Dockerize Your Spring Boot Application With Hot-Reloading

Kishieel

.

Follow

7 min read

.

Jan 28, 2023

--

1

Listen

Share

Photo by Dominik Lückmann on Unsplash

Recently, I had the amazing opportunity to join a new project based on Spring Boot in a microservices architecture. As the project was in its early stages of development and many of the people working on it were just starting to learn the current technology stack, I decided to simplify the steps needed to run the application in a local environment as well as its later deployment. I decided to have each service run in a separate Docker container and use Docker Compose to connect them all together.

Drawing on my experience working with Node.js, I wanted to ensure that services started in the locally-running containers also had the functionality of hot-reloading, where the source files of a particular service are updated and the service automatically reloads. As I delved deeper into this idea, I encountered a few challenges that required some troubleshooting. But, after successfully achieving my goal, I decided to share my experience here, in order to help others who may be facing similar challenges.

A School Of Patience And Eternal Waiting For Dependencies

One of the frustrating things I encountered was the constant waiting for dependencies to download every time I wanted to rebuild the container. The solution to this problem is known and relatively simple: you just need to split the `Dockerfile` into two stages. The first stage is responsible for only resolving dependencies and acts as a kind of cache, while the second stage is responsible for building the JAR. This way, when the source code changes but the `pom.xml` file remains unchanged, we don't have to wait for the dependencies to be downloaded again. Simple, right?

```
FROM maven:3-eclipse-temurin-17-alpine AS deps

WORKDIR /app
COPY pom.xml /app

RUN mvn dependency:go-offline


FROM maven:3-eclipse-temurin-17-alpine AS build

WORKDIR /app
COPY --from=deps /root/.m2/repository /root/.m2/repository
COPY . /app

RUN mvn package -DskipTests -o
```

Well… not quite. I quickly realized that the command `mvn dependency:go-offline` doesn't handle resolving all dependencies well, and the stage of building the JAR often ended in errors like the one mentioned below.

After digging through a lot of the internet, I realized that I wasn't the only one experiencing this problem. The solution turned out to be the qaware/go-offline-maven-plugin, which handles this task much better. To resolve the first problem encountered, I added this plugin to the `pom.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
<!-- other lines removed for better readability -->
```

```
<properties>
 <!-- other lines removed for better readability -->
 <go-offline-maven-plugin.version>1.2.8</go-offline-maven-plugin.version>
</properties>

<build>
 <plugins>
  <!-- other lines removed for better readability -->
  <plugin>
   <groupId>de.qaware.maven</groupId>
   <artifactId>go-offline-maven-plugin</artifactId>
   <version>${go-offline-maven-plugin.version}</version>
  </plugin>
 </plugins>
</build>
</project>
```

And I replaced the command `mvn dependecy:go-offline`in the `Dockerfile` with `mvn go-offline:resolve-dependencies` . The improved version looks like the one below.

```
FROM maven:3-eclipse-temurin-17-alpine AS deps

WORKDIR /app
COPY pom.xml /app

RUN mvn go-offline:resolve-dependencies


FROM maven:3-eclipse-temurin-17-alpine AS build

WORKDIR /app
COPY --from=deps /root/.m2/repository /root/.m2/repository
COPY . /app

RUN mvn package -DskipTests -o
```

Hot-Reloading: Because Nobody Likes Waiting For Containers To Rebuild

Now that the dependencies have been properly cached and the JAR file built, I would like the development of the application to proceed without further interaction with Docker and for the application to automatically rebuild without the need for a container restart when the source code changes.

In order to achieve this, we will need the `spring-boot-devtools` package, which will allow for automatic reloading of updated classes without the need for a full restart. The package is added to dependencies as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
<!-- other lines removed for better readability -->

<properties>
 <!-- other lines removed for better readability -->
 <spring-boot-devtools.version>3.0.1</spring-boot-devtools.version>
</properties>

<dependencies>
 <!-- other lines removed for better readability -->
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <version>${spring-boot-devtools.version}</version>
  <optional>true</optional>
 </dependency>
</dependencies>
</project>
```

With this package, we can run the command `mvn spring-boot:run`, which will automatically reset the application as soon as the JAR file changes. However, the command alone is not enough, as something also needs to compile the application as the source code changes.

To achieve this, we will use the inotify tool, which we will install in a separate stage of the `Dockerfile`, and then use in a script that will be run when the container starts. Inotify is a Linux kernel subsystem that acts as an event notification system, allowing applications to monitor changes to the file system. It can be used to detect changes to files or directories and respond accordingly, such as automatically reloading a program when its source code is modified.

The stage in the `Dockerfile` that we will use for local development will look like the following.

```
FROM maven:3-eclipse-temurin-17-alpine AS dev

WORKDIR /app
COPY --from=deps /root/.m2/repository /root/.m2/repository
COPY ./docker-entrypoint.sh /docker-entrypoint.sh

RUN apk add inotify-tools
RUN chmod +x /docker-entrypoint.sh

ENTRYPOINT ["/docker-entrypoint.sh"]
```

The script `docker-entrypoint.sh`, will consist of a background program that listens for changes in the /app/src/main directory and automatically recompiles the code whenever it is changed, as well as a command that runs the Spring Boot application.

```
#!/bin/sh

while inotifywait -r -e modify /app/src/main/;
do
 mvn compile -o -DskipTests;
done >/dev/null 2>&1 &

mvn spring-boot:run
```

Now that we have our container ready, we can test if it meets our goal of a self-restarting application. To do this, we can use the following commands:

```
docker build . --tag kishieel/spring-hot-reloading --target dev
docker run --rm -it -v $(pwd):/app -p 8080:8080 kishieel/spring-hot-reloading
```

The first command will build the container with the `dev` target and tag it with the name `kishieel/spring-hot-reloading`. The second command will start the container and map the host's current directory to the container's `/app` directory (this is necessary if we want to rebuild the application inside the container instead of on the host). It also maps the host's port `8080` to the container's port `8080`, allowing the application to be accessed via `localhost:8080`.

Once the container is running, you can check the functionality by using the curl command `curl http://localhost:8080/example -w "\n"`. If the response is correct, you can replace "Hello World!" with "Hello Hot Reloading!" in the `Application.java` file and, without doing anything more, call another curl to the same endpoint. If everything works fine, the change will be reflected in the output without having to manually rebuild the container.

But Wait! Are We Production Ready?

Well… not really. While the `Dockerfile` stage we created earlier is great for local development, it may not be the best option for production servers. One out of the many issues with the current `Dockerfile` is that it uses the root user to run the application. This can present a security risk as it allows the application to access the host system with full privileges. Additionally, the alpine image we are using is not as small as it could be, which can lead to longer download and startup times.

To address these issues, we can create a new stage in our `Dockerfile` specifically for production use. One common practice is to use a multi-stage build, where we first build the application in one stage and then copy the compiled artifacts to a smaller, runtime-only image in another stage. In this example, we will use the `eclipse-temurin:17.0.5_8-jdk-alpine` image as our runtime image.

In the `release` stage, we copy the built JAR file from the previous step. We also create a new user with the necessary privileges to run the application and start it. This time, we do this directly with Java, as Maven is no longer available in the current image.

```
FROM eclipse-temurin:17.0.5_8-jdk-alpine AS release

LABEL maintainer="kishieel"
WORKDIR /app

COPY --from=build /app/target/app.jar /app/app.jar

RUN addgroup --system app && adduser -S -s /bin/false -G app app
RUN chown -R app:app /app

USER app
CMD ["java", "-jar", "app.jar"]
```

To build this stage and start container, we can use the following commands:

```
docker build . --tag kishieel/spring-hot-reloading --target release
docker run --rm -it -p 8080:8080 kishieel/spring-hot-reloading
```

To ensure that the application is running properly, we can use the curl command to send a request to the endpoint, as we did previously. However, it's important to note that hot-reloading is not available in this version of the container, so we're simply testing if the application has started correctly.

This new stage addresses the security and size concerns we had with the previous stage, and is a more suitable option for production servers. However, it should be noted that this is not an exhaustive list of best practices for building Java containers with Docker, and further research should be done to ensure that your container is as secure and efficient as possible.

Summary

In this blog post, we discussed the process of creating a `Dockerfile` for a Spring Boot application with hot-reloading capabilities. We went through the challenges of caching dependencies and building JAR, as well as implementing hot-reloading within the container. Finally, we touched on the importance of ensuring that the image is production-ready before deployment.

I hope that this post has been helpful in guiding you through the process of creating a `Dockerfile` for your Spring Boot application with hot-reloading feature. If you're interested in viewing the complete version of the code, you can find it on my GitHub repository [link here]. I appreciate you taking the time to read through this post. I welcome any feedback and questions you may have.