Containerizing a Spring Boot App

1. Introduction to containerization and Docker
2. Understanding Docker images
3. A closer look at images and containers
4. How to containerize a Spring Boot app

Demo project: **`demo-containerization`**

1. Introduction to Containerization and Docker

- What is containerization?
- Containers vs. virtual machines
- Docker editions and platforms
- Downloading and Installing Docker for Windows
- Starting Docker for Windows

What is Containerization?

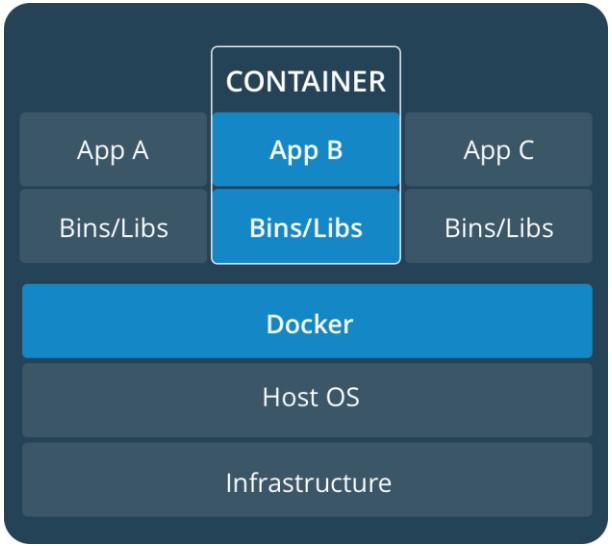
- Containerization is a way of wrapping an application, plus its environment, into a shrink-wrapped container
  - Makes it easy to deploy and run the application, because it runs in a virtualized environment

- Docker is a very popular containerization tool
  - You build an **image** that contains your app, properties, etc.
  - You then run the image - a running image is called a **container**

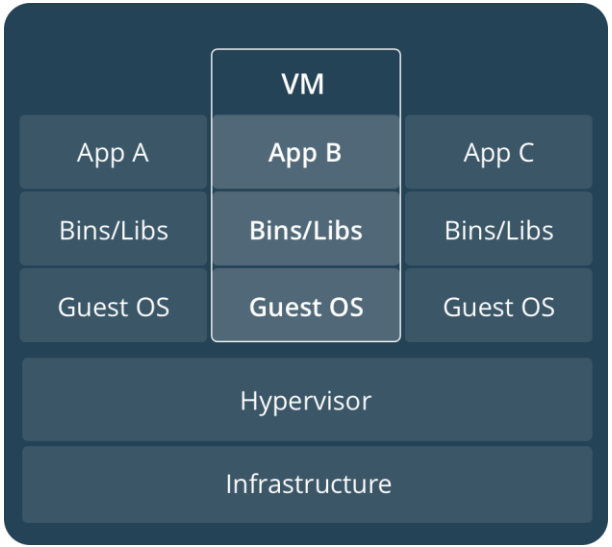
Containers vs. Virtual Machines

- Containers are much more lightweight that VMs
  - Containers run on top of the host OS, e.g. Linux
  - VMs are much bulkier because they actually contain a guest OS

Docker containers



Virtual machines



Docker Editions and Platforms

- Docker comes in two editions
  - Docker Community Edition (CE)
  - Docker Enterprise Edition (EE)

- You can install Docker on various platforms
  - Docker for Linux
  - Docker for Windows
  - Docker for Mac

- We'll be using Docker CE for Windows
  - Requires 64bit Windows 10 Pro/Enterprise/Education
  - For other versions of Windows, use Docker Toolbox instead

Downloading and Installing Docker for Windows

- Browse to:
  - https://hub.docker.com/editions/community/docker-ce-desktop-windows
- Click **Get Docker Desktop**, to download the installer
- When the installer has completely downloaded, run it
- When the installation is complete, click **Close**

Starting Docker for Windows

- To start Docker for Windows:
  - Hit the Windows button, and run **Docker Desktop** as administrator

- Give Docker a few minutes to start, then test it's working like so:
  - Open a Command Prompt window.
  - Run the command **docker version**
  - It should display a message indicating Docker is running

2. Understanding Docker Images

- Overview
- Images vs. containers
- Running a sample image
- Listing images in the local Docker registry

Overview

- A Docker **image** is a black box executable package
  - It includes everything needed to run an application

- E.g. a Docker image for a Java microservice might have:
  - A JVM
  - A web server (e.g. Tomcat)
  - Any additional JARs necessary, e.g. database drivers
  - A JAR containing your REST service

- In this section we're going to see how to download ("pull") and run a pre-built image from Docker Hub

Images vs. Containers

- When you run a Docker image…
  - Docker creates an in-memory instance of the image
  - This in-memory instance is called a **container**
  - You can run many container instances for an image, if you like

Running a Sample Image (1 of 3)

- Docker has a sample pre-built image called `hello-world`
  - You can run it as follows:

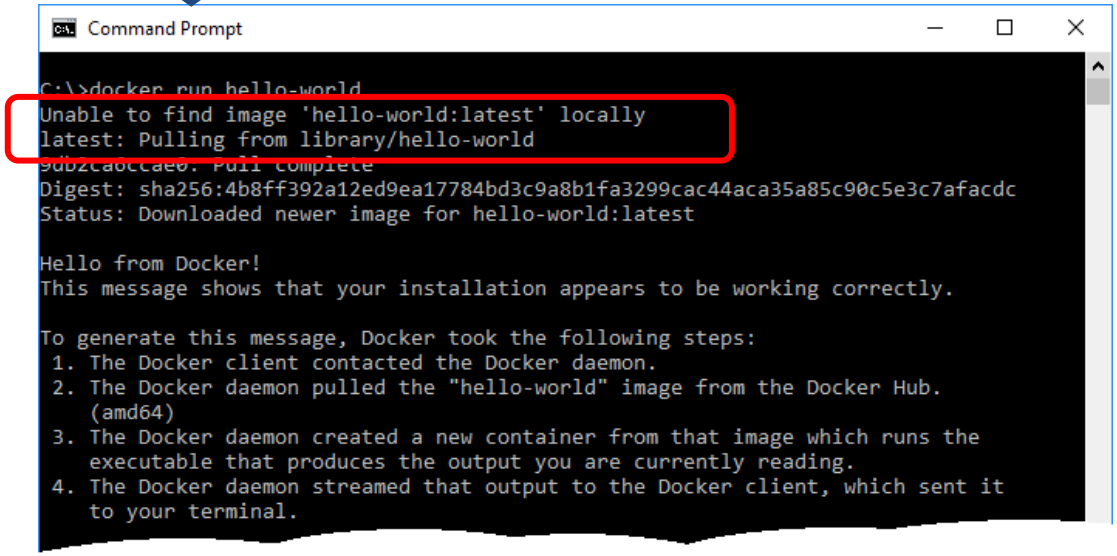
```
docker run hello-world
```

- Docker looks for the image in the local registry
  - The default location for images is `/var/lib/docker`
- If the image isn't in the local registry…
  - Docker pulls it from a global registry (e.g. Docker Hub)
  - Docker stores the downloaded image in the local registry
- Docker then runs the image
  - i.e. it creates a container, a running instance of the image

Running a Sample Image (2 of 3)

- Here's a screenshot of what happens
  - Note in particular, the "pull" request near the top

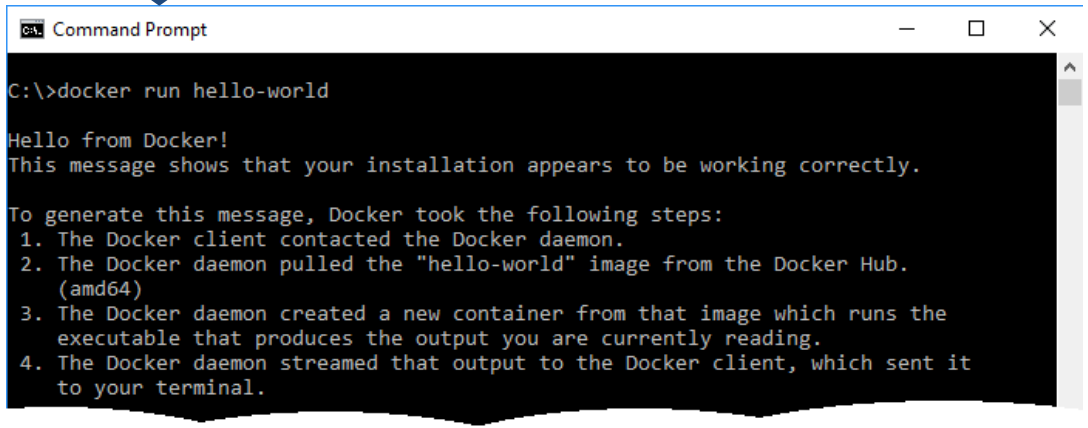
```
docker run hello-world
```



Running a Sample Image (3 of 3)

- Do another Docker run, and see what happens
  - Note there's no "pull" request this time – why not…?

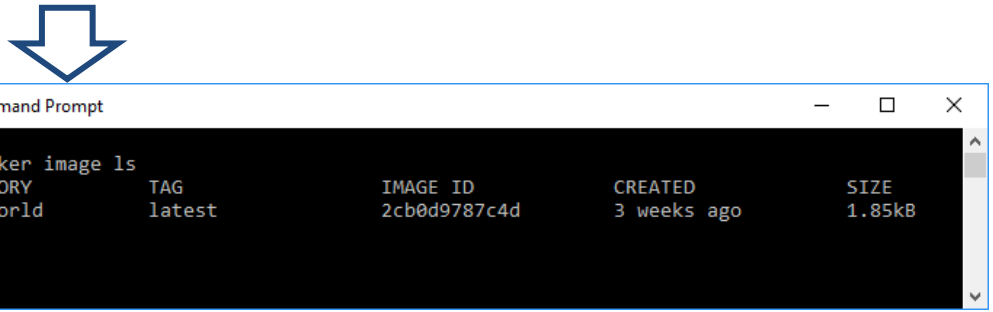
```
docker run hello-world
```



Listing Images in the Local Docker Registry

- You can get a list of all the Docker images in your local Docker registry, as follows:

```
docker image ls
```



3. A Closer Look at Images and Containers

- The power of containerization
- Running multiple containers from an image
- Running containers in detached mode
- Listing containers
- Stopping a container
- Pruning containers and images

The Power of Containerization (1 of 2)

- Docker Hub contains thousands of useful images, providing containerized shrink-wrapped functionality
  - E.g. Tomcat, MySQL, MongoDB, etc.

- E.g. run this command to download and run Tomcat

```
docker run -p 8123:8080 tomcat
```

- This downloads the Tomcat image into your local registry, and creates an instance of the image (i.e. a container)
  - Tomcat runs inside the container
  - Within the container, Tomcat listens on port 8080 by default
  - You can map it to any port on our computer, e.g. 8123 here

The Power of Containerization (2 of 2)

- You can ping Tomcat from your host computer
  - Specify port 8123

```
curl http://localhost:887
```

- Docker maps the request to port 8080 within the container, which means Tomcat handles the request

Running Multiple Containers from an Image

- You can easily spin up another Tomcat container
  - Tomcat will run on port 8080 within that container
  - You must map it to a different port in your host O/S

```
docker run -p 8246:8080 tomcat
```

- You can ping this instance of Tomcat from your host computer
  - Specify port 8246

```
curl http://localhost:8246
```

Running Containers in Detached Mode

- You can run a container in “detached mode”
  - Specify the `-d` option
  - The container runs in the background

```
docker run -d -p 8369:8080 tomcat
```

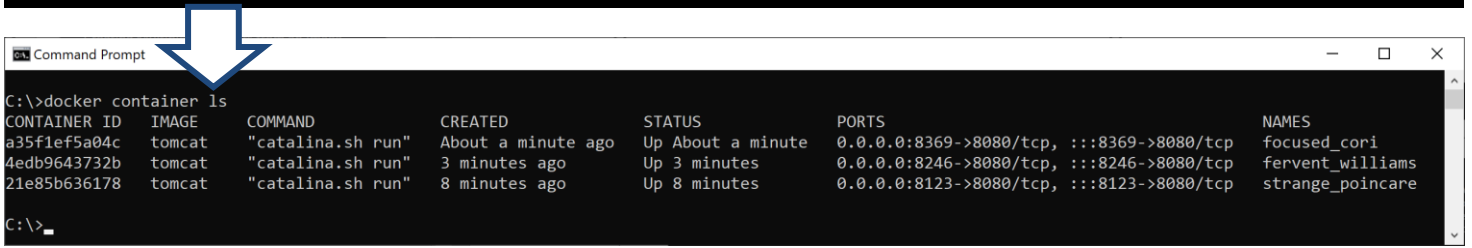
- You can ping this instance of Tomcat from your host computer
  - Specify port 8369

```
curl http://localhost:8369
```

Listing Containers

- You can get a list of all the containers currently running

```
docker container ls
```



- Each container has:
  - A unique container id (abbreviated)
  - The name of the image (of which this container is an instance)
  - The command that is executed within the container
  - Created timestamp and status
  - Port mappings
  - A name for the container (random name by default)

Stopping a Container

- To stop a container, run the following command with the container ID or name you want to stop

```
docker container stop focused_cori
```

- Even after you stop a container, Docker maintains information about that container (e.g. so you can view its logs)
  - You can list all containers (including stopped ones) as follows:

```
docker container ls -a
```

Pruning Containers and Images

- To completely remove all stopped containers:

```
docker container prune
```

- To completely remove all dangling images:

```
docker image prune
```

4. How to Containerize a Spring Boot App

- Overview
- Running the application as normal
- Bundling the application in a JAR
- Defining a Dockerfile
- Understanding the Dockerfile
- Building the image
- Viewing images in the local Docker registry
- Running a container

Overview

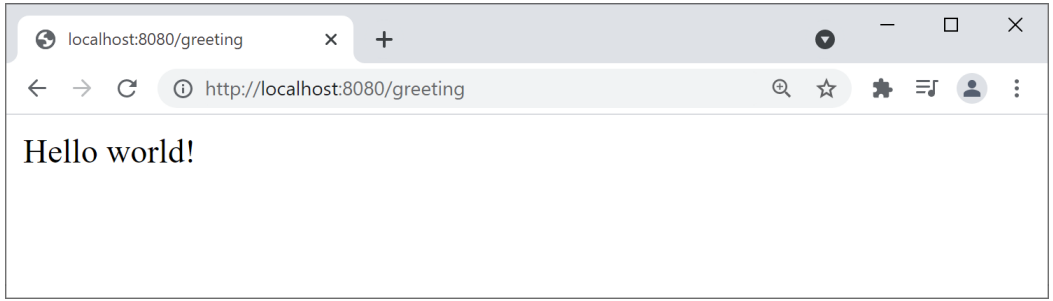
- In this section we'll see how to containerize a Spring Boot app
  - We'll build a Docker image that contains a Spring app
  - Then we'll run a container (i.e. an instance of the Docker image)
  - Our Spring Boot app will run on a JVM inside the container

- See demo project, `demo-containerization`
  - Take a look at `Application.java`
  - It's a simple Spring Boot app with a REST service

Running the Application as Normal

- You can run the application as normal
  - Right-click `Application.java`, then Run Application

- This runs the application directly on your host computer
  - The application contains an embedded web server (Tomcat)
  - Tomcat listens on port 8080 on your host computer
  - You can ping it via `http://localhost:8080/greeting`



Bundling the Application in a JAR

- If you want to run a Java app in a Docker container…
  - The first step is to bundle the app into a JAR file

- To bundle the app into a JAR:
  - Open a Terminal window in the project root folder
  - Run the following Maven command

```
./mvnw package -DskipTests
```

- This creates the JAR file:
  - `target/demo-containerization-0.0.1.jar`

Defining a Dockerfile (1 of 2)

- Now we're ready to see how to create a Docker image
  - Remember, a Docker image is a “black box” executable package
  - It includes everything needed to run an application

- In our case, we'll create a Docker image containing:
  - A JVM
  - Our Spring Boot JAR file
  - A command to execute the Spring Boot JAR file on the JVM

Defining a Dockerfile (2 of 2)

- In order to define a Docker image, define a special file named `Dockerfile` (by default)
  - Specifies build instructions, so Docker can build an image

- See this `Dockerfile` in the demo project (root folder)

```
FROM openjdk:17
ARG JAR_FILE
COPY ${JAR_FILE} myapp.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/myapp.jar"]
```

- See following slides for an explanation
  - Also see https://docs.docker.com/engine/reference/builder/

Understanding the Dockerfile (1 of 3)

```
FROM openjdk:17
ARG JAR_FILE
COPY ${JAR_FILE} myapp.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/myapp.jar"]
```

- A Dockerfile starts with a `FROM` instruction
  - Specifies the "base image" from which we are building
  - Our image will be based on OpenJDK version 17
  - OpenJDK is an open-source implementation of Java SE

- When we run this Dockerfile to build our image…
  - Docker will see if we've already downloaded `openjdk:17`
  - If we haven't, Docker will pull it from the Docker Hub

Understanding the Dockerfile (2 of 3)

```
FROM openjdk:17
ARG JAR_FILE
COPY ${JAR_FILE} myapp.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/myapp.jar"]
```

- When you build a Docker image, you can pass arguments into the Docker build command
  - In the Dockerfile, use `ARG` statements to capture these arguments
  - In our example, the `JAR_FILE` arg specifies the name of our JAR

- A Dockerfile specifies files to copy into the Docker image
  - Use `COPY` instructions to copy files into the Docker image
  - In our example, we copy our JAR file into the image
  - Inside the image, the JAR file will be named `myapp.jar`

Understanding the Dockerfile (3 of 3)

```
FROM openjdk:17
ARG JAR_FILE
COPY ${JAR_FILE} myapp.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/myapp.jar"]
```

- The `EXPOSE` instruction:
  - Acts as documentation about port(s) inside the container
  - Indicate this port must be mapped to a port on the host

- The `ENTRYPOINT` instruction:
  - Specifies what to actually execute inside the Docker image
  - In our example, we run our JAR on the JVM in the image

Building the Image

- Use the `docker build` command to build a Docker image
  - Type the following **all on one line**
  - It reads and executes the instructions in the Dockerfile

```
docker build -t my-spring-boot-app
             --build-arg JAR_FILE=target/demo-containerization-0.0.1.jar
             .
```

- `-t`
  - Specifies the tag name for the image
  - Tells Docker to create an image with this name in the local registry

- `--build-arg`
  - Specifies a value for a build argument
  - Followed by a name=value pair

Viewing Images in the Local Docker Registry

- You can view images in the Docker registry

```
docker image ls
```

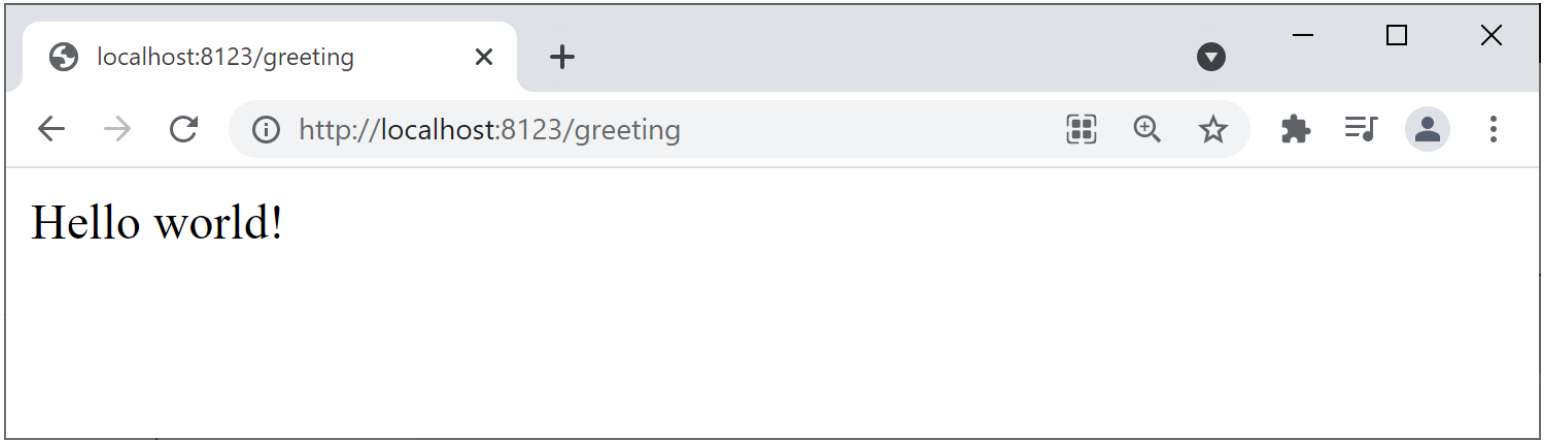
```
Command Prompt

C:\>docker image ls
REPOSITORY           TAG       IMAGE ID       CREATED          SIZE
my-spring-boot-app   latest    5ef351d00100   46 seconds ago   677MB
hello-world          latest    feb5d9fea6a5   4 days ago       13.3kB
tomcat               latest    bb832de23021   2 weeks ago      680MB

C:\>
```

Running a Container

- You can run a container as normal

```
docker run --name app -d -p 8123:8080 my-spring-boot-app
```

- You can then ping as normal, via the mapped port



Summary

- Introduction to containerization and Docker
- Understanding Docker images
- A closer look at images and containers
- How to containerize a Spring Boot app

Exercise

- Modify the Spring Boot demo app (e.g. add an `index.html` file) and then rebuild the JAR file:

```
./mvnw package -DskipTests
```

- Forcibly remove the old container and image:

```
docker container rm -f app
```

```
docker image rm -f my-spring-boot-app
```

- Rebuild the image and run another container:

```
docker build -t my-spring-boot-app
             --build-arg JAR_FILE=target/demo-containerization-0.0.1.jar
             .
```

```
docker run --name app -d -p 8123:8080 my-spring-boot-app
```