

Mastering Docker Image Building for Java

Microservices

Building the Docker image using Dockerfile:

What is Dockerfile?

A Dockerfile is a text file containing a set of instructions to build a Docker image. It automates the process of creating a containerized application by specifying the base image, dependencies, configurations, and commands required for the application to run.

◆ Step-by-Step Guide to Create a JAR File for Your Accounts Microservice and Run It Using Docker

Creating a JAR file for your **Accounts Microservice** and preparing it for **Docker** deployment involves several steps. Here's an improved, accurate, and organized approach to ensure clarity and efficiency:

● Step 1: Update pom.xml for Packaging

1. Open your **pom.xml** file in the **Accounts Microservice** project.
2. Locate the `<version>` tag in the `<project>` block.
3. Just below the `<version>` tag, add the following to specify the packaging format as **JAR**:

```
<packaging>jar</packaging>
```

✓ This ensures that your Spring Boot application is packaged as a JAR file during the build process.

● Step 2: Clean the Target Folder (Remove Old Compiled Files)

1. Navigate to the **target** folder inside your **Accounts Microservice** project directory.
2. Delete all the existing compiled classes and files inside the target folder to ensure a clean build.

✓ Removing old files avoids conflicts with outdated or incorrect builds.

● Step 3: Build the JAR File Using Maven

1. Open the **Command Prompt (CMD)** or **Terminal** in the **Accounts Microservice** project folder (where your **pom.xml** is located).
2. Run the following Maven command to build the JAR file:

```
mvn clean install
```

✓ This command will:

- Clean the project (removing old compiled files).

- Compile your Spring Boot application.
 - Run unit tests (if present) to ensure code stability.
 - Package the compiled code into a **JAR file**.
-

● Step 4: Verify the Generated JAR File

1. After the build is successful, navigate to the **target** folder.
2. You should see a newly generated JAR file named:

`accounts-0.0.1-SNAPSHOT.jar`

✓ This is called a **Fat JAR** because it contains:

- Your compiled code.
 - All required dependencies (except the Java Runtime).
-

● Step 5: Run the Application Using Maven

1. To run the JAR file directly using Maven, use the following command:

`mvn spring-boot:run`

✓ This command:

- Uses the **spring-boot-maven-plugin** in pom.xml.
 - Looks for the JAR file inside the target folder.
 - Starts the web application.
2. To stop the running application, press:

`Ctrl + C`

✓ This will gracefully terminate the running instance.

● Step 6: Run the Application Using Java Command (Alternative Approach)

1. Alternatively, you can run the JAR directly using the java command:

`java -jar target/accounts-0.0.1-SNAPSHOT.jar`

✓ This command:

- Launches your Spring Boot application using the generated JAR file.
 - Provides better control when deploying the application in environments like **Docker** or **Kubernetes**.
-

● Step 7: Creating the Dockerfile for Containerization

1. In your **Accounts Microservice** project root directory, create a new file named **Dockerfile** (without any file extension).
2. Add the following content to define your Docker build process:

```
# Base Image
FROM openjdk:21-jdk-slim

# Set the working directory
WORKDIR /app

# Copy the JAR file to the container
COPY target/accounts-0.0.1-SNAPSHOT.jar app.jar

# Expose the port on which your application runs
EXPOSE 8080

# Command to run the application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

✓ This Dockerfile will:

- Use the **OpenJDK 21** base image for better performance.
- Copy your generated JAR file into the container.
- Expose **port 8080** (or whichever port your application runs on).
- Run the JAR file when the container starts.

● Step 8: Build the Docker Image

Run the following command from your **Accounts Microservice** project folder (where the Dockerfile is located):

```
docker build -t accounts-microservice .
```

✓ This command builds the Docker image with the tag `accounts-microservice`.

● Step 9: Run the Docker Container

To start a container using the newly created Docker image, run:

```
docker run -d -p 8080:8080 --name accounts-container accounts-microservice
```

✓ Explanation:

- -d → Runs the container in detached mode (in the background).
- -p 8080:8080 → Maps your local port **8080** to the container's **8080** port.
- --name accounts-container → Assigns a custom name to your container.

🔴 Step 10: Verify the Running Container

To confirm the container is running successfully, use:

```
docker ps
```

✓ This will display the active containers along with their **ID**, **Name**, and **Port Mapping**.

✓ Additional Tips for Best Practices

- ✓ Ensure application.properties or application.yml is configured correctly for database connections.
- ✓ Use .dockerignore to exclude unnecessary files (e.g., .git, target/, .env) from the Docker image.
- ✓ Add **health checks** to your Dockerfile for better monitoring.
- ✓ Use environment variables for sensitive data like database credentials instead of hardcoding them.

🚀 Final Workflow Recap (Quick Steps)

1. Add `<packaging>jar</packaging>` in pom.xml.
2. Clean the target/ folder.
3. Run `mvn clean install` to build the JAR.
4. Verify the JAR file in the target/ folder.
5. Run the application using:
 - `mvn spring-boot:run` or
 - `java -jar target/accounts-0.0.1-SNAPSHOT.jar`.
6. Create a Dockerfile for containerization.
7. Build the Docker image using `docker build`.
8. Run the container using `docker run`.

Dockerfile for Spring Boot - Comprehensive Guide

Step 1: Base Image

```
FROM openjdk:21-jdk-slim
```

✓ **FROM** → Specifies the **base image** that the container will use.

- openjdk:21-jdk-slim is a lightweight image that contains **JDK 21** with minimal dependencies to reduce the image size.

🔗 Why Use openjdk:21-jdk-slim?

- The **slim** variant minimizes the attack surface and improves performance.
- Suitable for production as it excludes unnecessary files.

Step 2: Maintainer Information (Metadata)

```
LABEL "org.opencontainers.image.authors"="hello.com"
```

✓ **LABEL** → Adds **metadata** about the image for documentation and identification purposes.

🔗 Example to Add Multiple Labels:

```
LABEL version="1.0"
```

```
LABEL description="Accounts Microservice Image"
```

```
LABEL maintainer=" hello.com "
```

Step 3: Copying the JAR File

```
COPY target/accounts-0.0.1-SNAPSHOT.jar accounts-0.0.1-SNAPSHOT.jar
```

✓ **COPY** → Copies files or directories from your local machine (host) to the container's filesystem.

✓ In this case:

- **target/accounts-0.0.1-SNAPSHOT.jar** → The JAR file generated after building your Spring Boot app.
- **accounts-0.0.1-SNAPSHOT.jar** → The file name inside the container.

🔗 Why Use COPY Instead of ADD?

- **COPY** is safer for file transfers and is recommended for copying local files.
- **ADD** can auto-extract .tar files but may introduce unwanted behavior.

Step 4: Defining the Entrypoint

```
ENTRYPOINT ["java", "-jar", "accounts-0.0.1-SNAPSHOT.jar"]
```

✓ **ENTRYPOINT** → Defines the **main process** the container will execute.

✓ In this case:

- java → Refers to the **Java runtime** to execute the JAR file.
- -jar → Tells Java to **run the JAR file**.
- "accounts-0.0.1-SNAPSHOT.jar" → The file name inside the container.

Why Use ENTRYPOINT Instead of CMD?

- **ENTRYPOINT** makes the container behave like an **executable**.
- **CMD** is used for **default arguments** but can be overridden easily.

Example Combining Both:

```
ENTRYPOINT ["java", "-jar", "app.jar"]  
CMD ["--server.port=8085"]
```

Step 5: Exposing a Port (Optional but Recommended)

```
EXPOSE 8080
```

EXPOSE → Marks the **application port** for reference.

- Used for documentation purposes to indicate which port the application listens to.
- Ports are not published automatically unless specified in docker run.

Example Command to Publish the Port:

```
docker run -d -p 8080:8080 accounts-container
```

Step 6: Final Optimized Dockerfile

```
# Base Image with JDK 21  
FROM openjdk:21-jdk-slim  
  
# Metadata for image details  
LABEL "org.opencontainers.image.authors"=" hello.com "  
  
# Copy the JAR file to the container  
COPY target/accounts-0.0.1-SNAPSHOT.jar app.jar  
  
# Expose the application port  
EXPOSE 8080
```

```
# Command to run the application  
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Step 7: Build and Run Commands

Now we are going to give our Docker commands to the Docker server for image generation.

✓ Build the Docker Image:

```
docker build . -t dockerhub_username/accounts:s4
```

Explanation:

- **docker build** → Tells Docker to create a new image.
- **.** → Represents the **current directory** (where the Dockerfile is located).
- **-t dockerhub_username/accounts:s4** → Assigns a tag to the built image:
 - **dockerhub_username** → Your Docker Hub username.
 - **accounts** → Image name.
 - **s4** → Version tag.

✓ Run the Docker Container:

```
docker run -p 8080:8080 dockerhub_username/accounts:s4
```

- Runs the container in **foreground mode**. Logs will be displayed directly in the terminal. Press **CTRL + C** to stop it.

```
docker run -d -p 8080:8080 dockerhub_username/accounts:s4
```

- Runs the container in **detached mode** (background process). Use `docker logs <container_name>` to view logs.

🔗 Explanation of the Command

```
docker run -d -p 8081:8080 dockerhub_username/accounts:s4
```

This command runs a Docker container from the image **dockerhub_username/accounts:s4**. Here's a breakdown of each part:

- **docker run** → Creates and starts a new container.
- **-d** → Runs the container in **detached mode** (in the background).
- **-p 8081:8080** → Maps **port 8081** on your **host machine** to **port 8080** inside the **container**.
- **dockerhub_username/accounts:s4** → Refers to the **image name** and **tag**. The **:s4** is the **tag** (version identifier) for the image.

Why Can You Run the Same Container on Different Ports?

Docker containers are **isolated environments** that run independently. Each container has its own filesystem, network, and process space. This allows:

- ✓ Running multiple instances of the **same image** on **different ports**.
- ✓ Each instance behaves like a separate application instance.

For example:

```
docker run -d -p 8082:8080 dockerhub_username/accounts:s4
```

```
docker run -d -p 8083:8080 dockerhub_username/accounts:s4
```

Here, you're effectively running **three instances** of the same application on ports **8081**, **8082**, and **8083**.

Advantages of Running Multiple Instances

1. Scalability

- Running multiple containers on different ports allows you to handle **increased traffic** by distributing the load across instances.

2. Portability

- Since containers include the application and its dependencies, they run consistently across different environments (development, testing, production).

3. Fault Tolerance

- If one container fails, other running containers ensure your application remains available.

4. Efficient Resource Utilization

- Containers use system resources efficiently by sharing the host OS kernel.

5. Fast Deployment

- Starting a new container is much faster than starting a new virtual machine.

6. No Need for Maven or Dependencies

- **You are not required to download Maven, dependencies, or even Spring Boot** as long as you have Docker. Everything your application needs is packaged inside the container image.

Real-World Example: Load Balancing

In a production environment, multiple container instances are often deployed behind a **load balancer** to distribute incoming requests evenly. This improves performance and ensures reliability.

For instance:

- **http://localhost:8081 → Instance 1**
- **http://localhost:8082 → Instance 2**
- **http://localhost:8083 → Instance 3**

A **load balancer** can route requests to any of these instances, ensuring smooth traffic distribution.

◆ Key Takeaway

Running multiple instances using Docker containers helps achieve **scalability**, **portability**, and **high availability**, which are essential for modern application deployment. Plus, with Docker, you skip the hassle of managing **Maven**, **Spring Boot**, or **dependencies** — it's all packaged within the container.

⚠ Challenges with Dockerfile Approach to Generate Docker Image

1. Manual Process

- Writing and maintaining a Dockerfile requires manual effort, which can be error-prone.

2. Complex Configuration

- Managing environment variables, dependencies, and multi-stage builds can become complex.

3. Caching Issues

- Improper layering can lead to inefficient caching, increasing build time.

4. Security Risks

- Using outdated base images or failing to apply security patches can introduce vulnerabilities.

5. Image Size Optimization

- Without proper steps like removing unnecessary dependencies, image size can grow significantly.

6. Version Control Challenges

- Tracking changes in Dockerfiles across multiple services in microservices architecture can be cumbersome.

7. Dependency Conflicts

- Misconfigurations in dependencies may cause compatibility issues.

8. Port Management

- Defining correct port mappings can be tricky, especially for multiple services.

9. Build Time

- Complex builds with multiple steps may result in longer build times.

10. Best Practices Maintenance

- Following best practices (e.g., minimizing layers, using .dockerignore) requires continuous attention.

Another Approach to generate the Docker Image (using Buildpacks)

What Are Buildpacks?

Buildpacks are a set of tools that automatically detect, compile, and package your application into a container image (like a Docker image) without requiring a **Dockerfile**.

They simplify the process of building secure, production-ready images by handling:

- ✓ **Dependency Installation**
- ✓ **Source Code Compilation**
- ✓ **Image Packaging**
- ✓ **Layer Optimization**

◆ Why Were Buildpacks Introduced?



The traditional Dockerfile approach requires developers to:

- Write a Dockerfile manually
- Manage dependencies and environment configurations
- Handle security updates and performance optimizations

Buildpacks automate these tasks, making the process easier, faster, and more secure.

◆ How Do Buildpacks Work?

Buildpacks operate in **three key stages**:

1.  **Detection**
 - Identifies the application's language (e.g., Java, Node.js, Python).
 - Chooses the appropriate buildpack for the detected language.
2.  **Build**
 - Installs dependencies.

- Compiles and packages the application.

3. 📦 Export

- Generates a container image with all required dependencies and runtime environment.
- Ensures the image follows best practices for performance and security.

◆ Buildpacks Workflow Example (Spring Boot App)

Step 1: Add spring-boot-maven-plugin in pom.xml

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <image>
          <name> dockerhub_username/${project.artifactId}:s4</name>
        </image>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Step 2: Build the Docker Image

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

Step 3: Run the Container

```
docker run -d -p 8080:8080 dockerhub_username/accounts:s4
```

◆ Key Advantages of Buildpacks

- ✓ **No Dockerfile Required** — Automatically configures the image.
- ✓ **Language Detection** — Detects and builds apps in Java, Node.js, Python, etc.
- ✓ **Optimized Images** — Uses caching to speed up rebuilds.
- ✓ **Enhanced Security** — Ensures the base image is up-to-date.
- ✓ **Portability** — Buildpacks are compatible with **Docker**, **Kubernetes**, and major cloud providers like

AWS, Azure, and GCP.

- ✅ **Developer-Friendly** — Reduces the complexity of managing Dockerfile syntax.

◆ When to Use Buildpacks?

- ✅ When you want a **faster** and **simpler** way to build Docker images.
- ✅ When you need **consistent**, **secure**, and **optimized** images for production.
- ✅ When you prefer **automated dependency handling** instead of writing complex Dockerfile steps.

◆ Popular Buildpack Providers

- **Paketobuildpacks** (recommended for Java, Node.js, Python, etc.)
- **Heroku Buildpacks**
- **Google Cloud Buildpacks**

◆ Key Takeaway

Buildpacks provide an effortless way to create lightweight, secure, and efficient container images. They are ideal for microservices, cloud deployments, and CI/CD pipelines.

Generate Docker image card microservice with google lib

What is Google Jib?

Google Jib is an open-source Java containerization tool developed by Google. It simplifies the process of creating Docker images for Java applications without requiring Docker to be installed or writing a Dockerfile.

Generating Docker Image for Card Microservice with Google Jib

To generate a Docker image for your **Card Microservice** using Google Jib, follow these steps:

Step 1: Add Jib Plugin to pom.xml

In the <build> section of your pom.xml, add the Jib plugin:

```
<build>
  <plugins>
    <plugin>
      <groupId>com.google.cloud.tools</groupId>
      <artifactId>jib-maven-plugin</artifactId>
      <version>3.4.5</version>
```

```
<configuration>

  <to>

    <image>docker_hub_username/${project.artifactId}:s4</image>

  </to>

</configuration>

</plugin>

</plugins>

</build>
```

Step 2: Build the Image

Run the following Maven command to build the Docker image using Jib:

```
mvn compile jib:dockerBuild
```



Step 3: Run the Docker Container

Once the image is built successfully, run the container using:

```
docker run -d -p 8090:8090 docker_hub_username/cards:s4
```

Advantages of Google Jib Over Traditional Methods



















Feature	Google Jib	Traditional Dockerfile Approach
No Dockerfile Required	✅ Jib eliminates the need to write a Dockerfile.	❌ Requires writing a Dockerfile manually.
No Docker Daemon Required	✅ Jib does not require Docker to be installed for building images.	❌ Docker must be installed to build images.
Faster Builds	✅ Jib optimizes image layers to improve build speed.	❌ Builds may be slower due to inefficient layering.
Automated Dependency Management	✅ Jib efficiently handles dependencies, minimizing the risk of missing files.	❌ Requires manual setup in the Dockerfile.
Secure Build Process	✅ Builds images directly from Java code, reducing security risks.	❌ Involves manual steps that may introduce vulnerabilities.
Easy Configuration	✅ Configured directly in pom.xml or build.gradle.	❌ Requires additional Docker-specific configurations.

Automatic Layering	 Separates application code, dependencies, and resources into layers for faster updates.	 Requires manual layer optimization.
---------------------------	---	---

Why Use Google Jib?

- Great for **CI/CD pipelines** since it doesn't require Docker.
- Reduces **image size** by separating dependencies and app code into layers.
- Ensures **faster rebuilds** by caching unchanged dependencies.

Comparison of Dockerfile, Buildpacks, and Google Jib for Creating Docker Images

Feature / Aspect	Dockerfile	Buildpacks	Google Jib
Ease of Use	 Requires writing and maintaining a Dockerfile.	 Simplifies the process by auto-detecting dependencies and configurations.	 No need to write a Dockerfile; simple Maven/Gradle integration.
Docker Installation	 Requires Docker to be installed.	 Not mandatory; leverages Buildpack tools (e.g., pack).	 Does not require Docker to build images but requires Docker for running the image.
Build Speed	 Slower as all steps (e.g., copying files, installing dependencies) are executed in sequence.	 Faster since it efficiently handles caching and layering.	 Fastest build process by optimizing layers and dependencies.
Image Size Optimization	 Manual layer optimization needed for efficient caching.	 Automatically optimizes layers for efficient caching.	 Excellent layer optimization by separating dependencies, resources, and code.
Configuration Flexibility	 Full control via Dockerfile commands.	 Limited customization; follows predefined build logic.	 Less flexible than Dockerfile but customizable using Maven/Gradle.
Security	 Higher risk if insecure base images are used.	 Ensures secure base image selection via trusted sources.	 Builds directly from Java source code.

			reducing potential vulnerabilities.
Dependency Management	✗ Manual management required (e.g., COPY commands).	✓ Automatically detects dependencies via build system.	✓ Automatically manages dependencies through Maven/Gradle configuration.
Portability	✗ Docker-specific; may require tweaks for different environments.	✓ Highly portable and works across various platforms.	✓ Portable and does not rely on Docker during build.
CI/CD Integration	✗ Requires Docker in the CI/CD pipeline.	✓ Easily integrates with modern CI/CD tools.	✓ Ideal for CI/CD as it doesn't require Docker to build.
Resource Efficiency	✗ Consumes more resources as Docker builds everything together.	✓ Efficient resource usage through smart caching.	✓ Highly efficient due to smart layering and minimal rebuilds.
Learning Curve	✓ Easy to learn for beginners familiar with Docker.	✗ Slightly higher learning curve due to new concepts like pack.	✓ Simple for Java developers familiar with Maven/Gradle.

Summary of When to Use Each Approach

✓ Use Dockerfile if:

- You need **full control** over the image-building process.
- Your application includes **custom scripts** or complex configurations.

✓ Use Buildpacks if:

- You prefer an **automated build process** with minimal configuration.
- You want a **secure** and **portable** solution for Java, Python, Node.js, etc.

✓ Use Google Jib if:

- You are working on **Java projects** using **Maven/Gradle**.
 - You need **fast builds** without requiring Docker on the build machine.
-

Best Choice for Microservices in Spring Boot

For a scalable and efficient microservice architecture, **Google Jib** or **Buildpacks** is generally preferred due to faster builds, better caching, and reduced complexity. However, if you need custom configurations or scripts, **Dockerfile** may still be the best choice.
