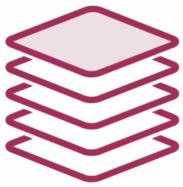
Docker

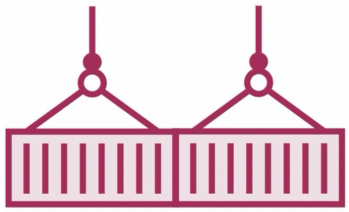
- Getting Started with Docker for Java
- Building Java Applications with Dockerfiles
- Building Java Applications with Build Tools and Plugins
- Running Multi-Container Java Applications with Docker Compose
- Configuring Java Applications in Containers
- Managing Application Logs with Docker
- Developing Java Applications in an IDE with Docker Support
- Debugging Java Applications Running in Containers

Let's create a Dockerfile to run our sample JAR and WAR applications
This works in a similar way Maven works. Maven uses a pom.xml file to resolve dependencies and build the project. Docker uses a Dockerfile. Maven downloads missing JAR files from Maven repositories. Docker downloads image files from registries like Docker Hub, and both store these files in a local repository or registry for future builds. Now, once you pull an image from the registry or build your own with a Dockerfile, you'll want to instantiate a container from that image and run it on the Docker Engine. The Docker Engine is a process that runs between a container and the underlying operating system and hardware

Images and Containers



```
class Application {
  private String name;
  // ...
}
```

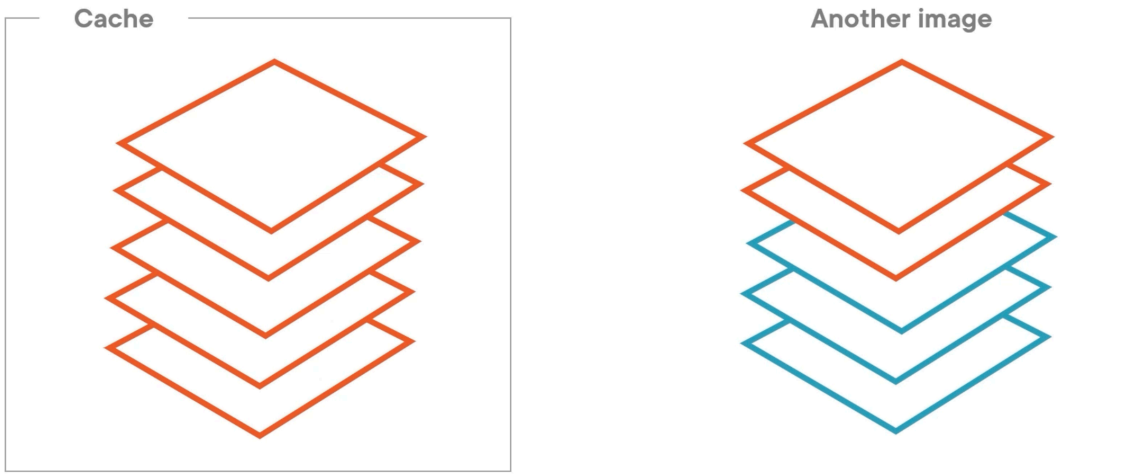


```
Application app1 = new Application();
app1.setName("todo-list");
Application app2 = new Application();
app2.setName("rest-api");
```

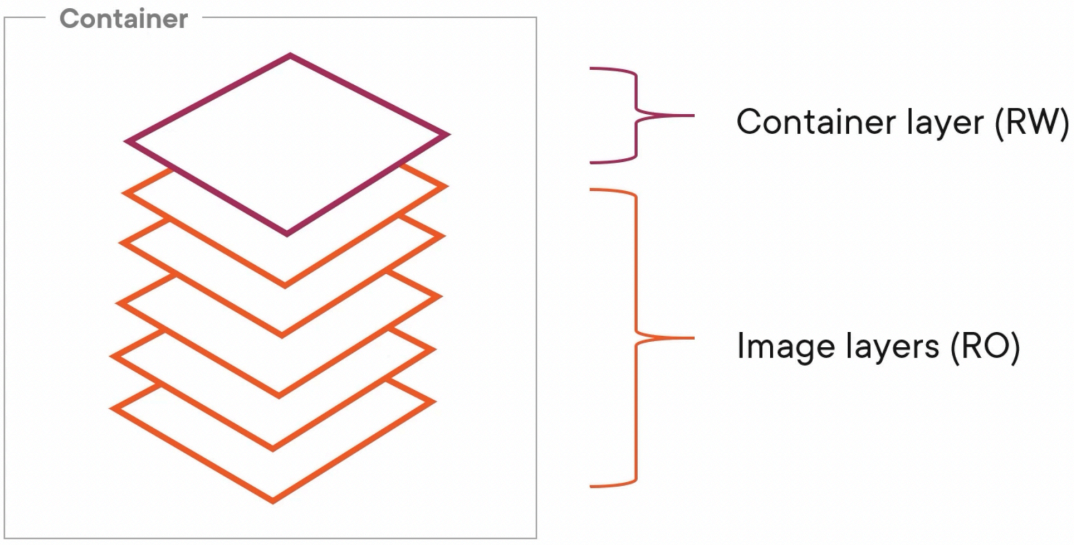
Dockerfile

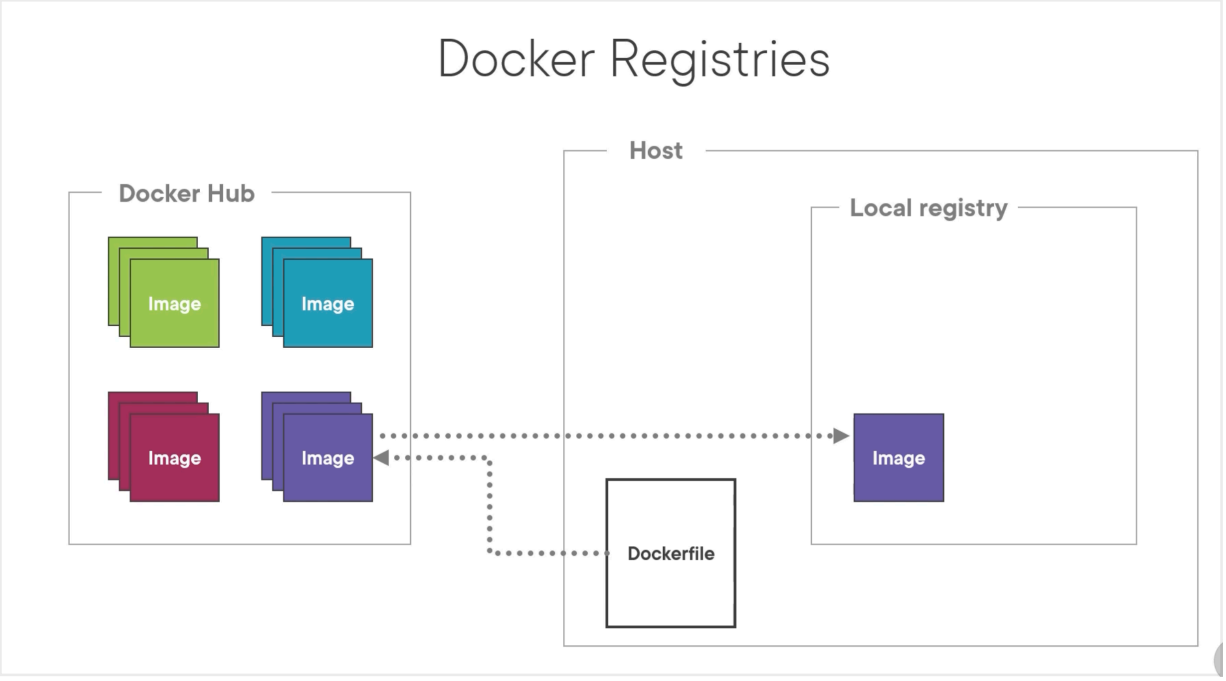
```
FROM openjdk:slim-buster
COPY . /my-app
WORKDIR /my-app
RUN javac App.java
CMD ["java", "App"]
```

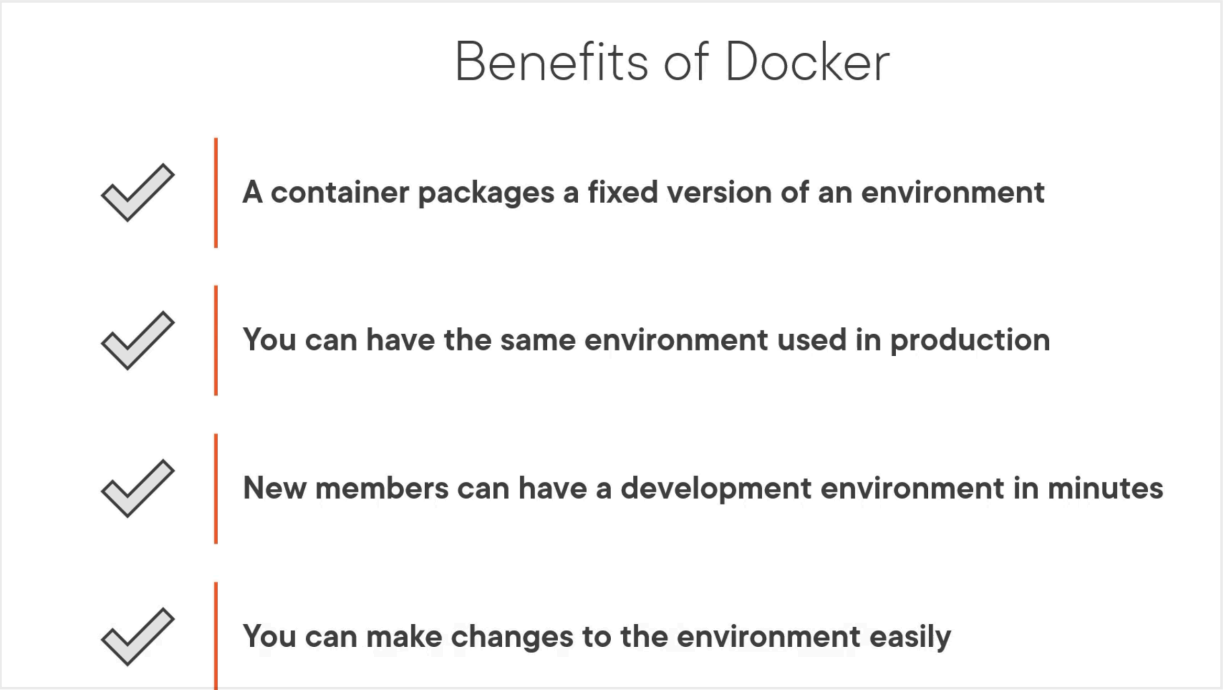
Image Layers



Container's Writable Layer







Let's create a Hello World program, hello.java public class Hello { public static void main(String[] args) { System.out.println("Hello World");. Let's save the file. Great. Now we need to compile the program. One of the great things about Docker is that it allows you to run a tool or service as a container without installing it. So we are going to use the docker run command to pull an image to compile and run our program. All right. Let's go to Docker Hub

. This is the first time I use this image, so Docker will pull the image and all its

layers and store them in my local cache. The docker run command creates and starts a container. So after a few moments, javac comprises the program and the container exits, and there you have it, the class file. We can use the same image and command options to run the program, but instead of javac, now it's just Java and the class hello, Hello World, great. But we don't need a JDK to run our program, a JRE will be enough. So instead of using the tag 11.0.10-buster, we can use 11.0.10-jre-buster. Docker will download the new image. Some instructions to build both images should be the same because some layers are already in the cache, but in the end, the result is Hello World. With Docker, we can switch tools or versions easily and we don't need to have Java installed in our machine to compile and run a program. But, is this the best way of developing Java applications with Docker? Of course not. This was just a demonstration to show what Docker can do. Besides, using commands is error-prone, it's better to use a Docker file to specify all the steps to build and run an application. In the next module, you'll learn many ways to develop Java applications with Docker. For now, let me give you one last piece of advice. If I list the images I have on my machine right now, I downloaded almost 1 GB of data, and each docker run command I executed created one container. We can use docker run with the option --rm, so the container is automatically removed when it exits. But to delete all unused data, every now and then run the command docker system prune -a to delete stopped containers, unused networks and images, and the build cache. All right, now let's wrap up this module.

Run “”hello world” from Docker

```
➜  hello vim Hello.java
➜  hello ls
Hello.java
➜  hello docker run -v ${PWD}:/hello -w /hello openjdk:11.0.10-buster javac Hello.java

Unable to find image 'openjdk:11.0.10-buster' locally
11.0.10-buster: Pulling from library/openjdk
0ecb575e629c: Pull complete
7467d1831b69: Pull complete
feab2c490a3c: Pull complete
f15a0f46f8c3: Pull complete
26cb1dfcbebb: Pull complete
242c5446d23f: Pull complete
f22708c7c9c1: Pull complete
Digest: sha256:9b0906be409b84fba6f02cbf752c8db3aefc833646cf5f5e29f21677b204fa86
Status: Downloaded newer image for openjdk:11.0.10-buster
➜  hello ls
Hello.class Hello.java
➜  hello docker run -v ${PWD}:/hello -w /hello openjdk:11.0.10-buster java Hello
Hello World
➜  hello
```

```
➜  hello docker run -v ${PWD}:/hello -w /hello openjdk:11.0.10-jre-buster java Hello
Unable to find image 'openjdk:11.0.10-jre-buster' locally
11.0.10-jre-buster: Pulling from library/openjdk
0ecb575e629c: Already exists
7467d1831b69: Already exists
feab2c490a3c: Already exists
2596367ee94b: Pull complete
78f07041e880: Pull complete
8546227cada0: Pull complete
Digest: sha256:23cb8bac240373d54357cf9bf813320a2beecd3948f402faf1c7de7407042976
Status: Downloaded newer image for openjdk:11.0.10-jre-buster
Hello World
```

```
➜  hello docker image ls
REPOSITORY   TAG                 IMAGE ID       CREATED       SIZE
openjdk      11.0.10-jre-buster  ecc8529099c8   10 days ago   301MB
openjdk      11.0.10-buster      82e02728b3fd   10 days ago   647MB
➜  hello docker container ls -a
CONTAINER ID   IMAGE                          COMMAND             CREATED
STATUS                          PORTS     NAMES
931b7836859f   openjdk:11.0.10-jre-buster   "java Hello"         About a minute ago
Exited (0) About a minute ago             clever_dubinsky
89f6e4c60d60   openjdk:11.0.10-buster       "java Hello"         2 minutes ago
Exited (0) About a minute ago             cranky_tereshkova
bea54567f795   openjdk:11.0.10-buster       "javac Hello.java"   2 minutes ago
```

Run JAR and WAR applications with a Dockerfile

Let's create a Dockerfile to run our sample JAR and WAR applications. In the precompiled directory, the directory for this clip, you can find a JAR and a WAR file. For the JAR, I'll create a jar.Dockerfile
With the FROM instruction, I'm going to specify the base image. I compile the JAR with Java 11, so I'll use openjdk:11.0.10, the slim JRE version, You can search for other images on Docker Hub if you want.
now with WORKDIR I'm going to set the working directory to our app directory. If the WORKDIR doesn't exist, it will be created, making the previous RUN instructions unnecessary. But I want to be explicit about what this Dockerfile does. Now let's copy our JAR file from the host to the container with the name app.jar. It will be copied to the work directory defined previously.
COPY is the recommended instruction to use in most cases. Now, with EXPOSE, I'm going to indicate the port on which the application, and therefore the container, will listen for connections. This instruction only works for documentation purposes. We still have to publish and map the port with the docker run command. And finally, with ENTRYPOINT I'll set the image initial command to "java", "-jar", "app.jar". Like RUN, ENTRYPOINT has a shell form and an exec form, which is the preferred one. Be careful with the commas, it's a common error to skip them. There's also a CMD instruction that you can use to specify the initial command of the container.

```
→  precompiled ls
api.jar web.war
→  precompiled vim jar.Dockerfile
```

```
FROM openjdk:11.0.10-jre-slim
RUN mkdir /app
#RUN ["executable", "param1", "param2"]
WORKDIR /app
COPY api.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
~
```

```
→  precompiled ls
api.jar          jar.Dockerfile web.war
```

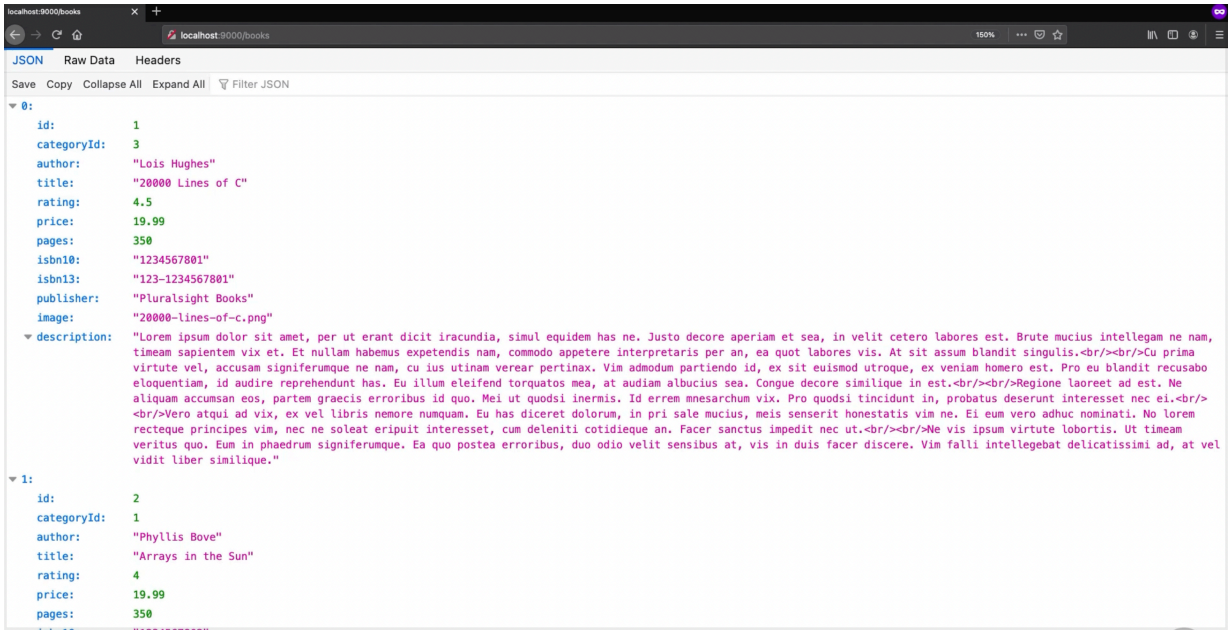
Create DockerImage

```
→  precompiled docker build -f jar.Dockerfile -t my-api .
```

```
→  precompiled docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
my-api       latest    0765529bb579   12 seconds ago   259MB
```

Run Image

```
precompiled docker run -p 9000:8080 -it my-api
```




```
→  precompiled docker container ls -a
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS
        PORTS     NAMES
6a08ecaaad57   my-api    "java -jar app.jar"   26 seconds ago   Exited (130) 10 second
s ago             ecstatic_mendeleev
```

How to Run WAR file in Tomcat


```
precompiled vim web.Dockerfile
```

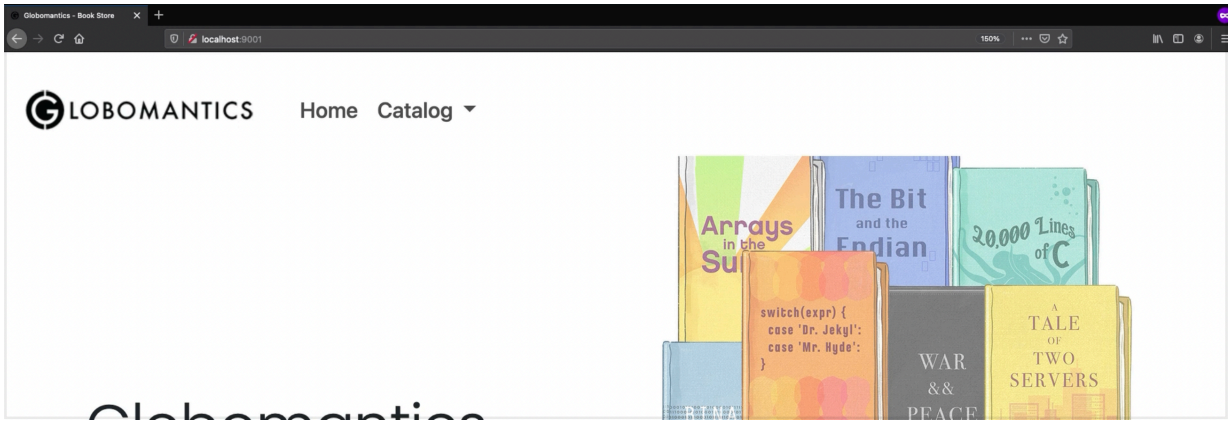
```
FROM tomcat:9
COPY web.war ${CATALINA_HOME}/webapps/ROOT.war
EXPOSE 8080
ENTRYPOINT ["catalina.sh", "run"]
~
```

Build Image

```
➜  precompiled vim web.Dockerfile
➜  precompiled docker build -f web.Dockerfile -t my-web-app .
```

Run Container

```
precompiled docker run -p 9001:8080 -it --rm my-web-app
```



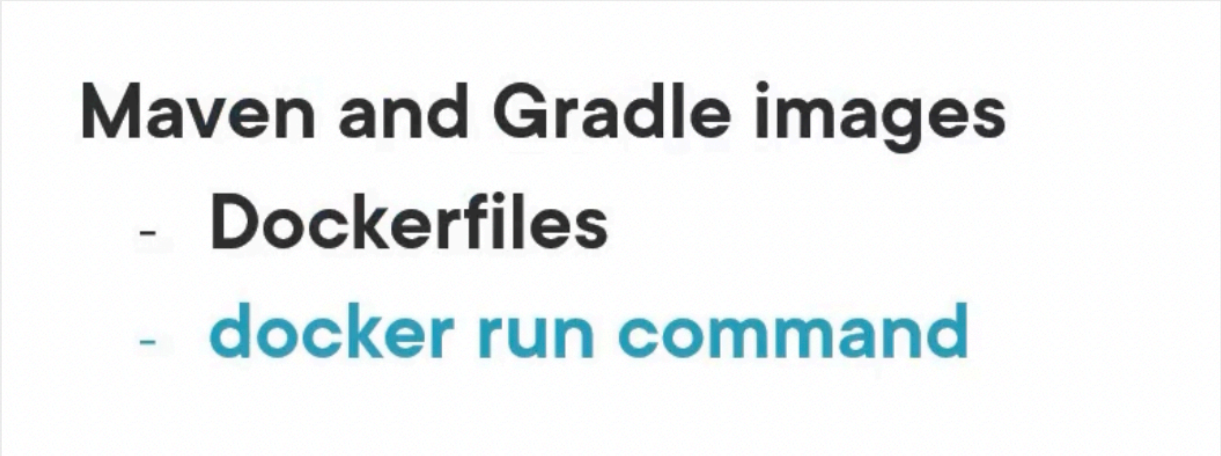
How to work with Maven in Docker

You can use the Maven and Gradle Docker images to build and run your application.
In the maven-gradle directory, the directory for this clip, you can find the api project, which contains both, a pom.xml file and a build.gradle file. I'll started with Maven. So let's create the file, maven.Dockerfile. As the base image, I'll use maven 3.6.3 with jdk-11, the slim version. Next, I'll set the working directory for the app. Remember, it'll be created if it doesn't exist. Now, I'm going to copy the pom file to a working directory, and with another copy instruction, the src directory. It's best to copy multiple files individually rather than all at once, so the build cache is only invalidated if a file changes.

Docker can create a layer with all the dependencies of the project that will be rebuilt only when the pom file changes

Next, we execute mvn package exposing port 8080, and for the ENTRYPOINT of the container, we execute the jar file of the target directory

, let's say in the application.properties file. Any change will do. And build a version two of the image. As you can see, this time, Maven doesn't download the app dependencies. It only has to recompile the project because only the source changed. However, every time the pom file changes, all the dependencies will be downloaded. The solution is to mount a volume so dependencies can be saved or shared between the host and one or more containers. The Gradle image is a step closer to this approach. Here is the Docker file for the Gradle image




```
→  maven-gradle cd api
→  api ls
build.gradle    gradlew         mvnw            pom.xml         src
gradle          gradlew.bat     mvnw.cmd        settings.gradle
```

```
api vim maven.Dockerfile
```

```
FROM maven:3.6.3-jdk-11-slim
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline
COPY src src
RUN mvn package
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "target/api.jar"]
~
```

Build Image

```
api docker build -f maven.Dockerfile -t my-api-maven .
```

Run container

```
api docker run -p 9010:8080 -it --rm my-api-maven
```

Make Changes

```
api vim src/main/resources/application.properties
```

New version

```
api docker build -f maven.Dockerfile -t my-api-maven:v2 .
```

```
FROM gradle:jdk11
USER gradle
WORKDIR /app
COPY --chown=gradle:gradle build.gradle .
COPY --chown=gradle:gradle src src
RUN gradle build
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "build/libs/api.jar"]
~
```



```
api vim gradle.Dockerfile
api docker build -f gradle.Dockerfile -t my-api-gradle .
```

```
api docker run -p 9011:8080 -it --rm my-api-gradle
```

```
→  api docker volume ls
DRIVER    VOLUME NAME
→  api docker run -it --rm \
> -v ${PWD}:/app \
> -v ${HOME}/.m2:/root/.m2 \
> -w /app \
> maven:3.6.3-jdk-11-slim \
> mvn clean package
```

Multi Stage Builds

```
FROM maven:3.6.3-jdk-11-slim AS build
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline
COPY src src
RUN mvn package

FROM tomcat:9
COPY --from=build /app/target/web.war ${CATALINA_HOME}/webapps/ROOT.war
EXPOSE 8080
ENTRYPOINT ["catalina.sh", "run"]
~
```

```
→  multi-stage ls
web
→  multi-stage cd web
→  web vim maven-multi.Dockerfile
→  web docker build -f maven-multi.Dockerfile -t my-web-maven-multi .
[+] Building 0.3s (2/4)
```

#pluralsight