beef up the application and improve the code. As you probably know, it's unwise to use `System.out.println()` statements: they are not configurable, so if they are used for debugging purposes, they will be written even in production. Let's replace this log statement with a call to a proper logging framework.

As my logging framework, I will use the Simple Logging Facade for Java (SLF4J), which is modularized. This choice requires me to add two dependencies: the API and a single implementation.

To do that, add the following to the POM file:

```
<dependencies>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.8.0-beta2</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
        <version>1.8.0-beta2</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

And update the `module-info.java` file:

```
module ch.frankel.jlink {
    requires org.slf4j;
```

```
    exports ch.frankel.blog.jlink;
}
```

To use jlink, you also need to add SLF4J to the `--module-path`, as follows:

```
<configuration>
  <modulePath>
    <path>  <!--enter the next two lines as a single
      ${project.build.directory}/${project.artifact
      ${project.version}.${project.packaging}
    </path>
    <path> <!--enter the next two lines as a single
      ${settings.localRepository}/org/slf4j/slf4j-ap
        1.8.0-beta2/slf4j-api-1.8.0-beta2.jar
    </path>
  </modulePath>
</configuration>
```

As the number of dependencies grows, this approach can quickly become tedious. It's easy to forget a dependency, and it's cumbersome to upgrade the version number in the `<dependencies>` section and here as well.

A better alternative is to copy every dependency into a dedicated directory, and use that directory as a part of the module path. The following code shows how you can configure the build process in the POM file to automatically do that during each run:

```
<plugin>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>3.1.1</version>
  <executions>
    <execution>
      <id>copy</id>
      <phase>package</phase>
      <goals>
        <goal>copy-dependencies</goal>
      </goals>
      <configuration>
        <outputDirectory>
          ${project.build.directory}/modules
        </outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Note that you can configure the module path once and for all, as follows, because every dependency will be copied to the `modules` folder:

```
<modulePath>
  <path> <!--enter the next two lines as a single li
   ${project.build.directory}/${project.artifactId}-
   ${project.version}.${project.packaging}
  </path>
  <path>${project.build.directory}/modules</path>
</modulePath>
```

Adding Nonmodule Dependencies

Unfortunately, jlink works only with modules and it will fail if not all dependencies are modules. The only way to fix that is to craft a custom `module-info.java` file, compile it, and update the dependent JAR with it. You can use the jdeps utility, which is part of the JDK, to determine the contents of the module file, namely the dependencies that need to be declared.

Here's an example usage:

```
$ jdeps --generate-module-info \
        . \
        $M2_REPO/org.apache…/commons-lang3-3.8.1.ja
```

The first line specifies that jdeps should generate the `module-info.java` file, the second line is the output directory, and the third line is the target JAR file to be analyzed. [The path was shortened to fit the page. —*Ed.*]

This command generates the following file:

```
module org.apache.commons.lang3 {
    requires transitive java.desktop;

    exports org.apache.commons.lang3;
    exports org.apache.commons.lang3.arch;
    exports org.apache.commons.lang3.builder;
    exports org.apache.commons.lang3.concurrent;
    exports org.apache.commons.lang3.event;
    exports org.apache.commons.lang3.exception;
    exports org.apache.commons.lang3.math;
    exports org.apache.commons.lang3.mutable;
    exports org.apache.commons.lang3.reflect;
    exports org.apache.commons.lang3.text;
    exports org.apache.commons.lang3.text.translate
    exports org.apache.commons.lang3.time;
    exports org.apache.commons.lang3.tuple;
}
```

Note that the folder's name is not randomly chosen: It's taken from the `Automatic-Module-Name` attribute in the JAR's manifest. This design allows a JAR to have a stable module name. If the attribute is missing, the module system will automatically infer a module name based on the JAR's name, which might not be suitable.

jdeps doesn't handle the compilation, but the ModiTect plugin provides a goal for achieving that. Let's update the POM file accordingly:

```
<plugin>
  <groupId>org.moditect</groupId>
  <artifactId>moditect-maven-plugin</artifactId>
  <version>1.0.0.Beta2</version>
  <executions>
    <execution>
      <id>add-module-info</id>
      <phase>package</phase>          <!-- comment 1
      <goals>
        <goal>add-module-info</goal>  <!-- comment 2
      </goals>
      <configuration>
        <outputDirectory>
          ${project.build.directory}/modules
        </outputDirectory>
        <modules>
          <module>
            <artifact>                <!-- comment 3
              <groupId>org.apache.commons</groupId>
              <artifactId>commons-lang3</artifactId>
            </artifact>
            <moduleInfo>              <!-- comment 4
              <name>org.apache.commons.lang3</name>
            </moduleInfo>
          </module>
        </modules>
        <overwriteExistingFiles>true</overwriteExist
      </configuration>
    </execution>
</plugin>
```

In this file, the line with `comment 1` binds plugin execution to the package phase; the line with `comment 2` calls the `add-module-info` goal; the section that starts with the line that has `comment 3` specifies the target dependency to update; and the section that starts with the line

that has `comment 4` shows that in the additional module information section, only the name is required.

The build process now modularizes the `commons-lang3` dependency. This snippet is compatible with the previous snippet, so every dependency is copied to the `modules` folder, and it will be overwritten by the modularized JAR if it's not a module already.

The `commons-lang3` dependency is simple in two regards: It has an `Automatic-Module-Name` in its manifest and it has no external dependencies. For that reason, it's pretty easy to make use of it.

If you were to replace `commons-lang3` with the Guava library, for example, you would simply change the library name in the plugin. However, jdeps would explore the whole dependency tree and all of Guava's dependencies would need to be modularized as well, just as Guava itself would. This configuration would be quite verbose, but unfortunately it would be necessary. I've posted a copy of the pom.xml file.

Conclusion

In this article, I used jlink to create a custom launcher for a simple application. As we saw, jlink enables you to create launchers that contain only the required modules. To use jlink, an application itself must be modularized.

At that point, the complexity of the process depends on the compatibility of dependencies regarding modularization. If dependencies are modules themselves, all is straightforward. If not, they need to be transformed into modules before going further. Fortunately, the ModiTect plugin offers such a feature.

I hope that this article will help you to create smaller distributions of your apps, suitable for containerization.

Also in This Issue

Nicolas Fränkel

Nicolas Fränkel (@nicolas_frankel) has many years of experience consulting for various customers in a wide range of contexts. Usually he works with the Java/Java EE and Spring technologies, and he also focuses on interests such as rich internet applications, testing, continuous integration and continuous delivery, and DevOps. He currently works for Exoscale and has authored several books on programming and testing.

Share this Page