**Dockerizing Your Spring Boot Application**

10 Best Practices with Code Examples



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.

**Use the right base image**

When Dockerizing a Spring Boot application, it’s important to choose the right base image for your application. The base image provides the underlying operating system 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, we recommend using an OpenJDK base image. OpenJDK is an open-source implementation of the Java Development Kit (JDK) and provides a Java runtime environment. 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:11  
COPY target/my-application.jar app.jar  
ENTRYPOINT ["java", "-jar", "/app.jar"]

In this example, we’re using the openjdk:11 base image to create a Docker image for our Spring Boot application. We're copying the my-application.jar file to the container and using the java command to run the application.

*Using the right base image for your Spring Boot application can help ensure that your application runs smoothly and efficiently in a Docker container. OpenJDK is a popular choice for Java applications, as it provides a lightweight and secure Java runtime environment.*

**Build a slim image**

When Dockerizing a Spring Boot application, it’s important to keep the size of the Docker image as small as possible. A smaller image size has several advantages, such as faster image transfer times, lower storage requirements, and faster container startup times.

One way to achieve a smaller image size is by using *multi-stage builds* in your Dockerfile. In a multi-stage build, you use multiple FROM instructions to define different stages in the build process. Each stage can have its own set of instructions and dependencies, and the final image only includes the files and dependencies from the last stage. Here’s an example Dockerfile that uses multi-stage builds to create a slim Spring Boot image:

# First stage: build the application  
FROM maven:3.8.3-jdk-11 AS build  
COPY . /app  
WORKDIR /app  
RUN mvn package -DskipTests  
  
# Second stage: create a slim image  
FROM openjdk:11-jre-slim  
COPY --from=build /app/target/my-application.jar /app.jar  
ENTRYPOINT ["java", "-jar", "/app.jar"]

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.

*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.*

**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.

*Using environment variables in this way makes it easy to change the configuration of your Spring Boot application without having to rebuild the Docker image. You can set environment variables using the -e option when running the container, or by using a Docker Compose file to define the environment variables.*

**Use Docker Compose**

When Dockerizing a Spring Boot application, it’s important to use Docker Compose to define your application’s *services*and *dependencies*. Docker Compose is a tool for defining and running multi-container Docker applications. It allows you to define your application’s *services*, *networks*, and *volumes*in a single file, making it easy to manage and deploy your application. Here’s an example Docker Compose file that defines a Spring Boot application and a MySQL database:

version: '3'  
services:  
 db:  
 image: mysql:5.7  
 environment:  
 MYSQL\_ROOT\_PASSWORD: my-secret-pw  
 MYSQL\_DATABASE: my-database  
 volumes:  
 - db\_data:/var/lib/mysql  
 web:  
 build: .  
 ports:  
 - "8080:8080"  
 environment:  
 SPRING\_DATASOURCE\_URL: jdbc:mysql://db:3306/my-database  
 SPRING\_DATASOURCE\_USERNAME: root  
 SPRING\_DATASOURCE\_PASSWORD: my-secret-pw  
volumes:  
 db\_data:

In this example, we have defined two services: db and web. The db service uses the official MySQL image and sets the root password and database name using environment variables. It also creates a named volume db\_data for persistent storage.

The web service builds the Spring Boot application using the . build context and exposes port 8080. It also sets environment variables for the database URL, username and password, which are used by the Spring Boot application to connect to the MySQL database.

*Using Docker Compose in this way allows you to easily manage and deploy your Spring Boot application and its dependencies. You can start all the services defined in the Docker Compose file with a single command, and you can scale the services up or down depending on the demand. Additionally, you can use Docker Compose to define additional configuration options such as volumes, networks and environment variables, making it easy to manage and deploy your application.*

**Use a reverse proxy**

When Dockerizing a Spring Boot application, it’s important to use a *reverse proxy* to handle incoming traffic and distribute it to your application’s containers. *A reverse proxy is a server that sits between your application and the internet, and forwards requests to your application’s containers based on certain rules.*

Using a reverse proxy has several advantages, such as *load balancing*, *SSL termination*, and *improved security*. By using a reverse proxy, you can distribute incoming traffic evenly across multiple containers, terminate SSL connections at the proxy level to reduce the load on your application’s containers, and add an additional layer of security to your application. Here’s an example Docker Compose file that defines a Spring Boot application and an Nginx reverse proxy:

version: '3'  
services:  
 web:  
 build: .  
 environment:  
 SPRING\_PROFILES\_ACTIVE: production  
 ports:  
 - "8080:8080"  
 proxy:  
 image: nginx  
 ports:  
 - "80:80"  
 volumes:  
 - ./nginx.conf:/etc/nginx/nginx.conf:ro  
 depends\_on:  
 - web

In this example, we have defined two services: web and proxy. The web service builds the Spring Boot application and exposes port 8080. The proxy service uses the official Nginx image and forwards requests to the web service based on the rules defined in the nginx.conf file.

Here’s an example nginx.conf file that defines the rules for forwarding requests to the web service:

events {  
}  
  
http {  
 server {  
 listen 80;  
  
 location / {  
 proxy\_pass http://web:8080;  
 proxy\_set\_header Host $host;  
 proxy\_set\_header X-Real-IP $remote\_addr;  
 proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;  
 proxy\_set\_header X-Forwarded-Proto $scheme;  
 }  
 }  
}

In this example, we’re using the proxy\_pass directive to forward requests to the web service on port 8080. We're also setting various headers to preserve the original client IP and protocol information.

*Using a reverse proxy in this way can help improve the scalability, security, and performance of your Dockerized Spring Boot application. By using a reverse proxy, you can easily distribute incoming traffic across multiple containers, add an additional layer of security to your application, and reduce the load on your application’s containers by terminating SSL connections at the proxy level.*

**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-application  
$ 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","ExitCode":0,"Output":"\n"}]}

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.

*Using health checks in this way can help improve the reliability and availability of your Dockerized Spring Boot application. By using health checks, you can automatically detect and recover from issues in your application, ensuring that your application is always available to your users.*

**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"]

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.

*Using Docker caching in this way can help reduce the time it takes to build a new Docker image and speed up the deployment process. By taking advantage of Docker caching, you can avoid unnecessary rebuilds and ensure that your Docker images are built as quickly and efficiently as possible.*

**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.*

**Use container orchestration**

When Dockerizing a Spring Boot application, it’s important to use container orchestration tools to *manage*and *scale*your application in a production environment. Container orchestration tools can help you *automate the deployment*, *scaling*, and *management*of your Docker containers, making it easier to manage a large number of containers in a distributed environment.

Some popular container orchestration tools for Docker include *Kubernetes*, *Docker Swarm*, and *Apache Mesos*. These tools provide features such as load balancing, automatic scaling, service discovery, and rolling updates, which can help ensure that your application is available and responsive to your users. Here’s an example Kubernetes deployment file for a Spring Boot application:

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: my-application  
 labels:  
 app: my-application  
spec:  
 replicas: 3  
 selector:  
 matchLabels:  
 app: my-application  
 template:  
 metadata:  
 labels:  
 app: my-application  
 spec:  
 containers:  
 - name: my-application  
 image: my-registry/my-application:1.0  
 ports:  
 - containerPort: 8080

In this example, we’re using a Kubernetes deployment file to deploy a Spring Boot application. The deployment file specifies that we want to run three replicas of the application, and uses a selector to identify the pods that should be part of the deployment. The deployment file also specifies the container image that should be used to run the application, and the port that the application should listen on.

*Using container orchestration tools in this way can help improve the scalability, reliability, and availability of your Dockerized Spring Boot application. By using container orchestration tools, you can easily manage and scale your application in a distributed environment, making it easier to ensure that your application is available and responsive to your users.*

**Conclusion**