ECGR 4090/5090 Cloud Native Application Architecture
Lab 7: Docker

Virtualization of compute resources is key to decoupling physical hardware from the applications. One option to virtualization is through the use of Virtual Machines (VMs such as Virtualbox) by which virtual machines dedicated to an application can be spun up. While VMs are well isolated from each other, they are resource intensive, and have long startup times. An alternative solution is virtualization at the OS level known as containers. Containers can be thought of as isolated instances of a normal OS process. They are lightweight since they share the OS kernel with other containers. A large number of containers can be spun up on a single physical machine rapidly. They provide an isolated view of the processes within the container, the file system, and the network. Linux kernel supports containers through features such as namespaces, and cgroups, allowing physical resources (CPU and memory) to be partitioned among containers.

Docker is the most popular of the container technologies. As you will see in this lab, Docker allows you to package your application along with the environment needed to run it on a host. This allows applications to be easily migrated between physical machines. Also, you can easily share applications with other people. From the Docker docs (https://docs.docker.com/get-started/overview/)

Docker provides tooling and a platform to manage the lifecycle of your containers:

- Develop your application and its supporting components using containers.
- The container becomes the unit for distributing and testing your application.
- When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

Here's how they are used in the Cloud Native world (from Docker docs)

- Developers write code locally and share their work with their colleagues using Docker containers.
- They use Docker to push their applications into a test environment and execute automated and manual tests.
- When developers find bugs, they can fix them in the development environment and redeploy them to the test environment for testing and validation.
- When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.

Interestingly, Docker itself is written in Go!

In short, containers are the units in which applications are developed and deployed in production in the Cloud native world.

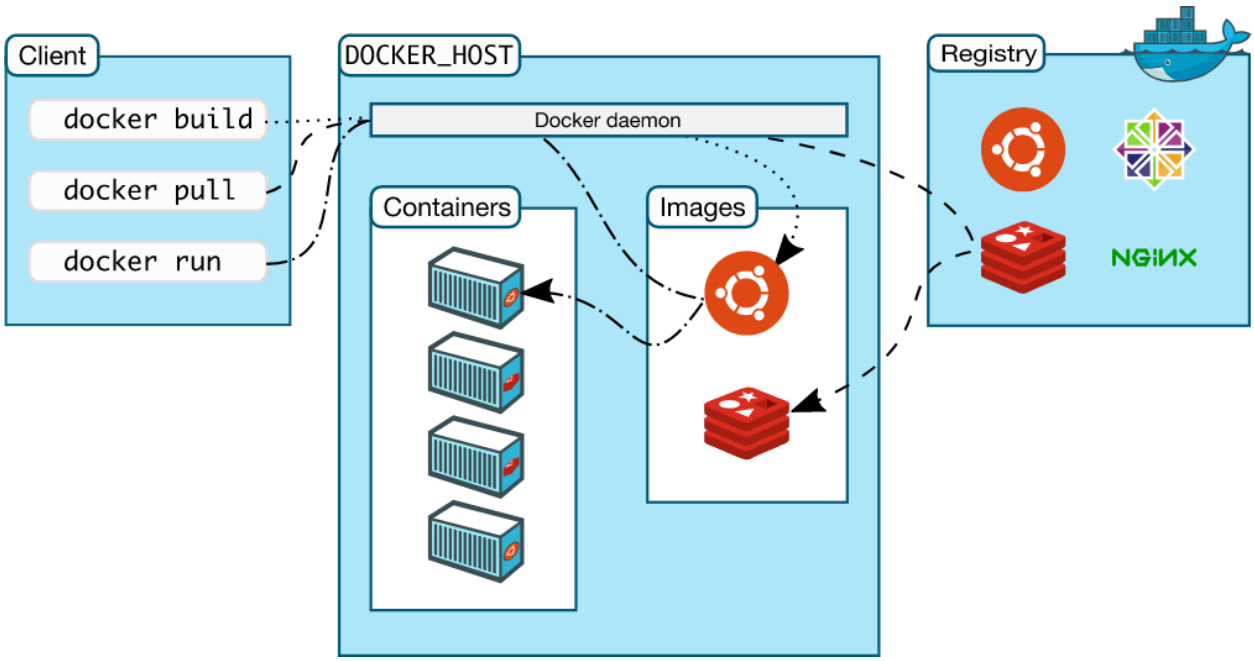
In this lab, we will install Docker, get familiar with basic Docker commands, and deploy the Go web server from Lab 4 in a Docker container.

Installation

$ *sudo apt-get update*
$ *sudo apt install docker.io*
// Verify
$ sudo docker run hello-world
If you get the message “Hello from Docker!” you are good to go.

Docker architecture
Key terms are bolded. Make sure you understand them.
(From https://docs.docker.com/get-started/overview/)

Docker uses a client-server architecture. The **Docker client** talks to the **Docker daemon**, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. As a user you interact with the Docker client.



Docker images is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the Ubuntu image, but install the Apache web server and your application, as well as the configuration details needed to make your application run.

To build your own image, you create a **Dockerfile** with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

A Docker container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the **Docker API or CLI**. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

Docker basics

Let's spin up a Docker container

$ sudo docker run -it ubuntu /bin/bash

When you run this command, the following happens (assuming you are using the default registry configuration):

- If you do not have the Ubuntu image locally, Docker pulls it from your configured registry, as though you had run docker pull ubuntu manually.
- Docker creates a new container, as though you had run a docker container create command manually.
- Docker allocates a read-write filesystem to the container, as its final layer. This allows a running container to create or modify files and directories in its local filesystem.
- Docker creates a network interface to connect the container to the default network, since you did not specify any networking options. This includes assigning an IP address to the container. By default, containers can connect to external networks using the host machine's network connection.
- Docker starts the container and executes /bin/bash. Because the container is running interactively and attached to your terminal (due to the -i and -t flags), you can provide input using your keyboard while the output is logged to your terminal.
- When you type exit to terminate the /bin/bash command, the container stops but is not removed. You can start it again or remove it.

If everything ran successfully, you are in the bash shell of the container. List the files in the container
 # ls -l

Try out these Docker commands -

In another terminal
list the containers running in your system. Note the name of the container. You will see the container names listed. You can assign your own name when you run the container (recommended)

$ sudo docker container ls

To find out more information about the container
$ sudo docker inspect <container name>

To stop the container
$ sudo docker stop <container name>
To start a container
$ sudo docker start <container>

To exit the container (from the container shell)
 # exit

List Docker images on your system
$ sudo docker images

Delete docker image
$ sudo docker image rm <image name OR image id>

Here's a useful command cheat sheet from docker
https://www.docker.com/sites/default/files/d8/2019-09/docker-cheat-sheet.pdf

Building Go application Docker containers from Dockerfile

Docker builds images automatically by reading the instructions from a Dockerfile -- a text file that contains all commands, in order, needed to build a given image. Let's build a Docker file to containerize the webserver application from Lab 4

In a new directory (say, lab7) copy webserver.go from Lab 4

Important: Modify the listening port in the code to ":8000" (instead of "localhost:8000)

Now create a file named Dockerfile with the following content

```
FROM golang:1.15-alpine AS build

WORKDIR /src/
COPY webserver.go /src/
RUN CGO_ENABLED=0 go build -o /bin/webserver

FROM scratch
COPY --from=build /bin/webserver /bin/webserver
ENTRYPOINT ["/bin/webserver"]
```

Alpine is a tiny Linux distribution for containers. The golang:alpine image has the tools and libraries needed to compile Go programs. We use this as the base image.
We instruct Docker to create a src directory in the container and copy the webserver.go program from our machine to the container. We then instruct it to build a binary executable.

To save memory we then start from an even smaller *scratch* image without any of the Go tools. We copy the executable and copy the binary to the *scratch* image. This is what is executed when the container is run. This process is called a **multistage build**.

Let's first create the image (note the dot at the end of the command) -

$ sudo docker image build -t webserver .

Check to see if the image is created
$ sudo docker image ls

Note the tiny size of the container (only 6.4 MB on my system). With a full fledged Ubuntu image the size could be around 100 MB or more. Since Cloud Native applications will have hundreds of such containers running, it is important to reduce the image size to the minimum. Also, the smaller image has a less attack surface, and is potentially more secure.

Run the image in a container mapping the container port 8000 to local port 8000
$ sudo docker container run -p 8000:8000 webserver

In another terminal access the containerized webserver
$ curl localhost:8000/list

You should see the same output as in Lab 4.
You have now seen how to containerize an application and expose it to the outside world using a REST API. The container could be uploaded to a DockerHub repository, and downloaded to wherever you need it (for example, on your Cloud machine).

Be sure to update your Gitllab/Github repository if you haven't already done so.

To do -
Containerize the gRPC movie server of Lab 5. The gRPC client can remain on your local machine. This is an alternate way of exposing the containerized application - using RPCs. The gRPC server runs in the Docker container while the client is invoked as in Lab 5. The output is the same as in Lab 5.
Hint 1: The following resource should be helpful -
https://docs.docker.com/language/golang/build-images/
Hint 2: Take care of the package paths. Think about the paths server.go is seeing inside the container.