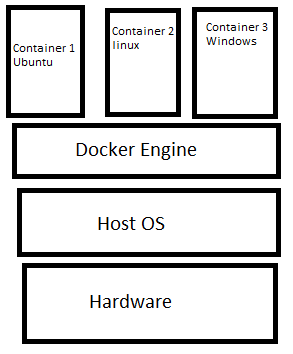
**Docker**

Docker replaces vms. Vms virtualize hardware whereas docker virtualizes os. In docker, you have a single OS (or kernel to be precise), and the resources are shared between the containers. Hence it is lightweight and boots in seconds.

In docker, all of the containers share the same os but it is appearing for each container as if they have their own os.

A container is a way to package application with all the necessary dependencies and configuration and this package is portable, meaning that it can run on any platform. Containers share the same kernel.

Containers live in a special repository called container repository.



Containers are made up of layers. We have linux image layer, application image layer and so on. Images are just binary numbers representing the exact copy of the software.

**Container vs Image**

Container is a running environment for an image. An image is a template for creating an environment of our choice. This could be a database, a web app and etc. An image is a snapshot. You can create multiple snapshots in versions then you can point to a version that you want at a particular time.

So we have an image and from this image we run a container. A container only lives as long as there is a process running inside.

**Pulling an image and running container**

Let’s pull an image from the docker hub and create a container from this image.

After downloading an image, we run a container from the image we pulled. 🡺 docker run nginx:latest

Latest is the name of the tag. To look at the running containers 🡺 docker container ls or docker ps

To run a container in a detached mode so that we don’t just hang in the command line🡺

docker run –d nginx:latest

To stop a contaner 🡪 docker stop containerId or containerName

To remove a container 🡺 docker rm containerId or containerName

To remove all the container from a single command🡺

docker rm –f $(docker ps -aq) – here –f is for forcing removal so that running containers are also removed. –a is to display all containers (running and stopped containers) and –q is for only displaying ids.

**Exposing Port**

We wanna go from the host to the container (we wanna access the container from our browser). In this case we have to expose the port. We use –p 🡺 docker run –d –p 8080:80 nginx:latest

So whenever we type “localhost:8080” we want that to be mapped to port 80 on the container.

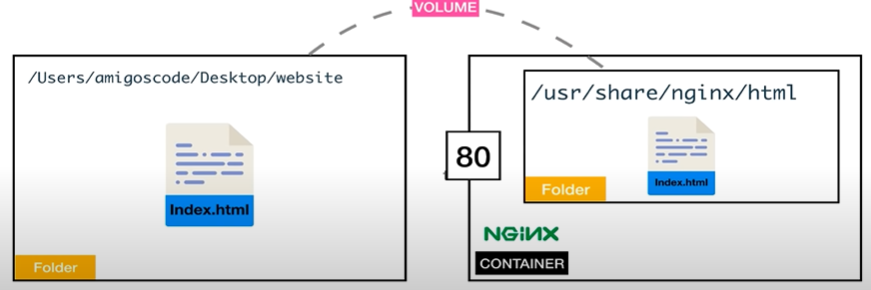
**Container names**

If we don’t specify a container name when we run it, a random name is given. If we want to have a custom container name 🡺

docker run –name containerName –d –p 8080:80 nginx:latest

**Docker Volumes**

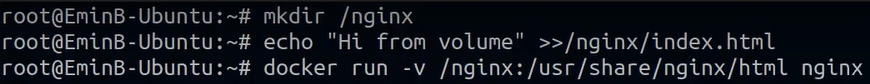
When we have data in a container, once we stop the container the data is gone. So in order to have some persistent data we can take it from the host file system. Volumes allow sharing of data between host and container and also between containers. Files & Folders. A folder in physical host file system is mounted into the virtual file system or docker.



The folder is in the host.



ro- read-only so that it can’t be changed.



We create a ngnix dir and we write “Hi from volume” string with echo and then we mount that file (index.html) into the container.

**Creating custom images**

To create a custom image, we create a Dockerfile. We never build an image from scratch. We always use an existing image as our base image.

From nginx:latest

RUN apt-get update

RUN apt-get install nginx -y

First our base image is nginx. Then update and then install nginx.

**Interactive mode (-it)**

We can get into a running container 🡺 Docker exec –it containerId bin/sh – docker execute in an interactive mode into this container that is running and when you get there execute /bin/sh. If it is alpine then bin/sh for the terminal.

We use this so that we get inside a container and we can run commands inside this container.

When we exit the container we can specify a command which will remove the stopped container because if we use docker ps –a we will see this container. So we use 🡺

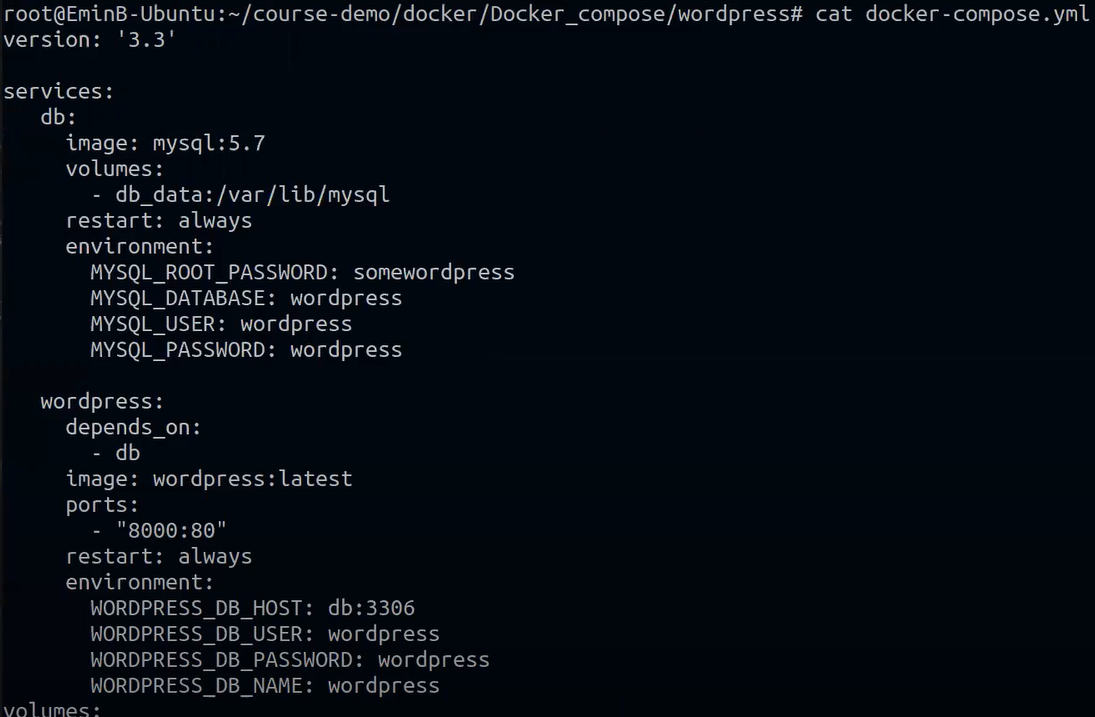
Docker run –it --rm containerId bin/bash Now when we exit from this container it will remove this container.

**Docker start vs docker run**

Docker run creates a new container but “docker start” restarts already stopped container.

**Docker compose**

Docker compose is just a structured way to contain common docker commands. We just don’t want to have all the commands in one line. With docker-compose up we build images and run containers from these images🡺



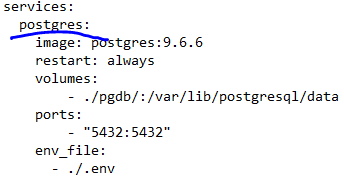
Services are basically our containers which have their own image.

When if want to activate docker-compose we use the following command 🡺 docker-compose –f fileName.yaml up OR if the file name is the same as docker-compose then 🡺 docker-compose up

If we need to stop a running container which was activated from docker-compose we can stop the container manually (docker stop containerId) or with docker-compose down command.

docker-compose build command just builds the images without running containers from them.

**Note that** if we have one container from postgres image and one from .net app then the connection string is not localhost but it is actually the name of the service that we give it to the postgres image🡺



It is gonna be our server name for the connection string. So don’t use localhost but the service name!

**DockerFile**

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. It is a script made up of instructions on how to build a Docker image. In this script, there are two types of instructions that can define the process running in the container:

* **ENTRYPOINT**
* **CMD**

In short, CMD defines default commands and/or parameters for a container. CMD is an instruction that is best to use if you need a default command which users can easily override. If a Dockerfile has multiple CMDs, it only applies the instructions from the last one.

On the other hand, ENTRYPOINT is preferred when you want to define a container with a specific executable. You cannot override an ENTRYPOINT when starting a container unless you add the --entrypoint flag.

Combine ENTRYPOINT with CMD if you need a container with a specified executable and a default parameter that can be modified easily. For example, when containerizing an application use ENTRYPOINT and CMD to set environment-specific variables.

It is important to discus the forms of instructions. Docker ENTRYPOINT and CMD can have two forms:

* Shell form
* Exec form

The syntax for any command in shell form is:

<instruction> <command> 🡺

ENTRYPOINT dotnet RuneCube.dll

The syntax for instructions in exec form is:

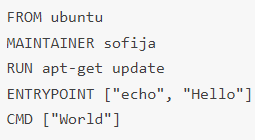
<instruction> ["executable", "parameter"]

CMD ["dotnet", "RuneCube.dll"]

The exec for is the most preferred. It converts that into json which is why we use double quotes not single quotes.

* **CMD echo "Hello World"** (shell form)
* **CMD ["echo", "Hello World"]** (exec form)
* **ENTRYPOINT echo "Hello World"** (shell form)
* **ENTRYPOINT ["echo", "Hello World"]** (exec form)

CMD commands are overridden if we provide extra commands. But with entrypoint you cannot overridde even though you give some extra parameters unless you explicitly use. With entrypoint the commands will be appended. There is a way to override the ENTRYPOINT instruction – you need to add the --entrypoint flag prior to the container\_name when running the command.





If we provide additional parameter then 🡺





So, use entyrpoint if we always want the command to run. Use CMD when we can override some command.