Containerizing Java app + MySQL (monolithic)

Ways of creating docker image:

- Dockerfile
- Buildpack plugin
- Jibs
- And many more…

We'll focus on the Dockerfile and Buildpack plugin methods for containerizing a Java application.

Note: Replace placeholder values in {} with your desired values

(e.g., {db_name} →shop-db).

# MySQL Container with Docker Network

Before building the Java image, we'll set up a Docker network and a MySQL database container so that Java tests can run smoothly.

1. Create a Custom Docker Network

To allow containers (e.g., your Java app and MySQL) to communicate with each other, you'll first need to create a Docker network.

```
docker network create {first-app-network}
```

- This creates a bridge network named first-app-network.
- This network allows seamless communication between the app and database containers.
- It eliminates the need to expose internal container ports to your host machine.
- You can reference containers by name (e.g., mysql-container-name) within the same network.

2. Start a MySQL Container

```
docker run -d \
  --name {mysql-container-name} \
  --network {first-app-network} \
  -e MYSQL_DATABASE={db_name} \
  -e MYSQL_USER={mysql-username} \
  -e MYSQL_PASSWORD={mysql-password} \
  -e MYSQL_ROOT_PASSWORD={mysql-root-password} \
  -v mysql-data:/var/lib/mysql \
  -p {5050}:3306 \
mysql
```

This launches a MySQL container with the specified credentials and configuration.

3. Confirm Everything Is Running

```
docker ps
```

4. Connect to the MySQL Server

Since we mapped port 3306 in the container to port 5050 on your local machine, you can connect using either of the following methods:

- From Your Local Machine (host system):

```
mysql -h 127.0.0.1 -P {5050} -u {xyz} -p
```

- Directly from the Container:

```
docker exec -it {app-db} mysql -u {xyz} -p
```

You'll be prompted to enter the password (xyz@123).

Method 1: Using Dockerfile

Dockerfile for maven projects:

```
# ---- Stage 1: Build the application ----
FROM maven:3.9.9-amazoncorretto-21-debian AS builder

# Set the working directory in container
WORKDIR /app

# Copy Maven POM and download dependencies
COPY pom.xml .
RUN mvn dependency:go-offline -B

# Copy the actual source code
COPY src ./src

# Package the application
RUN mvn clean package -DskipTests

# ---- Stage 2: Create a lean runtime image ----
FROM bellsoft/liberica-openjre-debian:21

# Set environment variables
ENV JAVA_OPTS="-Xms256m -Xmx512m"
ENV SPRING_DATASOURCE_URL=jdbc:mysql://{mysql-container-name}:3306/{db_name}
ENV SPRING_DATASOURCE_USERNAME={mysql-username}
ENV SPRING_DATASOURCE_PASSWORD={mysql-password}
ENV APP_HOME=/app

# Set working directory
WORKDIR $APP_HOME

# Copy the jar from the builder stage
COPY --from=builder /app/target/{java-app-0.0.1}.jar {app}.jar

# Expose the application port
EXPOSE 8080

# Run the jar file
ENTRYPOINT ["sh", "-c", "java $JAVA_OPTS -jar {app}.jar"]
```

Dockerfile for gradle projects:

```
# ---- Stage 1: Build the application ----
FROM gradle:8.5-jdk21-alpine AS builder
# Set the working directory in container
WORKDIR /app

# Copy Gradle files
COPY build.gradle settings.gradle ./
COPY gradlew ./
COPY gradle ./gradle

# Download dependencies
RUN ./gradlew dependencies --no-daemon

# Copy the actual source code
COPY src ./src

# Package the application
RUN ./gradlew clean build -x test --no-daemon

# ---- Stage 2: Create a lean runtime image ----
FROM bellsoft/liberica-openjre-debian:21

# Set environment variables
ENV JAVA_OPTS="-Xms256m -Xmx512m"
ENV
SPRING_DATASOURCE_URL=jdbc:mysql://{mysql-container-name}:3306/{db_name}
ENV SPRING_DATASOURCE_USERNAME={mysql-username}
ENV SPRING_DATASOURCE_PASSWORD={mysql-password}
ENV APP_HOME=/app

# Set working directory
WORKDIR $APP_HOME

# Copy the jar from the builder stage
COPY --from=builder /app/build/libs/{java-app-1.0.0}.jar {app}.jar

# Expose the application port
EXPOSE 8080

# Run the jar file
ENTRYPOINT ["sh", "-c", "java $JAVA_OPTS -jar {app}.jar"]
```

About this Dockerfile

This setup uses multi-stage builds:

Stage 1: Compiles the JAR using Maven or Gradle.
Stage 2: Runs the JAR using a lightweight JRE (BellSoft Liberica JRE 21).

📝 Note: You can use any JDK vendor like Temurin, AdoptOpenJDK, or BellSoft. BellSoft's version is used here for its lightweight footprint.

Tips

- Match the JAR file name in the Dockerfile to your actual build output.
- Create a .dockerignore file to avoid unnecessary files in your Docker context.

Build and Run Your Docker Image:

Build:

```
docker build -t {java-app-image}:{1.0.0} ./
```

Run:

```
docker run -d \
  --name {java-app-container} \
  --network {first-app-network} \
  -p {8080}:8080 \
  {java-app-image}
```

Access Shell:

```
docker exec -it {java-app-container} bash
```

Attach Shell:

```
docker logs {java-app-container}
```

Check Logs:

```
docker attach {container-id}
```

Method 2: Buildpack plugin

Why Use Buildpacks?

- No Dockerfile required
- Automatically optimized container images
- Layered caching for faster builds
- Great for rapid prototyping

There are several ways to create Docker images using Buildpacks:

- Using the pack CLI – the official Buildpacks command-line tool
- Using Docker BuildKit with buildx and the Buildpacks driver
- Using CI/CD Buildpack providers, such as GitHub Actions or GitLab CI

While these methods are powerful and flexible, we'll focus on the simplest approach for now — using the built-in Buildpacks plugin available in Gradle or Maven.

This method may consume significant memory and disk space during the build process.If you're looking for a more lightweight alternative, consider using Jib by Google, which offers lower memory and disk usage with a very similar process.

Maven (pom.xml) Setup

Update your Spring Boot plugin section

```
<build>
    <finalName>
            ${project.artifactId}-0.0.1
    </finalName>
    <plugins>
     <plugin>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-maven-plugin</artifactId>
       <configuration>
         <image>
             <name>${project.artifactId}</name>
         </image>
         <!-- Other existing configurations -->
       </configuration>
       <executions>
```

```
              <execution>
                  <goals>
                      <goal>build-image</goal>
                  </goals>
              </execution>
            </executions>
          </plugin>
        <!-- Other plugins -->
      </plugins>
</build>
```

Gradle (build.gradle) Setup

```
...
bootJar {
    archiveBaseName.set("hello-docker")
    archiveVersion.set("1.0.0")
}

bootBuildImage {
    imageName = "${bootJar.archiveBaseName.get()}"
    environment = [
        'BP_JVM_VERSION': '21'
    ]
}
...
```

⚠️ Ensure your DB credentials are configured properly if running tests. Otherwise, skip tests using -DskipTests or -x test.

Build the Image

```
# Maven
./mvnw spring-boot:build-image -DskipTests

# Gradle
./gradlew bootBuildImage -x test
```

Run the Container

```
docker run -d \
  --network {first-app-network} \
  -e SPRING_DATASOURCE_USERNAME={mysql-username} \
  -e SPRING_DATASOURCE_PASSWORD={mysql-password} \
  -e SPRING_DATASOURCE_URL=
jdbc:mysql://{mysql-container-name}:3306/{db_name} \
  --name {java-app-container} \
  -p 8080:8080 \
  {java-app-image}
```

Further Reading

Dockerfile Reference (Official Docs)