Dockerizing Spring Boot: Best Practices for Efficient Containerization

kiarash shamaii

.

Follow

9 min read

.

Jan 17, 2024

--

5

Listen

Share

First of all why use Docker, Docker is a powerful tool that allows developers to package their applications into containers that can be easily deployed and run on any platform. When it comes to Dockerizing a Spring Boot application, there are some best practices that every developer should follow to ensure the application runs smoothly and efficiently. In this article, we will explore these best practices and provide code examples and explanations to help you *Dockerize* your Spring Boot application.

There are a lot of base official images for Dockerizing spring-boot application as a java developer we need focus on the size of image specially when project get big.

Use the right base image

When Dockerizing a Spring Boot application, it's important to choose the right base image for your application. As you maybe know all images in Docker have base layer of Linux kernel so we don't need add this part to our image, so our base image provides the underlying kernel and dependencies required by your application. Choosing the right base image can help ensure that your application runs smoothly and efficiently in a Docker container.

For a Spring Boot application, Must of document recommend using an OpenJDK base image. OpenJDK is an open-source implementation of the Java Development Kit (JDK) and provides a Java runtime environment and Java Development tools . OpenJDK base images are available in different versions, such as Java 8, Java 11, and Java 16. Here's an example Dockerfile that uses an OpenJDK 11 base image :

```
FROM openjdk:17-jdk-slim
COPY target/springBootDockerized-0.0.1-SNAPSHOT.jar springBootDockerized-0.0.1-SNAPSHOT.jar
ENTRYPOINT ["java" , "-jar" , "/springBootDockerized-0.0.1-SNAPSHOT.jar"]
```

But it is very important we don't need JDK we just need JRE java runtime environment I recommend use JRE layers in link for OpenJDK official you could find this :

```
eclipse-temurin
```

As a sample spring-boot application add a Dockerfile to root something like this :

```
#bad practice for dockerized use JDK
#FROM openjdk:17-jdk-slim
#COPY target/springBootDockerized-0.0.1-SNAPSHOT.jar springBootDockerized-0.0.1-SNAPSHOT.jar
#ENTRYPOINT ["java" , "-jar" , "/springBootDockerized-0.0.1-SNAPSHOT.jar"]


FROM eclipse-temurin:17.0.5_8-jre-focal as builder
WORKDIR extracted
ADD ./target/*.jar app.jar
RUN java -Djarmode=layertools -jar app.jar extract

FROM eclipse-temurin:17.0.5_8-jre-focal
WORKDIR application
COPY --from=builder extracted/dependencies/ ./
COPY --from=builder extracted/spring-boot-looder/ ./
COPY --from=builder extracted/snapshot-dependencies/ ./
COPY --from=builder extracted/application/ ./
```

```
EXPOSE 8085

#use it before springboot 3.2
#ENTRYPOINT ["java" , "org.springframework.boot.loader.JarLauncher"]
#use it for after springboot 3.2

ENTRYPOINT ["java" , "org.springframework.boot.loader.launch.JarLauncher"]
```

In this example, the first stage uses a Maven base image to build the Spring Boot application and generate a jar file. The second stage uses an OpenJDK slim base image, which is a smaller version of the base image that only includes the Java runtime environment.

The `COPY --from=build` instruction copies the jar file from the first stage to the second stage, and the `ENTRYPOINT` instruction specifies the command that should be run when the container is started.

the last line of first part docker explain :

- `java`: This is the command to run Java applications or execute Java bytecode.
- `-Djarmode=layertools`: This is a system property specified using the `-D` flag. It sets the value of the `jarmode` property to `layertools`. This is another way to enable the "layertools" mode for manipulating layers within a modular JAR file.
- `-jar app.jar`: This specifies that the JAR file to be executed is named "app.jar". The `-jar` option indicates that the specified file is an executable JAR file.
- `extract`: This is an argument or command passed to the application within the JAR file. It instructs the application to perform a specific action, which, in this case, is to extract the contents of the JAR file.

Using multi-stage builds in this way allows us to create a slim Docker image that only includes the required dependencies and files for running the Spring Boot application. By doing so, we can reduce the size of the image and improve the performance of the application**.**

an other way use Build-pack.io which make image automatically for you in your pom add this in plugin tag :

```
    <build>
       <plugins>
           <plugin>
               <configuration>
                   <image>
                       <name>kia/test</name>
                   </image>
               </configuration>
               <groupId>org.springframework.boot</groupId>
               <artifactId>spring-boot-maven-plugin</artifactId>
           </plugin>
       </plugins>
   </build>
```

after that run this command :

```
mvn spring-boot:build-image
```

with this command spring boot make image for you perfectly.

Use environment variables

When Dockerizing a Spring Boot application, it’s important to use environment variables to configure your application. Using environment variables allows you to change the configuration of your application without having to rebuild the Docker image.

Spring Boot applications can use the `application.properties` or `application.yml` file to specify configuration properties. These properties can be overridden at runtime using environment variables, which Spring Boot automatically maps to properties. Here’s an example Dockerfile that sets an environment variable to configure the active profile for the Spring Boot application:

```
FROM openjdk:11
ENV SPRING_PROFILES_ACTIVE=production
COPY target/my-application.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

In this example, we’re setting the `SPRING_PROFILES_ACTIVE` environment variable to `production`, which will activate the `production` profile in the Spring Boot application.

When the container is started, the `java` command specified in the `ENTRYPOINT` instruction is run with the `-jar` option to start the Spring Boot application. Since we've set the `SPRING_PROFILES_ACTIVE` environment variable, the application will automatically use the `production` profile.

Use health checks

When Dockerizing a Spring Boot application, it's important to use health checks to *monitor* the health of your application and ensure that it's running correctly. Health checks can be used to detect when your application is unhealthy and automatically perform *recovery* or *scaling* based on the health of the application.

To add a health check to your Docker image, you can use the `HEALTHCHECK` instruction in your Dockerfile. The `HEALTHCHECK` instruction tells Docker how to check the health of your application. Here's an example Dockerfile that adds a health check to a Spring Boot application:

```
FROM openjdk:11
COPY target/my-application.jar app.jar
HEALTHCHECK --interval=5s \
           --timeout=3s \
           CMD curl -f http://localhost:8080/actuator/health || exit 1
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

In this example, we're using the `HEALTHCHECK` instruction to check the health of the Spring Boot application. The `--interval` option specifies how often the health check should run, and the `--timeout` option specifies how long to wait for a response. The `CMD` instruction runs the health check command, which is a `curl` command that checks the `/actuator/health` endpoint of the application.

When you run the container, you can use the `docker ps` command to view the health status of the container:

```
$ docker ps
CONTAINER ID   IMAGE                COMMAND                  CREATED         STATUS          PORTS                     NAMES
e8e1a6440e5e   my-application:1.0   "java -jar /app.jar"     5 seconds ago   Up 4 seconds    0.0.0.0:8080->8080/tcp   my-appli
$ docker inspect --format='{{json .State.Health}}' my-application
{"Status":"healthy","FailingStreak":0,"Log":[{"Start":"2023-03-25T09:21:08.272130387Z","End":"2023-03-25T09:21:08.310105965Z",
```

In this example, the `docker ps` command shows that the container is up and running on port `8080`. The `docker inspect` command shows the health status of the container, which is currently `healthy`. If the health check fails, the container will be marked as `unhealthy`, and you can use tools like Docker Compose or Kubernetes to automatically recover or scale the container.

Use Docker caching

When Dockerizing a Spring Boot application, it's important to use Docker caching to speed up the build process and reduce the time it takes to build a new Docker image. *Docker caching allows you to reuse previously built layers of your Docker image, avoiding the need to rebuild those layers each time you build a new image*. Here's an example Dockerfile that uses Docker caching to speed up the build process:

```
FROM openjdk:11 as builder
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline

COPY src/ ./src/
RUN mvn package -DskipTests

FROM openjdk:11
COPY --from=builder /app/target/my-application.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

each step assume a stage of layers which cache in docker registry,

In this example, we're using a multi-stage build to first build the Spring Boot application in a separate layer, and then copy the built jar file into the final image. By using a separate layer for the build process, we can take advantage of Docker caching to avoid rebuilding the dependencies each time we build a new image.

The first stage of the build process uses the `openjdk:11` base image and copies the `pom.xml` file to the container. It then runs the `mvn dependency:go-offline` command to download all the dependencies required by the application. This command ensures that all the required dependencies are available locally, which will speed up the subsequent builds.

The second stage of the build process uses the `openjdk:11` base image and copies the source code to the container. It then runs the `mvn package` command to build the application jar file. Since we have already downloaded the dependencies in the previous stage, Docker will use the cached layer and skip the dependency download step.

Finally, the `COPY --from=builder` instruction copies the built jar file from the builder stage to the final image, and the `ENTRYPOINT` instruction specifies the command that should be run when the container is started.

Use a .dockerignore file

When Dockerizing a Spring Boot application, it's important to use a `.dockerignore` file to *exclude* unnecessary files and directories from the Docker build context. The build context is the set of files and directories used by Docker to build the Docker image. By using a `.dockerignore` file, you can exclude files and

directories that are not required by the Docker image, reducing the size of the build context and improving the build performance. Here's an example `.dockerignore` file for a Spring Boot application:

```
# Ignore all files in the root directory
*
# Include the src directory
!src/
# Include the pom.xml file
!pom.xml
# Exclude the target directory and its contents
target/
```

In this example, we're using the `.dockerignore` file to exclude all files in the root directory (`*`), except for the `src/` directory and the `pom.xml` file, which are required for building the Spring Boot application. We're also excluding the `target/` directory, which contains the built artifacts and is not required by the Docker image.

By using a `.dockerignore` file, we can reduce the *size* of the build context and improve the build performance. Docker will only copy the files and directories that are included in the build context, and will ignore the files and directories that are excluded in the `.dockerignore` file.

Using a `.dockerignore` file is a good practice for Dockerizing a Spring Boot application, as it helps ensure that the Docker image is built as efficiently and quickly as possible.

Furthermore, using a `.dockerignore` file can also help improve the *security* of your Docker image. By excluding unnecessary files and directories, you can reduce the attack surface of your Docker image and minimize the risk of exposing sensitive information or credentials. For example, if you have configuration files or credentials stored in the build directory, excluding them in the `.dockerignore` file will prevent them from being included in the Docker image.

It's also worth noting that the `.dockerignore` file follows a similar syntax to the `.gitignore` file, which is used to exclude files and directories from Git repositories. If you're familiar with the `.gitignore` file, you should find the `.dockerignore` file easy to use.

In summary, using a `.dockerignore` file is a good practice for Dockerizing a Spring Boot application. It can help reduce the size of the build context, improve the build performance, and improve the security of your Docker image.

Use Labels

When Dockerizing a Spring Boot application, it's important to use labels to add *metadata* to your Docker image. Labels are key-value pairs that can be added to a Docker image to provide additional information about the image, such as the version, maintainer, or build date. Here's an example Dockerfile that uses labels to add metadata to a Spring Boot application:

```
FROM openjdk:11
LABEL maintainer="John Doe <john.doe@example.com>"
LABEL version="1.0"
LABEL description="My Spring Boot application"
COPY target/my-application.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

In this example, we're using the `LABEL` instruction to add metadata to the Docker image. We've added labels for the maintainer, version, and description of the image. These labels provide additional information about the Docker image and help users understand what the image contains and how it was built.

You can view the labels of a Docker image using the `docker inspect` command:

```
$ docker inspect my-application
[
   {
       "Id": "sha256:...",
       "RepoTags": [
           "my-application:latest"
       ],
       "Labels": {
           "maintainer": "John Doe <john.doe@example.com>",
           "version": "1.0",
           "description": "My Spring Boot application"
       },
       ...
   }
]
```

In this example, the `docker inspect` command shows the labels of the `my-application` Docker image. The labels provide additional information about the image and can help users understand how the image was built and how to use it.

Using labels in this way can help improve the usability and maintainability of your Docker image. By adding metadata to your Docker image, you can help users understand what the image contains and how it was built. This information can be useful for debugging, troubleshooting, and maintaining the Docker image over

time.