Docker build command with tag:

**Docker build -t sandip2011/simple-web:version .**

Specify file to build image:

**docker build -f <file\_name> .**

Running a built container

**Docker run <image\_name>**

Running a container with port mapping as docker have it network inside container:

**docker run -p {port-request-in-local-machine}:{port-request-in-container} <imag>**

**docker run -p 8000:8080 <image>**

To execute extra command in a running container:

**docker exec -it <container\_id> <command>**

**# specify base image**

**FROM node:alpine**

**#if directory specified in WORKDIR does not exists it will automatically create**

**WORKDIR /app**

**# WORKDIR after specifying this argument any following command**

**#will be executed relative to path specified in WORKDIR**

**COPY ./package.json ./**

**#Install dependencies**

**# Doing only this step will not find package.json file**

**#as the image using 2nd steps consist of filesystem of first**

**#step only**

**#so we need to our local files and directories into container**

**RUN npm install**

**#After we install the dependency we will copy everything else and now cache we be used**

**#until this step**

**COPY ./ ./**

**# setup a command**

**CMD [ "npm", "start" ]**

**#we will split the copy step into two different COPY step**

**# In first step we will copy only the files required to install**

**#dependency**

**#then we will copy other files and folders of the project**

**#so that any change to the file and folder do not have to start whole build process**

**# build process uses caches and whenever it finds changes in file it will re run**

**#from that step without using cache**

Docker Volume

#specify base image

FROM node:alpine

WORKDIR /app

COPY ./package.json ./

#Install dependency

RUN npm install

COPY ./ ./

#command

CMD ["npm", "run", "start"]

Instead of copying directory to container, map directory inside the container to local directory. So that instantaneous change on the source code can be reflected

**docker run -v $(pwd):/app <image\_id>**

**docker run -p 3000:3000 -v $(pwd):/app b6a352c2a88d**

**docker run -p 3000:3000 -v /app/node\_modules -v $(pwd):/app b6a352c2a88d**

Here ,docker volume is Mapping everything on the container /app directory on the local directory $(pwd) except we do not have node\_modules folder on local directory but when we run npm install while building it is created inside the container but is not present in local directory to map into so we need to specify as do not map **-v /app/node\_modules** only map **-v $(pwd):/app (map pwd into app folder)**

We can shortenhand this syntax using docker-compose.yml to configure volumes and ports:

version: '3'

services:

web:

build:

context: .

dockerfile: Dockerfile.dev

ports:

- "3000:3000"

volumes:

- /app/node\_modules

- .:/app

**docker-compose up --build**

Pulling Image From Docker Hub

**docker pull <image\_name>**

Pushing an image into docker hub

* Login in docker hub
* Create repository
* Login to hub through command line

**docker login --username=sandip2011 --email=sandipneupane65@gmail.co**

* Tag the image with repository name in hub

**docker tag imageID Repositoryname**

* Push the image

**docker push repository\_name**

Docker-compose

version: '3'

#services mostly means container

services:

redis-server:

image: 'redis'

node-app:

#build from the DockerFile in current directory

build: .

ports:

- "8000:8081"

# here (-) in yml file represents an array

Docker-compose commands:

**docker-compose up (**similar to **docker run <image\_name>)**

It automatically looks up for the file named docker-compose.yml and starts the services

**docker-compose up --build ( similar to :**

* **docker build .**
* **docker run <image\_name>**

**)**

It builds the the image and runs the container out of those images

Start all container in background

**docker-compose up -d**

Stop Container

**docker-compose down**

**Restart policies docker-compose**

* **“no” (**never start the container if it crashes or stops**)**
* **always**(always start the container if it crashes or stops)
* **on-failure(**only start the container if exited with error code**)**
* **unless-stopped(**Always restart unless forcibly stopped**)**

version: '3'

#services mostly means container

services:

redis-server:

image: 'redis'

node-app:

#build from the DockerFile in current directory

restart: "no"

build: .

ports:

- "8083:8083"

# here (-) in yml file represents an array

**Container status with docker-compose**

**docker-compose ps**

Docker file for nginx

#specify base image

#here as represents the tag every step from below this is tagged as builder phase

FROM node:alpine as builder

WORKDIR /app

COPY ./package.json ./

#Install dependency

RUN npm install

COPY ./ ./

RUN npm run build

#it is going to create build folder in /app/build

ENV CI=true

#to specify next we just need to specify another base image

FROM nginx

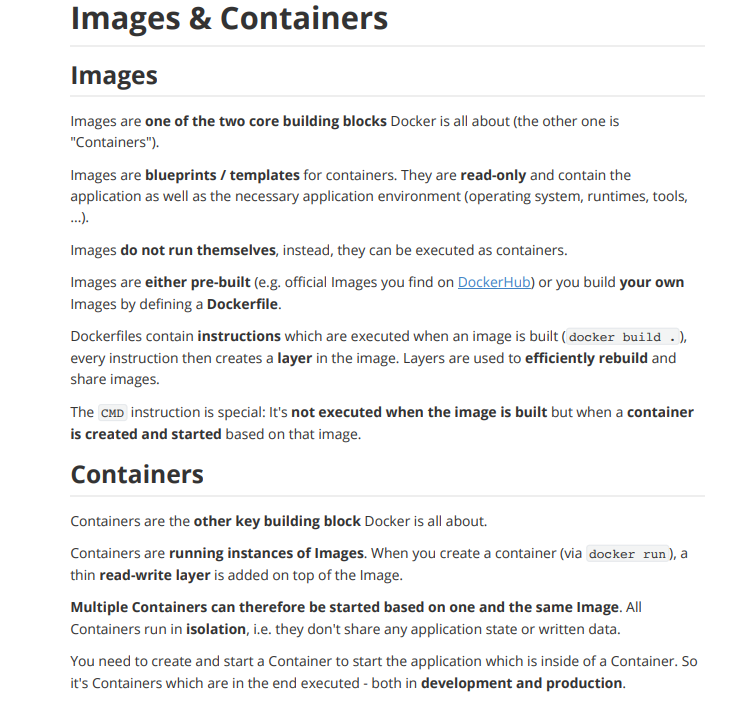
#here we are not using any of the other folders then build we just need this folder

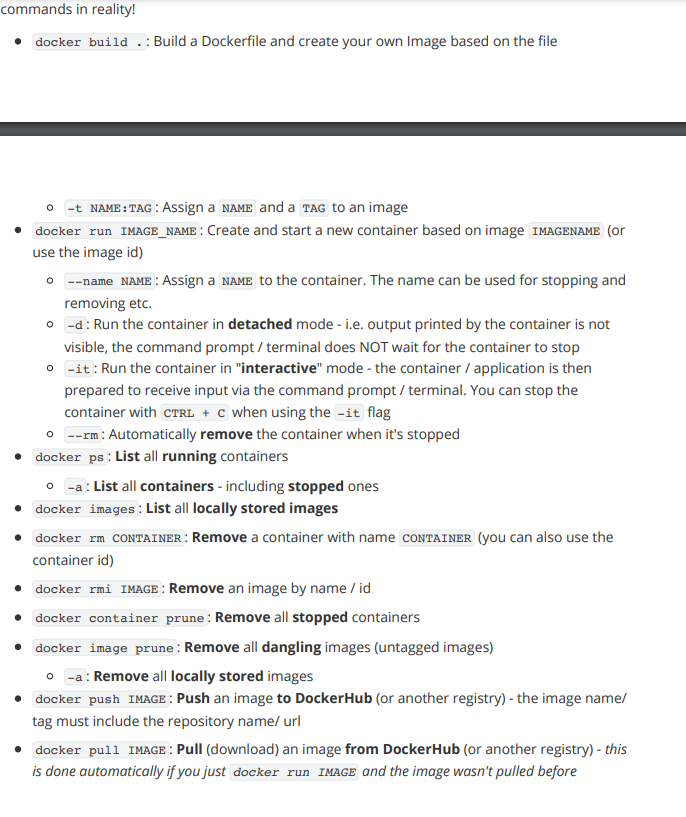
COPY --from=builder /app/build /usr/share/nginx/html

#here in this docker file we creating two set of layers and we are just copying build folder

#from the previous layer to nginx layer

#when second FROM statement is defined previous block of statement will all be stopped or closed





**Removing Anonymous Volumes**

We saw, that anonymous volumes are **removed automatically**, when a container is removed.

This happens when you start / run a container with the --rm option.

If you start a container **without that option**, the anonymous volume would **NOT be removed,** even if you remove the container (with docker rm ...).

Still, if you then re-create and re-run the container (i.e. you run docker run ... again), a n**ew anonymous volume will be created**. So even though the anonymous volume wasn't removed automatically, it'll also **not be helpful** because a different anonymous volume is attached the next time the container starts (i.e. you removed the old container and run a new one).

Now you just start **piling up a bunch of unused anonymous volumes** - you can **clear them** via docker volume rm VOL\_NAME or docker volume prune.

**Bind Mounts - Shortcuts**

Just a quick note: If you don't always want to copy and use the full path, you can use these **shortcuts**:

**macOS / Linux**: -v $(pwd):/app

**Windows**: -v "%cd%":/app

I don't use them in the lectures, since I want to show an approach that works for everyone (and I don't want to switch between both all the time), but you can use these shortcuts depending on which OS you are working on to save some typing.

For example, consider adding the following to entries:

* Dockerfile
* .git

This would ignore the Dockerfile itself as well as a potentially existing .git folder (if you are using Git in your project).

In general, you want to add anything which isn't required by your application to execute correctly.

**Environment Variables & Security**

One important note about **environment variables and security**: Depending on which kind of data you're storing in your environment variables, you might not want to include the secure data directly in your Dockerfile.

Instead, go for a separate environment variables file which is then only used at runtime (i.e. when you run your container with docker run).

Otherwise, the values are "baked into the image" and everyone can read these values via docker history <image>.

For some values, this might not matter but for credentials, private keys etc. you definitely want to avoid that!

If you use a separate file, the values are not part of the image since you point at that file when you run docker run. But make sure you don't commit that separate file as part of your source control repository, if you're using source control.