Introduction to Docker

Difficulty Level: Beginner (no prior Docker experience is assumed)

Hands-On Session

Time: approximately 40 minutes

In this session, we will cover some basic Docker commands and a simple build-ship-run workflow. We will start by running some containers, then we will build a Docker image of an application. Finally, we will look into an approach to modify a running application in the container.

Step 0: Set-up

- Create a Docker ID: https://docs.docker.com/docker-id/#log-in

- You can install Docker on your computer, or you may use it on a VM in the cloud. You can also use the Docker online platform by going to the following link: https://tinyurl.com/y9sg7eq7

- Information and instructions on how to install Docker can be found at these links:
  https://docs.docker.com/get-docker/
  https://runnable.com/docker/install-docker-on-linux

  DockerHub account instructions can be found here:
  https://docs.docker.com/docker-id/

Step 1: Run some pre-built Docker containers - first alpine, and then ubuntu

- `docker container run alpine date`
- `docker container ls --all`
- `docker container run --interactive --tty alpine /bin/sh`
- `ls`
- `exit`
- `docker container ls --all`
- `docker start -a -i <containerid>`
- `docker container run --interactive --tty --rm ubuntu /bin/bash`
- `exit`
- `docker container ls --all`
- `docker container run --interactive --tty --rm ubuntu bash`
- `bash --version`
- `ls > test`
- `cat test`

Commented [1]: This is the command that will be run in the alpine container. Once the command is run, the container will exit. This example, though very simple, demonstrates that one can have the configuration scripts for their software distributed as a Docker container - people do not need to know what those scripts are - they can just run the container built to use those scripts.

Commented [2]: Docker keeps a container running as long as the process it started inside the container is still running. In this case the process running the command "date" exits as soon as the output is written. This means that the container will stop. This container will not be deleted though and will exist in the "Exited" state.

Commented [3]: start a stopped container, and interactively attach to it

Commented [4]: start an interactive container using ubuntu image and remove the container upon exit

Commented [5]: These two options help in providing an interactive session for "bash"

Commented [6]: This flag will help in removing the container when it is done executing the process "bash"

Commented [7]: We can have multiple independent containers on the same host

Commented [8]: you can now run commands inside the container - example, check the version of the bash command

- `exit`

Commented [9]: these are regular Linux commands

- `docker start -a -i <containerid>`

Commented [10]: Use the id of the container associated with ubuntu. Notice the difference between attaching to this container versus attaching to the previous alpine container.

Step 2: Package and run an application using Docker

- First, let us create a Dockerfile. Add the following contents to a file named Dockerfile:

Commented [11]: To create the file, you can use the "vi" command - vi Dockerfile - and then press "i" to start inserting text in the file. To save and exit, press "esc" and type "wq!".

```
FROM alpine

CMD [“echo”, “hello world!”]
```

Commented [12]: alpine is a lightweight Linux OS. Here, we are using the alpine image as our base image on top of which we will build our Docker container. If this image is not present locally, it will be fetched from Docker Hub (if you are connected online).

- Execute the Docker build command to create a Docker image:

```
docker build .
```

Commented [13]: Don't forget the "." at the end of the build command

- Run the docker Docker image -> and get a live container:

```
docker run --name hello_world 47520815f314
```

Commented [14]: This is the id of the container image that was created in the build step. You will have a different id for your image - check the output of the build command to get the image id.

```
docker start -a -i hello_world
```

Commented [15]: For running the docker container again, we are using the name that we gave to the docker container "hello_world" instead of using the ID

- To see the running images:

```
docker images
```

- To see the running processes:

```
docker ps
```

Commented [16]: this command will not show any running container for this exercise because our container exited immediately after printing "hello world".

Step 3: Adding Volumes

You can use the technique of mounting volumes in your Docker container in case you would like to modify a running application in a Docker container without having to run the Docker build command again or make the data available on the host server inside the Docker container. The volumes can persist even if the container itself is deleted.

Exercise 1:

- Create a Dockerfile

```
FROM ubuntu

RUN mkdir -p ubuntu1 && cd ubuntu1 && echo "hello hello bye bye" >> file

VOLUME /ubuntu1

CMD /bin/sh
```

- Build the image

```
docker build .
```

- Run the container

```
docker run --rm -it d04047740d42
```

Commented [17]: docker container id - will be different for you

- List files and directories inside the container - you will notice “ubuntu1” is created

Commented [18]: This volume is visible from the host system too - run the command "docker volume ls" to see the active volumes, and run the command "docker volume inspect <volume id>" to look at the contents of the volume

```
ls
```

- Exit from the container

```
exit
```

Exercise 2:

- Create a directory named “src”

```
mkdir src
```

- Switch to the directory “src”

```
cd src
```

- Create a file named “Hello.java” and paste the contents below in it:

```
public class Hello { public static void main(String... ignored) {
System.out.println("Hello, World!"); } }
```

- Change directory

```
cd ..
```

- Create a Dockerfile

```
FROM openjdk:8u131-jdk-alpine

WORKDIR /src

ENTRYPOINT javac Hello.java && java Hello
```

Commented [19]: WORKDIR: this instruction sets the working directory for the RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile.

Commented [20]: ENTRYPOINT: allows you to configure a container that will run as an executable.

- Build the Docker image

```
docker build -t my-openjdk .
```

- Run the command below to create a container with a volume mounted

Commented [21]: In the previous exercise we had "VOLUME" mentioned inside the Dockerfile

```
docker run --rm -it -v $(pwd)/src:/src my-openjdk
```

Commented [22]: Container and the volume will be cleared after the command inside the Docker file executes - the command runs the Java code to print "Hello World".

Commented [23]: Flag for volume mount

Commented [24]: absolute path of the directory named "src" on the host system

Commented [25]: absolute path in the container

Commented [26]: image name

- Edit the file named “Hello.java” inside the “/src” directory on the host server:

```
public class Hello { public static void main(String... ignored) {
System.out.println("Hello, World from GHC18!"); } }
```

- Run the docker command below - you will notice that without building the image again, we are able to see the changes made in the Hello.java file on the host server

```
docker run --rm -it -v $(pwd)/src:/src my-openjdk
```