**Docker tutorials**



Docker swam init –advertise-addr 10.1.0.2

Docker swam join –token <manager-token> 10.1.0.2.2377

Docker swam join –token<worker-token>10.1.0.2.2377

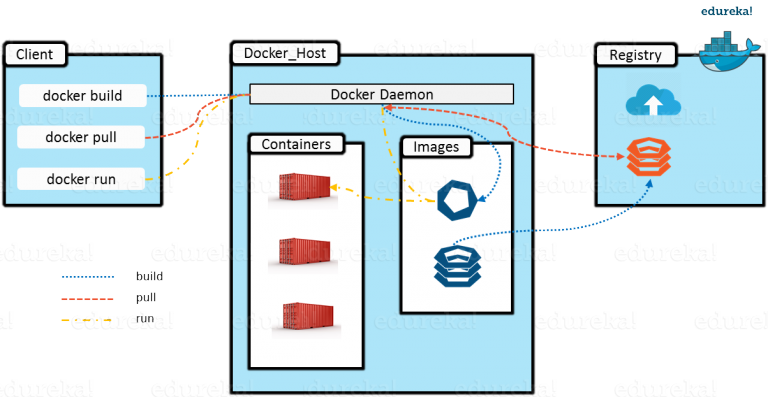
Docker node ls

Docker service create –replicas 3 -p 80:80 –name web nginx

Docker service ls

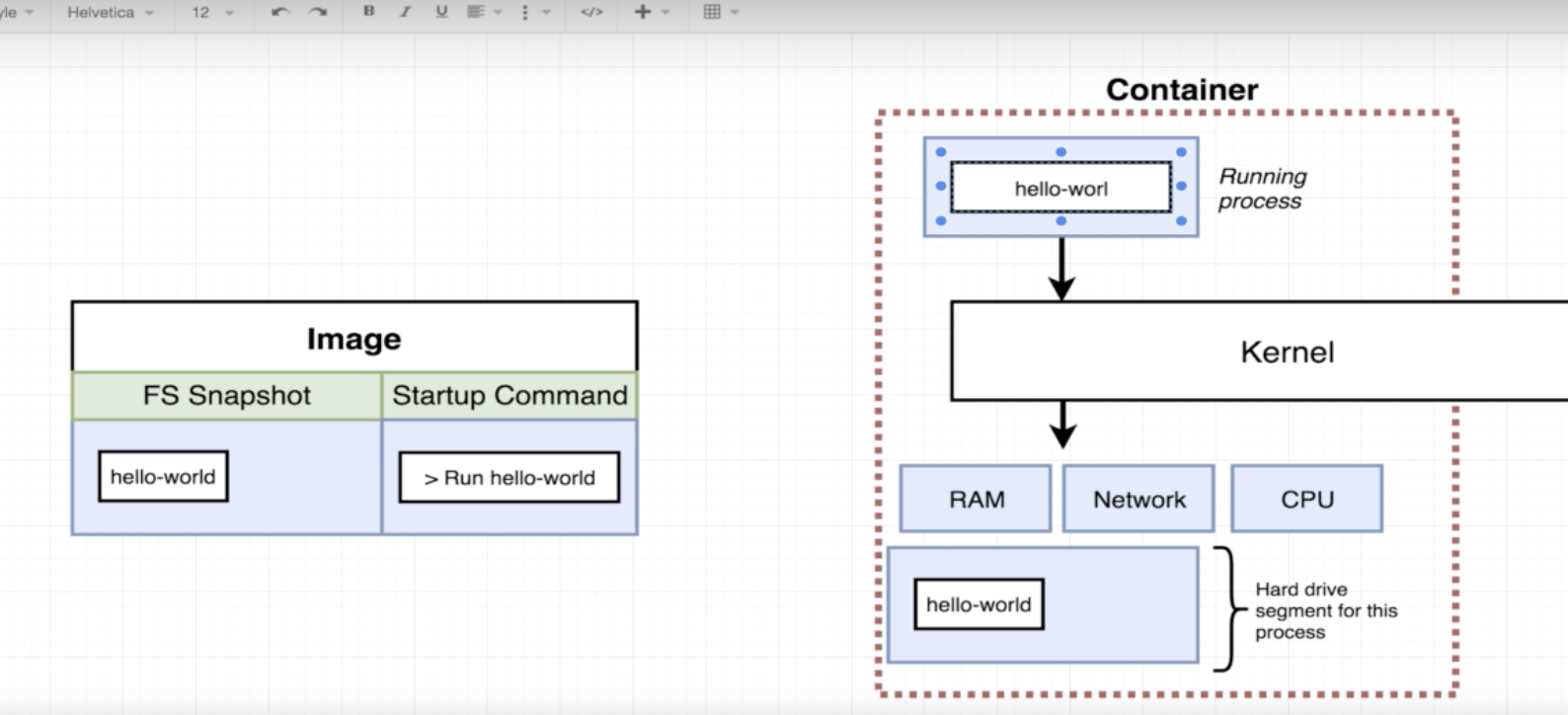
Docker service scale web=3

Docker service ps web



The **Docker daemon** is a service that runs on your host operating system

A **Docker host** is a physical computer system or virtual machine running Linux

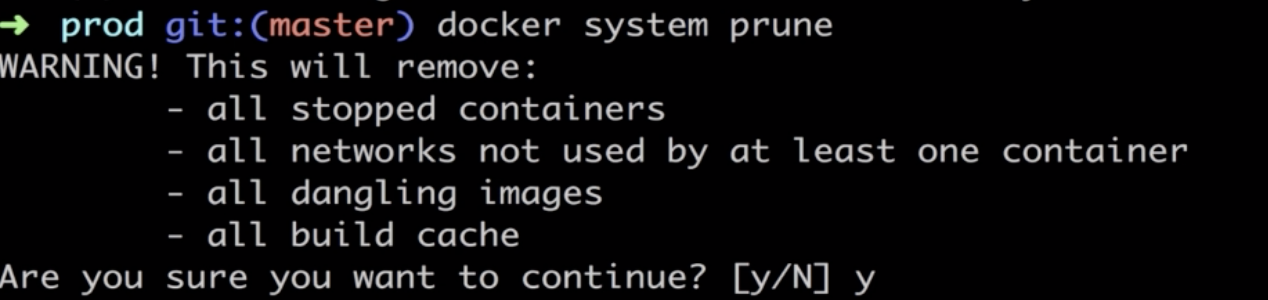


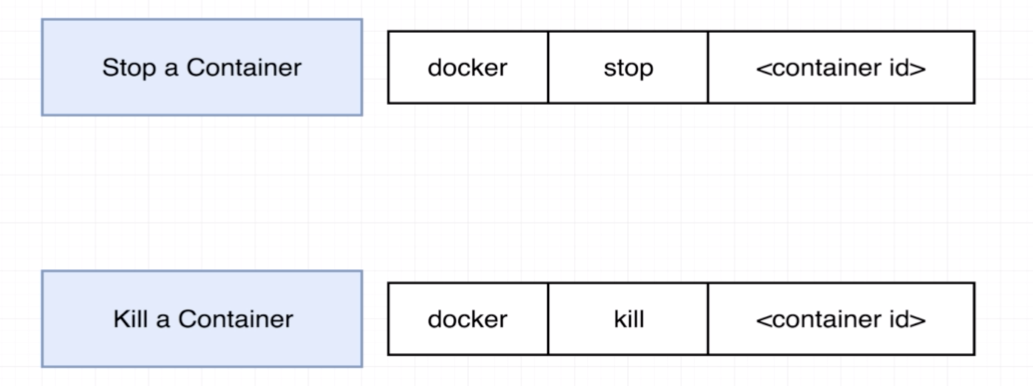
**Closing Container :**

Docker run = Docker create + Docker start

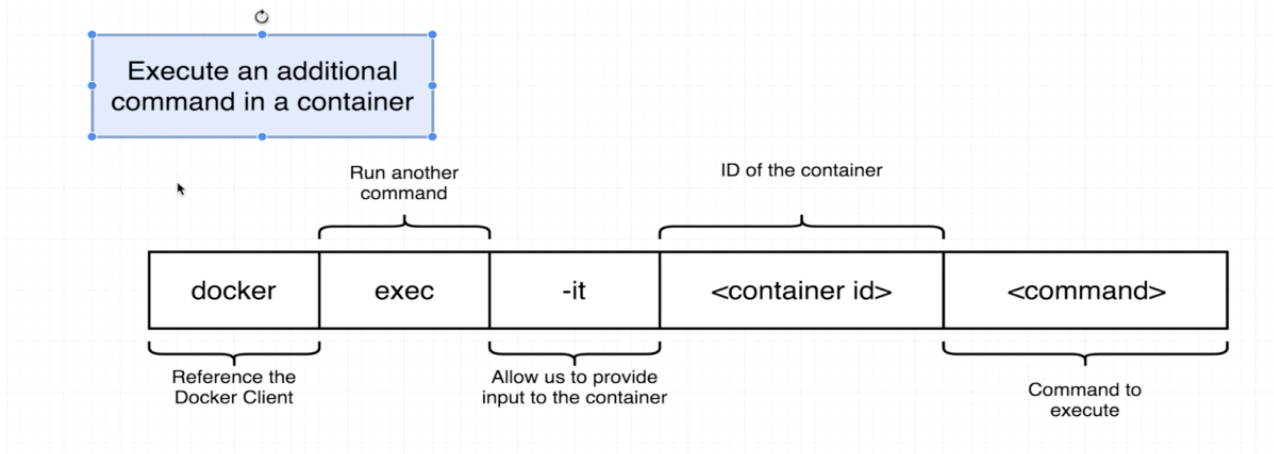
Docker logs <image id> [will give all logs for the container]

Docker system prune (to delete all unnecessary images, cache etc)





**To execute additional command in the container**

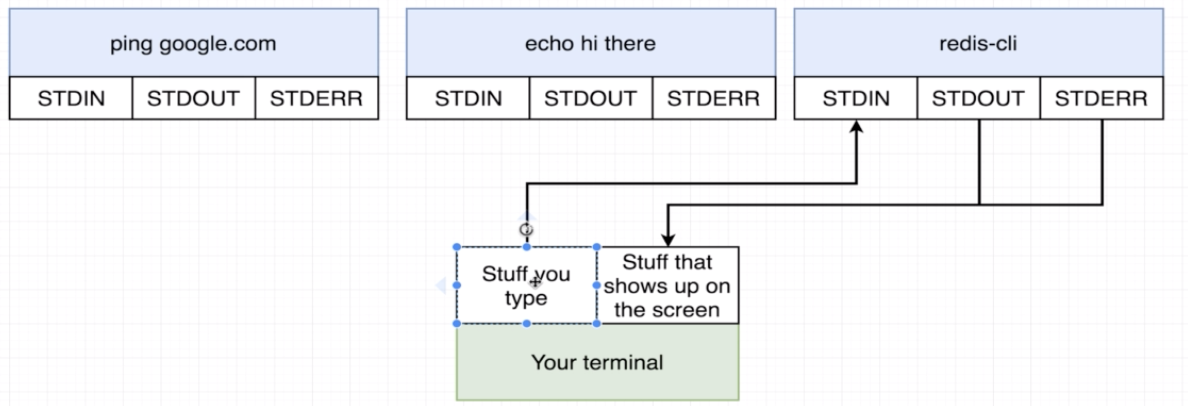


Docker exec -it <container\_id> <command>

Or

Docker run -it <image\_name> <command>

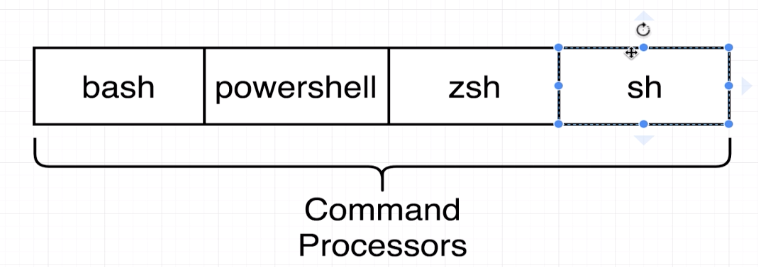
**What is -it flag is doing for us?**



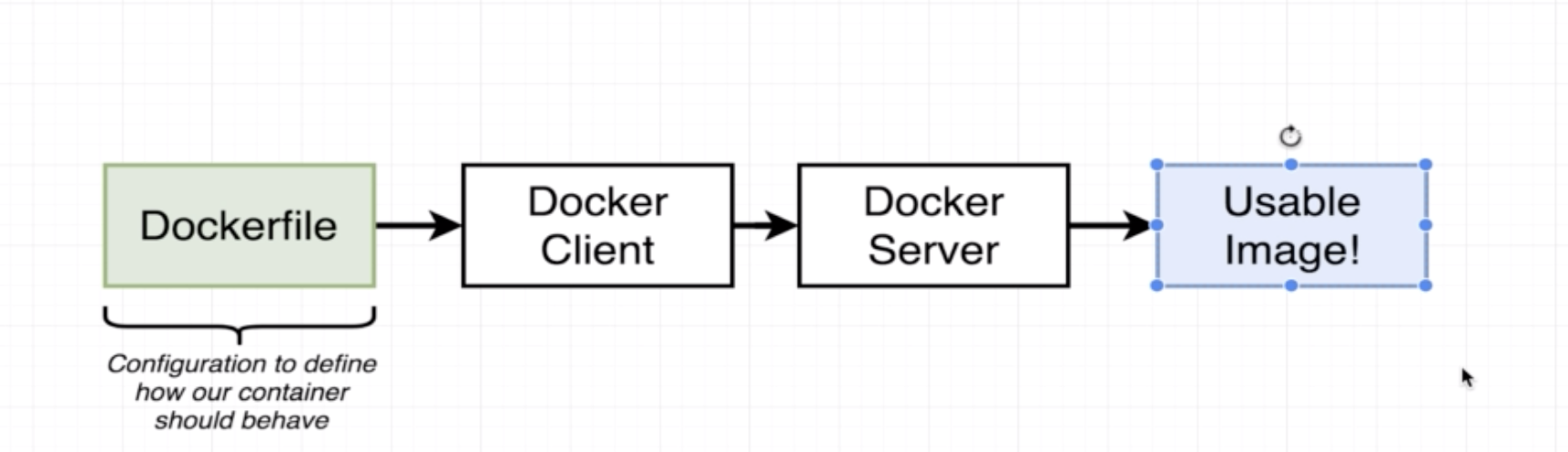
**How to open a shell in docker container**

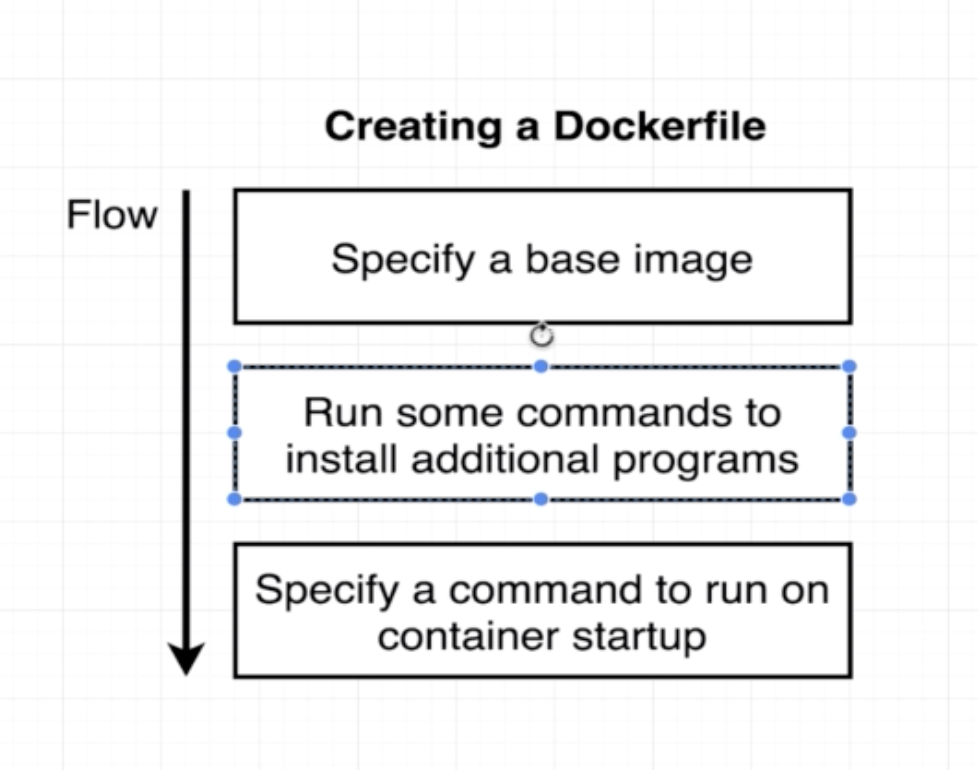
**docker exec -it <container\_id> sh**

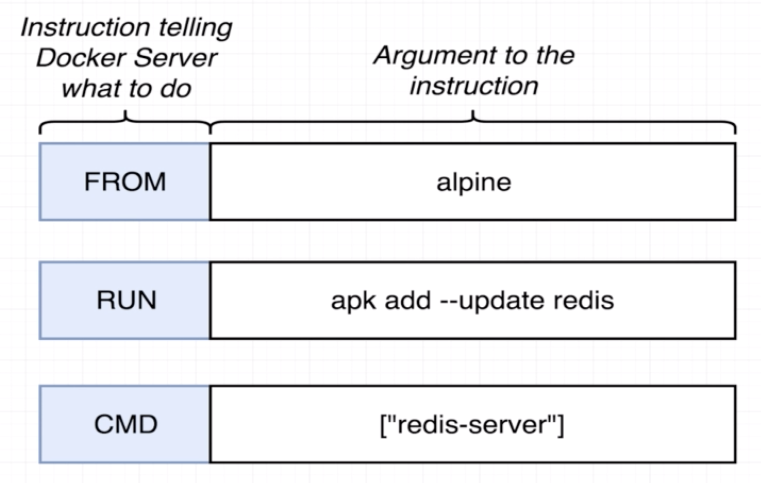
**docker run -it <container name> sh**



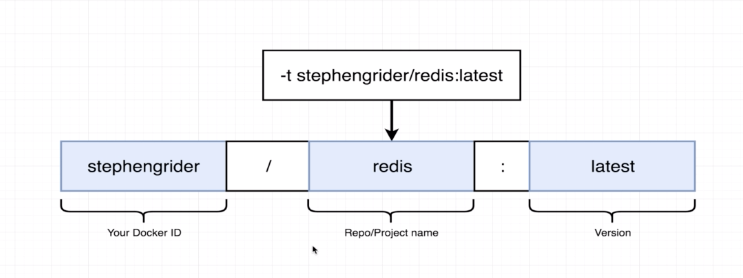
**DOCKERFILE**







**Docket tagging**



To build a docker container

Docker build -t Mahshadul/redis:**latest**

To run container

Docker run Mahshadul/redis

--------

**Let do more deep**

* **Create an node js app**
* **Then create dockerfile**

This lecture is for people who didn't want to write out the code in the last video.**If you already created the index.js file and package.json file, skip this section!**

Here's what you'll need to do:

1) Create a new file called package.json and copy paste the following into it:

{

"dependencies": {

"express": "\*"

},

"scripts": {

"start": "node index.js"

}

}

2) Create a new file called index.js and copy paste the following into it:

const express = require('express');

const app = express();

app.get('/', (req, res) => {

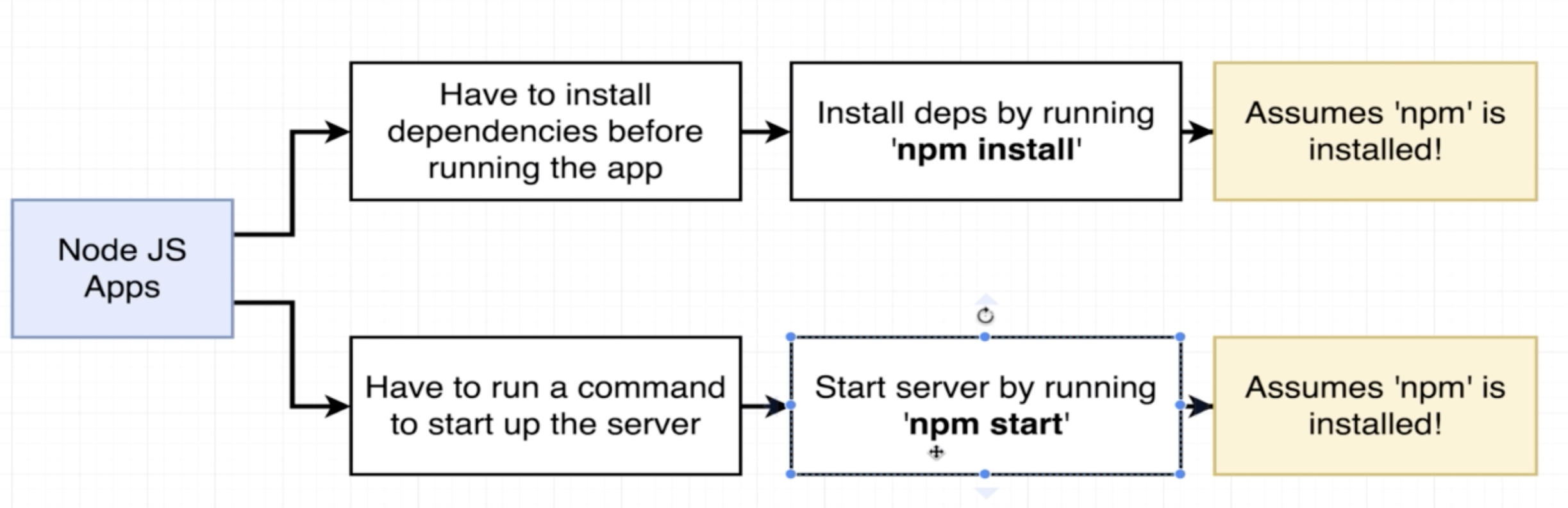
res.send('How are you doing');

});

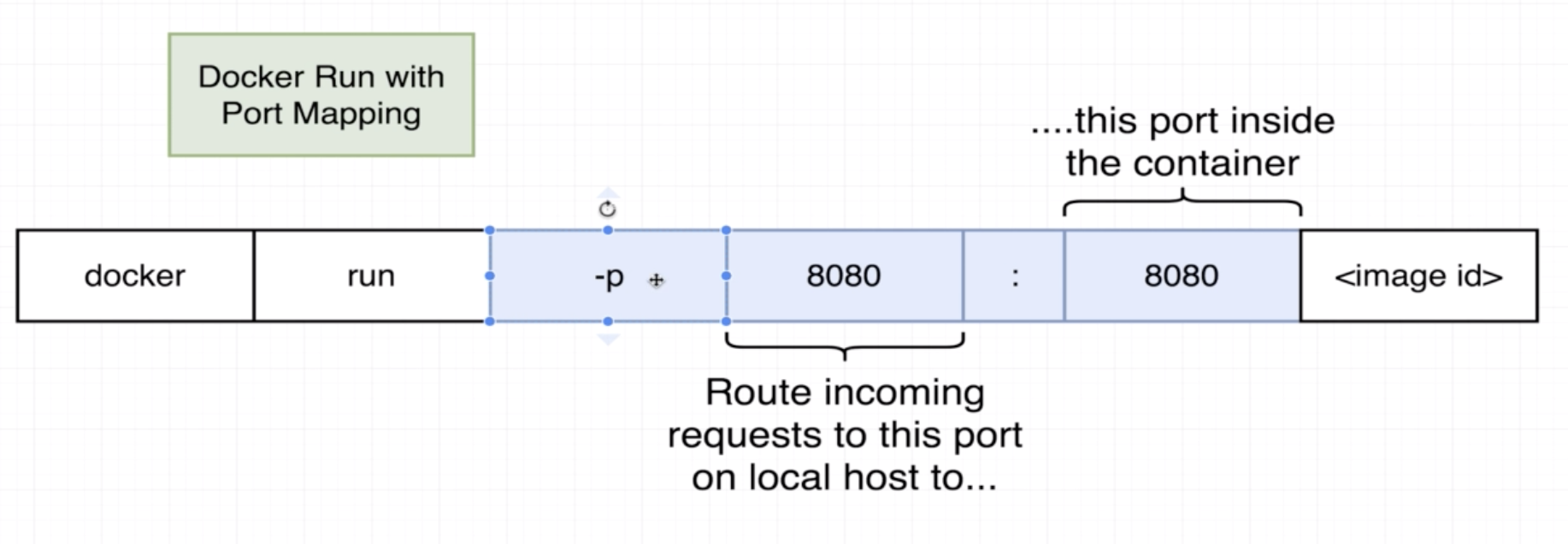
app.listen(8080, () => {

console.log('Listening on port 8080');

});



**Port mapping of local machine and container**



Now docker file should be this

#Specify a base image

FROM node:alpine

# Before copy the file, lets create root folder in the container so, i will have my files are organized

WORKDIR /usr/app

# before do install command we need dependency (package.json) need to be inside the container

COPY ./package.json ./

#install dependency

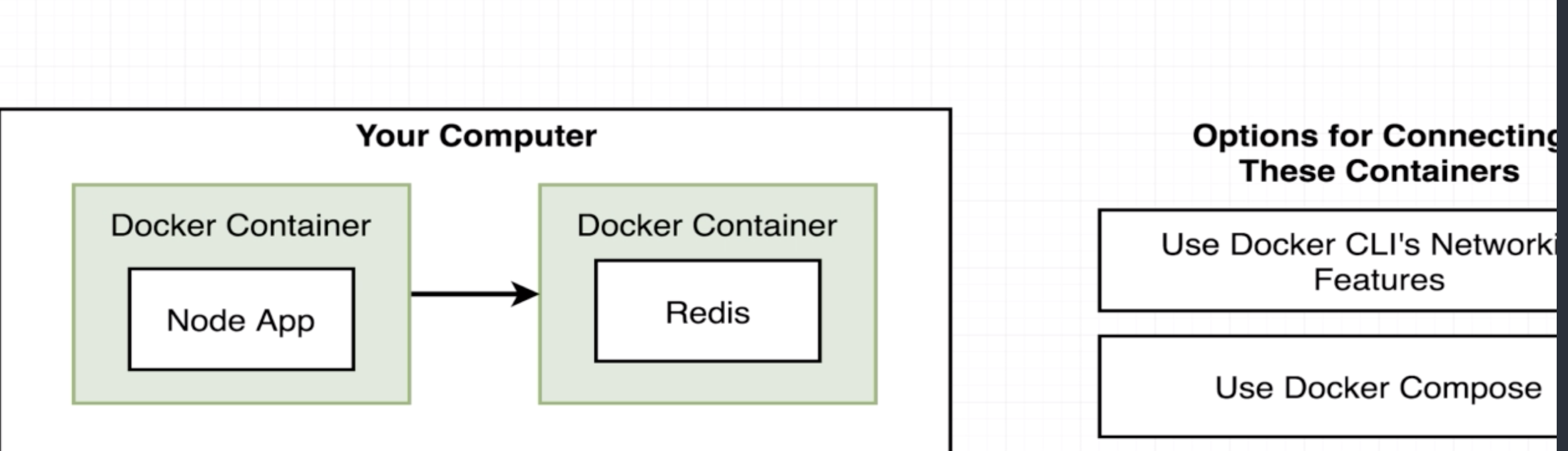
RUN npm install

# now we need to copy othe file inside the container to be able to run necressary application

COPY ./ ./

# defalut command start server

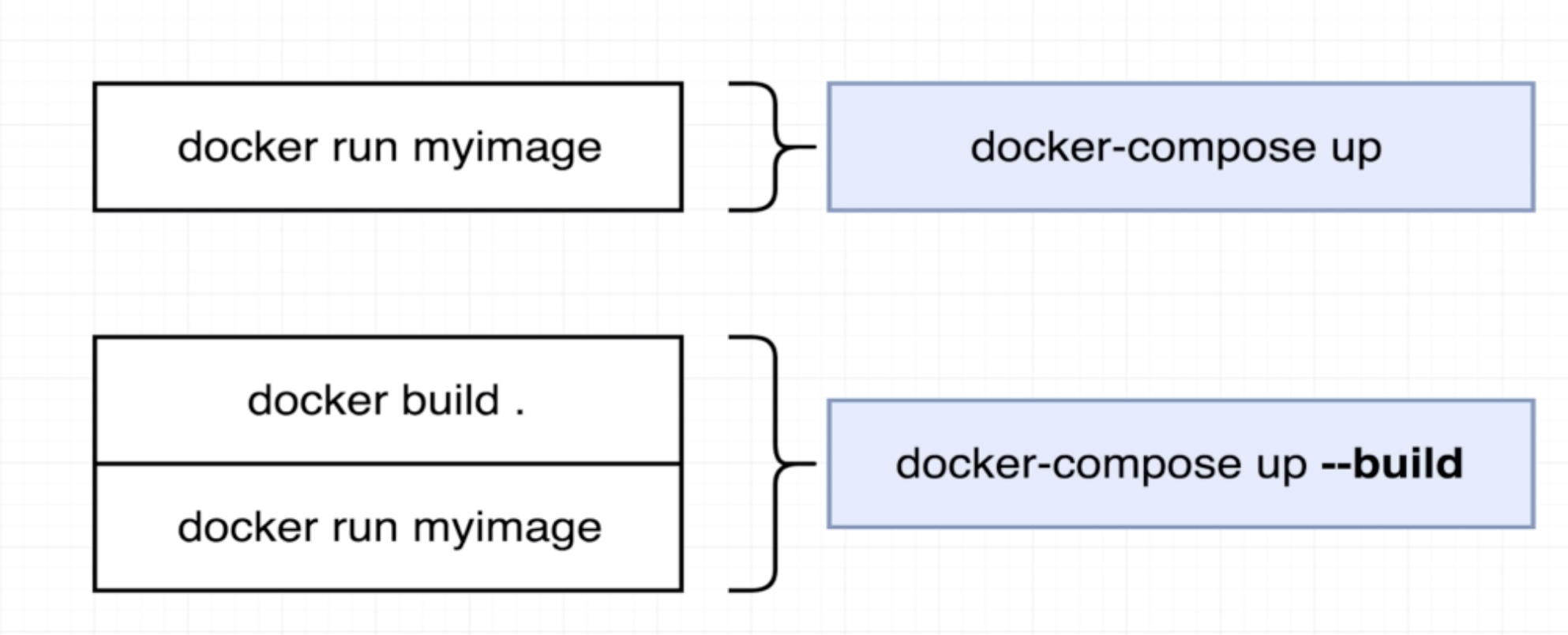
CMD ["npm", "start"]

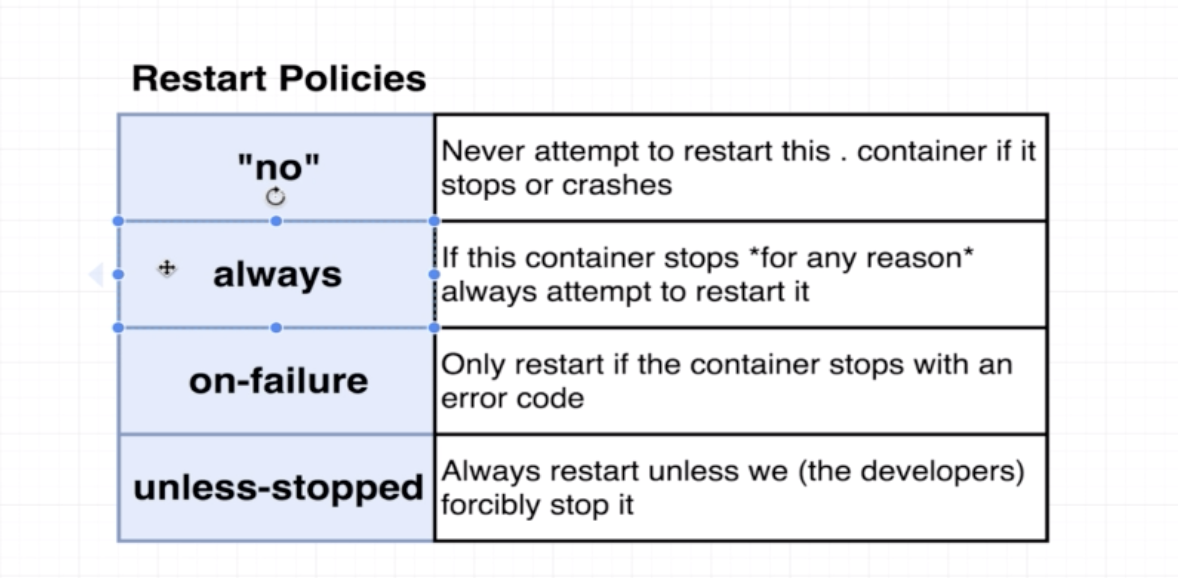


Don’t use Docker CLI, absolutely bad practices

Use Docker compose/Swarm

You need to create file **docker-compose.yml**





Docker-compose down

To Create an React app:

* Need node js install
* Then run
  + npm install -g create-react-app
  + create-react-app frontend
* Inside that directory, you can run several commands:
* **npm start**
* Starts the development server.
* **npm run build**
* Bundles the app into static files for production.
* **npm test**
* Starts the test runner.
* **npm run eject**
* Removes this tool and copies build dependencies, configuration files
* and scripts into the app directory. If you do this, you can’t go back!
* We suggest that you begin by typing:
* cd frontend
* npm start

No create a Dev environment with Dockerfile

* Create a **dockerfile.dev** with following code

FROM node:alpine

WORKDIR '/app'

COPY package.json .

RUN npm install

COPY . .

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

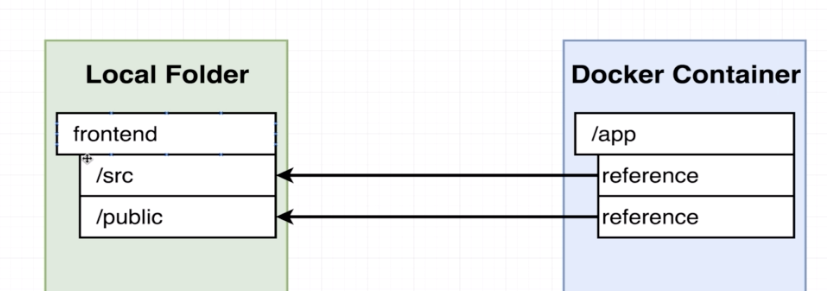
Then run

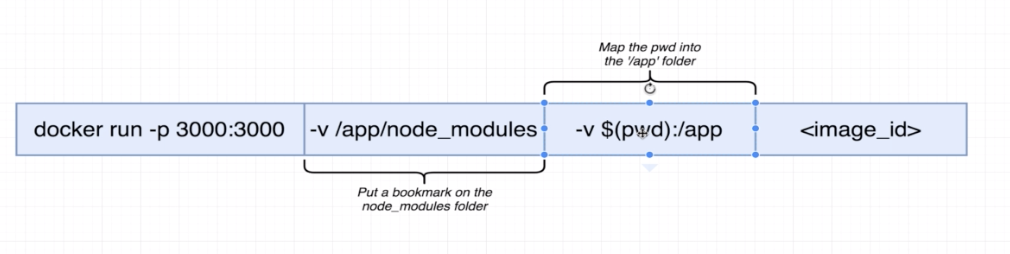
docker build -f Dockerfile.Dev .

docker run -p 3000:3000 <image\_id>

\*\*to make more effective, delete node package file from project

\*\* Every time we change on app, we want to see server has updated, but docker does not do it unless we do followings





Example:

docker run -p 3000:3000 -v /app/node\_modules -v $(pwd):/app 63a492efcbd3

**Docker Compose**

Lets create a file **docker-compose.yml**

version: '3'

services:

web:

build:

context: .

dockerfile: Dockerfile.dev

ports:

- "3000:3000"

volumes:

- /app/node\_modules

- .:/app

Then run

$docker-compose up

It will build and run server at localhost 3000

**Executing test**

docker run -it <image id> npm run test

**Live update tests:**

We can do this to live update test

* Lets change/increase number of test
* Bring up docker compose instance (already has local file references)

$docker-compose up

Open new terminal

$docker ps (copy image id)

@docker exec -it <image id> npm run test

This is not ultimate solution because we need to remember id etc…

**Best practice:**

**Create a new service on docker-compose for test**

version: '3'

services:

web:

build:

context: .

dockerfile: Dockerfile.dev

ports:

- "3000:3000"

volumes:

- /app/node\_modules

- .:/app

tests:

build:

context: .

dockerfile: Dockerfile.dev

volumes:

- /app/node\_modules

- .:/app

command: ["npm", "run", "test"]

then run

$docker-compose up –build

Adding -–build to make sure changes are coming along

(but this solution does not allow to rerun test form terminal, if need then need to run $docker-compose up –build again)

DEPLOYING TO prod:

* Create .travis.yml
* Setup travis CI
* Connect with github project
* Add access id and key as environment variable in travis ci
* Update .travis.yml
* Add EXPOSE 80 on Dockerfile after FROM nginx

FROM node:alpine as builder

WORKDIR '/app'

COPY package.json .

RUN npm install

COPY . .

RUN npm run build

FROM nginx

EXPOSE 80

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

.travis.yml

sudo: required

services:

- docker

before\_install:

- docker build -t mahshadul/docker-react-app -f Dockerfile.dev .

script:

- docker run mahshadul/docker-react-app npm run test -- --coverage

deploy:

provider: elasticbeanstalk

region: "us-east-1"

app: "dockerized-react-app"

env: "DockerizedReactApp-env"

bucket\_name: "elasticbeanstalk-us-east-1-947233605766"

bucket\_path: "dockerized-react-app"

on:

branch: master

access\_key\_id:

secure: "$AWS\_ACCESS\_KEY"

secret\_access\_key:

secure: "$AWS\_SECRET\_KEY"