Lab 4. Building Container Images

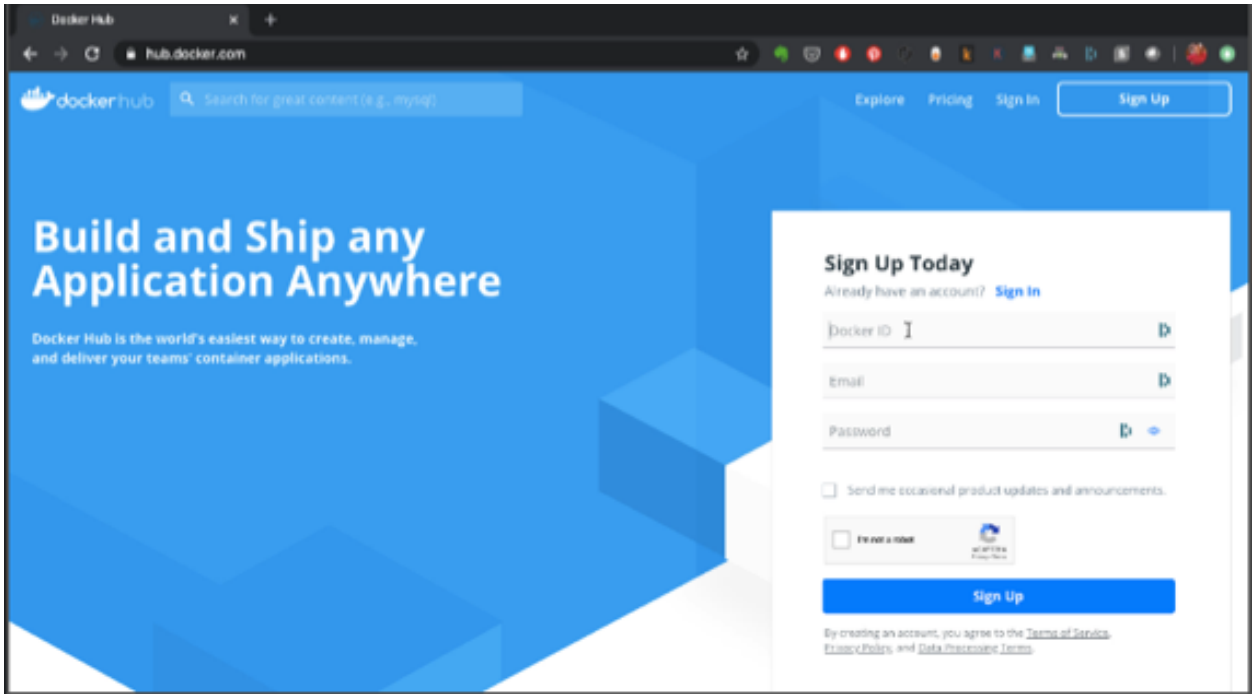
When you start with a container-based software delivery, the first step is always to take your application and package it as a container image along with its run time environment. Mastering the image build process with all of its aspects is an indispensable skill in the container world. The objective of this lab exercise is to help you acquire that skill. You will learn how to package applications with Docker by building images with Linux as the base operating system.

In this exercise we will discuss:

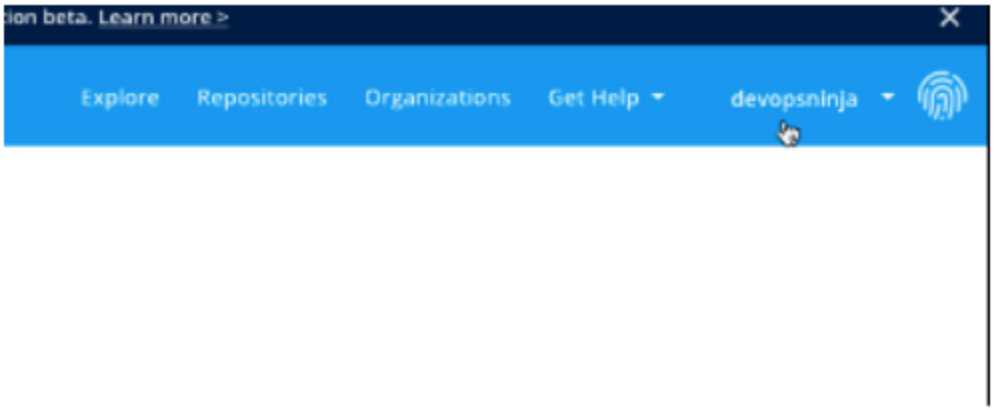
- How to get signed up to the Docker Hub registry and create your Docker ID.
- How to manually test build a Docker image by launching a container with base image and modifying it.
- How to automate the process of building Docker images by writing a Dockerfile.
- How to construct a Dockerfile and what the best practices of writing one are.
- Why you might need a multi-stage Dockerfile and how it works.
- How to package a Spring/Maven-based Java application as well as a C application.

Creating a Registry Account (Docker Hub)

By the end of this exercise, you will have at least a couple of images built. These images can then be published to a registry. You can start with Docker Hub as the default registry and publish your images under your own account. In order to do that, you need to have a Docker Hub account. If you have created a Docker ID while setting up your Docker Desktop software, it's the same account. If not, follow this process to sign up to Docker Hub and validate your account.



- Go to hub.docker.com
- Sign up by providing a username, email and password
- Verify your email address
- Log in

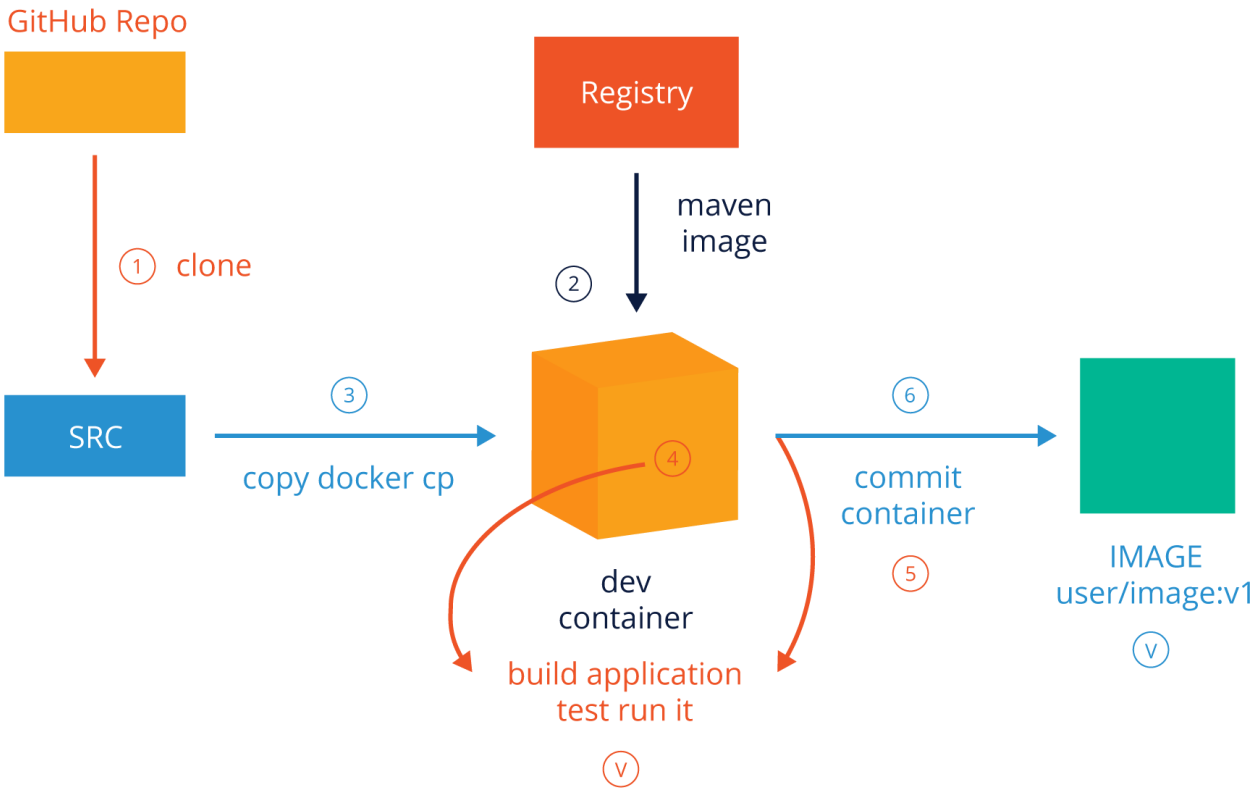


Once logged in, write down your Docker ID located in the top-right corner of the screen. It's your username in lowercase letters. This is an important step because this is the username/ID that you should use while tagging your images later in this chapter.

Test Building a Docker Image (Manual Approach) for a Spring Boot Application Build with Maven/Java

Before you automate the process of building an image, you need to know what you are going to automate. Doing a manual test build of your application helps to understand this process. We will look at the example of this Spring Boot application (spring-petclinic) and try building an image.

You need a Maven + JDK environment to build this application, which in itself, can be created by using a container image. This is followed by the step-by-step instructions on how to create a development environment and build a sample application.



1. Create a fork of devopsdemoapps/spring-petclinic, Spring PetClinic Sample Application to your GitHub account.
2. Clone the code from a GitHub repository to the local development host.
3. Create a container-based development environment and build the application. Use a pre-built Maven image which contains the tools.
4. Copy over the source code to this dev container.
5. Connect to the container and perform all the tasks necessary to build the application.
6. Do a test run of the application to ensure its working fine.
7. Once tested, commit the container's changes to an image using **`docker container commit`** command.

By the end of Step 6, you should get a manually built Docker image. Go ahead and follow along to convert these six steps into action.

Step 1

Begin by cloning the source repository and switching to the work directory:

```
git clone https://github.com/xxxxxx/spring-petclinic.git
cd spring-petclinic
```

Step 2

Create a dev environment with Maven to build this application:

```
docker run -idt -p 9091:8080 --name dev schoolofdevops/maven:spring bash
docker ps -l
```

Step 3

Copy over the source to the container:

```
docker cp . dev:/app
docker exec -it dev bash
```

Step 4

By following the above command sequence, you started the container and logged into it using the **exec** command. Now, build the application and test run it with:

```
cd /app
mvn spring-javaformat:apply
mvn package -D skipTests
mv target/spring-petclinic-2.3.1.BUILD-SNAPSHOT.jar /run/petclinic.jar
```

Step 5

Test the application by actually launching it inside the container:

```
java -jar /run/petclinic.jar
```

This should start running the application. You have already mapped port **8080** on the dev container to **9090** on the Docker host. You can now access and verify the application is running by using one of the links below:

- http://localhost:9091 (if using Docker Desktop)

- http://DOCKERHOST_IP:9091 (use IP/hostname of Docker host if created using a VM, or a cloud server).

Step 6

So far you started with a base image, added your application and tested it by running it. The dev container that you created contains the application that is built and ready. Now, you can commit the container's changes into an image by running:

```
docker container commit dev <dockrhub user id>/petclinic:v1
```

where,

- **`dev`**: is the name of the container that you created to build the application.
- **`<dockrhub user id>/petclinic:v1`**: is the image tag (ensure you replace **`<dockrhub user id>`** with your registry, e.g. DockerHub, ID).

***NOTE**: Credentials are not validated at the time of committing the image which is local, but when you try to push it to the registry.*

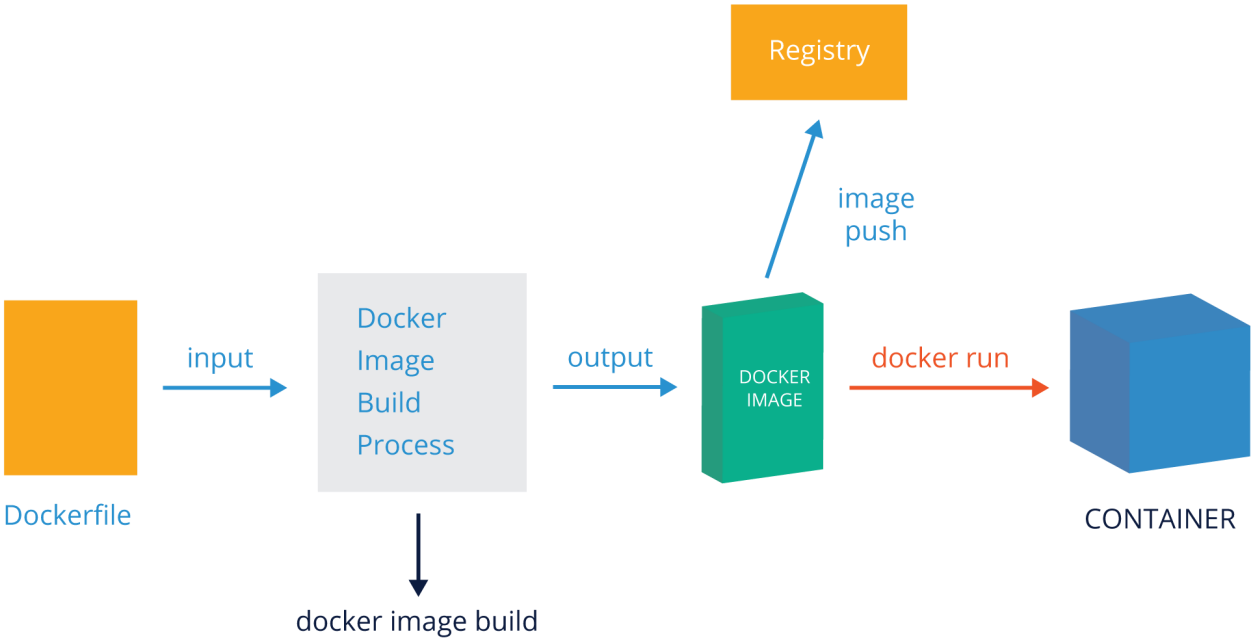
You can list examine the image created with the following commands:

```
docker image ls
docker image history <dockrhub user id>/petclinic:v1
```

Using Dockerfile to Automate Image Builds

Docker provides a way to codify the image build process. Think of it as converting the above process into code using a declarative language.

Following diagram depicts the use of Dockerfile:



- Dockerfile is fed to the image build process, which is run with the **`docker image build`** command.
- The **`docker image build`** command uses Dockerfile as an input, reads each of the instructions, and based on that builds the image, one layer at a time.
- The image built is stored locally, and can be tagged and pushed to a registry.
- It can then also be used to launch a container.

Go ahead and create a Dockerfile for this project by adding the following content to it:

File: spring-petclinic/Dockerfile

```
FROM schoolofdevops/maven:spring

WORKDIR /app

COPY .  .

RUN mvn spring-javaformat:apply && \
   mvn package -DskipTests && \
   mv target/spring-petclinic-2.3.1.BUILD-SNAPSHOT.jar /run/petclinic.jar

EXPOSE 8080

CMD java  -jar /run/petclinic.jar
```

To build an image with this Dockerfile use the following command:

```
docker image build -t <dockrhub user id>/petclinic:v2 .
```

Try building it again:

```
docker image build -t <dockrhub user id>/petclinic:v2 .
```

This time, it does not build anything, but instead uses cache.

Test the image:

```
docker container run --rm -it -p 9091:8080  <dockrhub user
id>/petclinic:v2
```

Publishing Images to the Registry

Now that you have built the images, it's time to publish them to the registry. In order to do that, you need to:

- Authenticate with the registry
- Be authorized to publish an image to the user/organization you have used in the tag. For example, if you try to push an image with the tag **`schoolofdevops/petclinic:v2`**, you need to have access to the "schoolofdevops" organization/user account. This is why tagging your images properly before you attempt to push them is so important!

You can also tag one of your images as **`latest`**, because **`latest`** is not an implicit tag and it won't be automatically set/updated. Instead, you have to set it by yourself by pointing it to one of the image versions. The following commands take you through the process of authenticating, tagging and publishing your images to the registry:

Try to push the image:

```
docker image push <dockrhub user id>/petclinic:v1
```

Log in to Docker Hub with your Docker ID:

```
docker login
```

Push image with v1 version:

```
docker image push <dockrhub user id>/petclinic:v1
```

Tag v2 as **`latest`**:

```
docker image tag <dockrhub user id>/petclinic:v2 <dockrhub user
id>/petclinic:latest
```

Push images with the **`latest`** tag only. Earlier, this used to push all tags, but the behavior has changed to push only those tagged as **`latest`**:

```
docker image push <dockrhub user id>/petclinic:v2
docker image push <dockrhub user id>/petclinic
```

By the end of this process, you should see the images appear in the registry. If you are using Docker Hub, you can go to the web console and check the presence of your images in the image listings.

Decoding Dockerfile Syntax

Dockerfile has its own syntax of the following type:

```
INSTRUCTION arguments
```

Some of the key instructions in Dockerfile can be found below:

- FROM
  It defines the base image.
- WORKDIR
  It defines the directory from which subsequent commands are run (e.g. COPY, CMD, ENTRYPOINT, etc.).
- COPY
  It is used to copy files to containers.
- RUN
  It runs a command, typically installing something inside the container or building an application.
- ENV
  It defines environment variables which are available while building and running containers using the **`image build`** command.
- EXPOSE
  It defines which port your application will listen to.
- ENTRYPOINT
  It defines a command or a script to run before launching the actual command/application.
- CMD
  It defines the command/application/process to start while launching a container with the **`image build`** command with this Dockerfile.

DOCKERFILE INSTRUCTIONS

FROM
Base image used for the build.

WORKDIR
Current working directory to change to.

COPY
Copy files / source code to container.

ADD
Add remote files to container.

RUN
Run a command inside build container.
Installs, builds apps.

CMD
Command to launch with docker run.

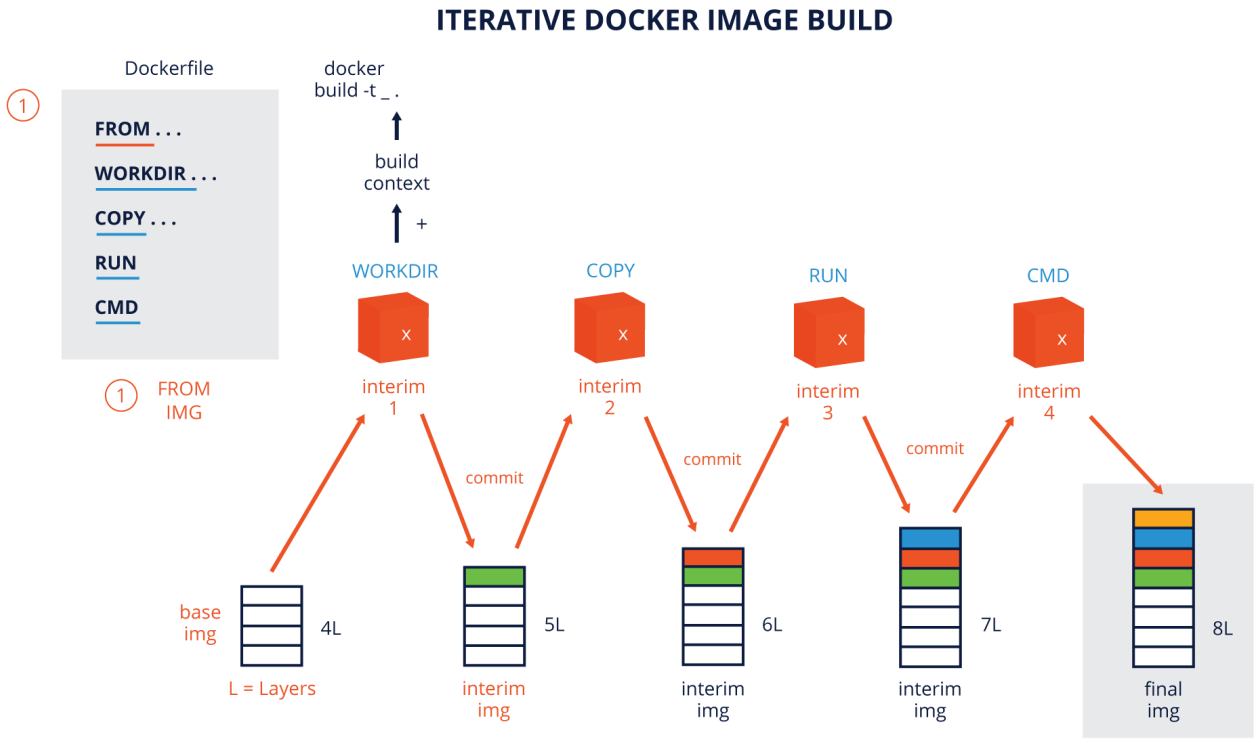
ENTRYPOINT
Run this after launching a container, before CMD.

ENV
Environment variables to use during built and add to container.

EXPOSE
Ports to expose.

Iterative Image Build with Dockerfiles Explained

Even though **`docker build`** follows a similar sequence of tasks as the manual test build above, it takes an iterative approach towards building this image as shown in the image below:



When you launch the **`image build`** command this is what happens:

1. Docker reads the FROM instruction and using the image defined with it as a base, it launches the first intermediate container.
2. Docker copies all the files in the build context to the Docker daemon and subsequently to the intermediate container. These files are then available throughout the build process as this container gets committed into the intermediate images for each step.
3. Docker daemon reads the next instruction (just one) in the Dockerfile and takes action based on it, e.g. copies a file, runs commands, defines metadata, etc.
4. Immediately after that, it commits the changes into an image, and deletes the intermediate container created earlier. This is what adds a new layer to the image.
5. This process is then repeated until all instructions in the Dockerfile are processed, one at a time, creating and deleting intermediate containers and building intermediate images.
6. At the end, the image is tagged with the option that you provided with the **`-t`** flag to the **`docker build`** command.

Summary

In this lab, you learned two ways to create container images, one using a manual approach and another one using Dockerfile. You also learned the art of writing Dockerfiles - what instructions to use, what does each instruction do, and how to write optimized Dockerfiles. Last but not least, you got to see how to work with the container image registries, how to tag and publish images, etc.