5/15/2021

Bipin Kumar

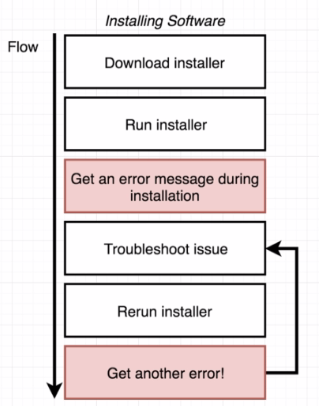
shael.dhn88@gmail.com

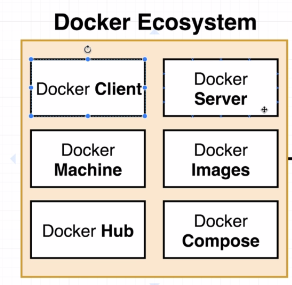
**Docker**

Docker and Kubernetes

What is Docker? Why use docker?

Suppose you want to install a software you will follow below steps you download installer and run installer if you get any error you will troubleshoot issue and then again run installer and so.

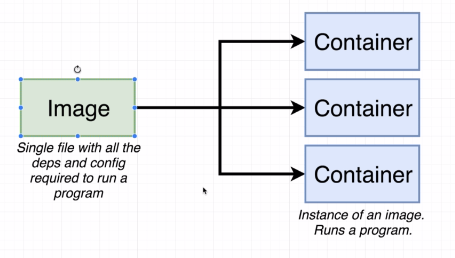
Docker makes it really easy to install and run software without worrying about setup or dependencies.



Docker is platform or ecosystem around creating and running containers.

What is container?

A container is instance of an image which run images and having all dependency already installed to run that project.

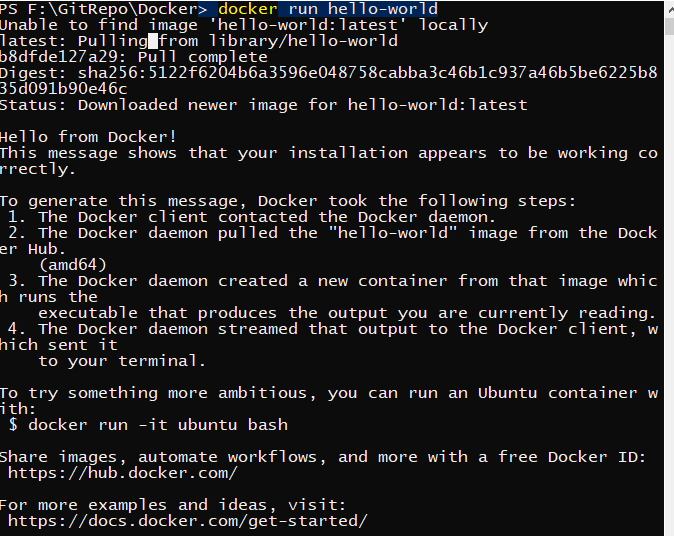


Commands and use:

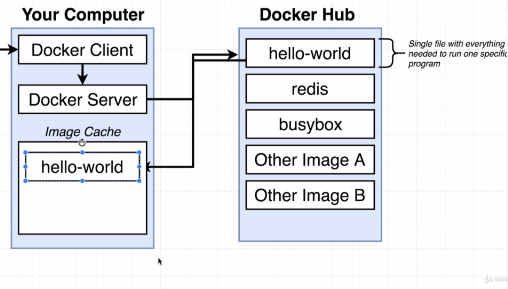
**docker version**: it is used to show you current version of docker installed in your system.

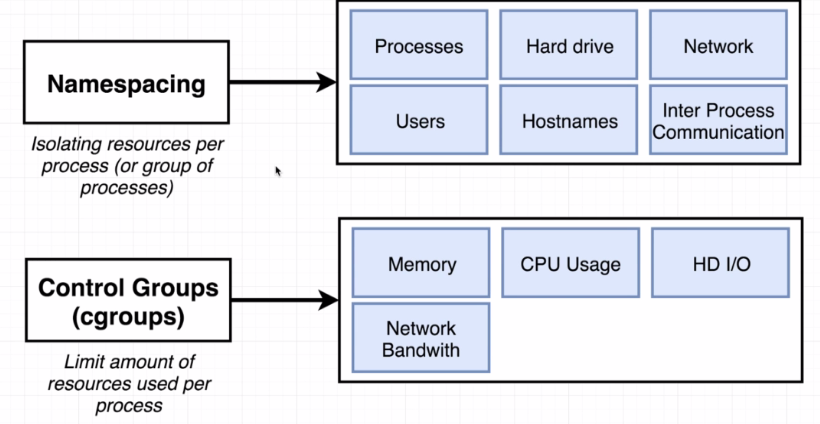
**docker run hello-world :**

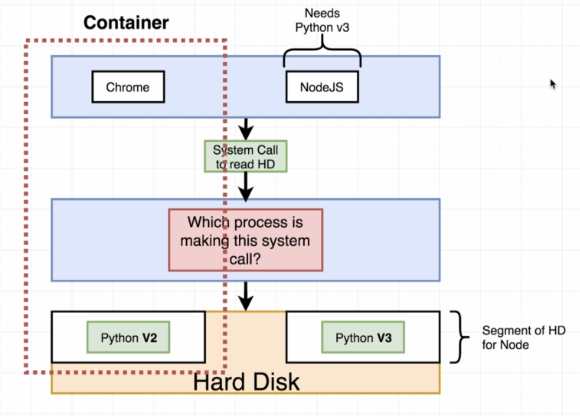
When you run above command you can see below messages printed on your console.



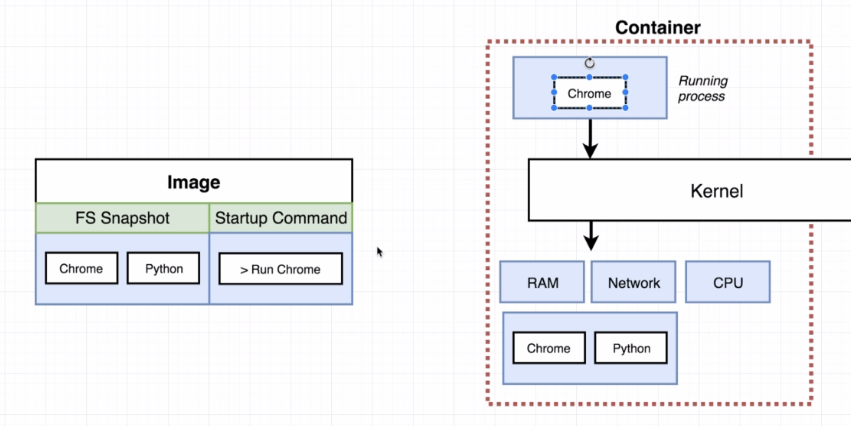
So, what it is doing that it first checks if hello-world image is available locally. Once it didn’t find it try to download it from docker-hub and then it tries to install and run that image.

When you run same command next time you will don’t find that message that image don’t available at locally.



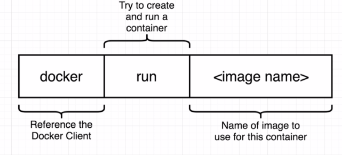


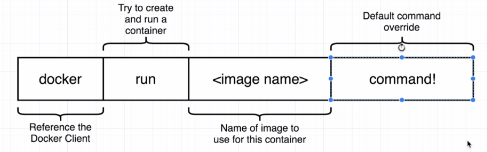
Running container:



Creating and Running a container from an image.

docker run <image name>

e.g. docker run hello-world

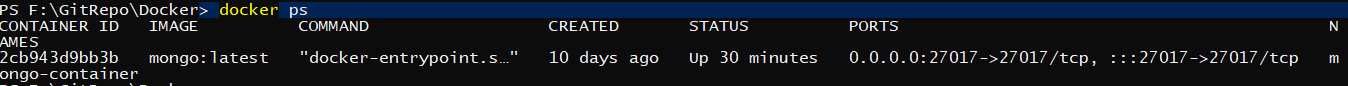


e.g. docker run busybox echo hi there

Listing running command:

**docker ps**: it list down all the image running on docker.

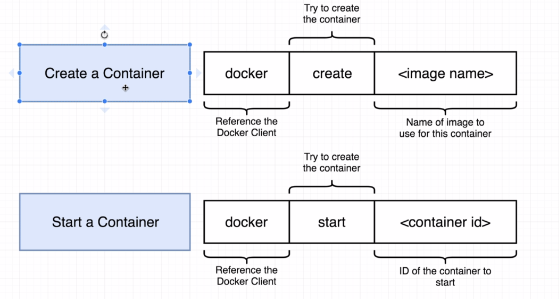
e.g.



**docker ps –all**: list down all the container which we ran.

Container lifecycle:

Docker run = docker create + docker start



docker create hello-world

docker start -a e0e0574557ce40cdfa3e5ad93058b05d64d935a3e161a52e527973ccf29b268a

here -a means whatever output is there print that.

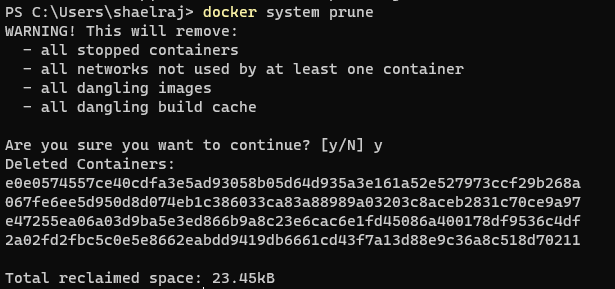
**Restarting container:**

docker ps -all

docker start -a <container-id>

**Remove stopped container:**

docker system prune

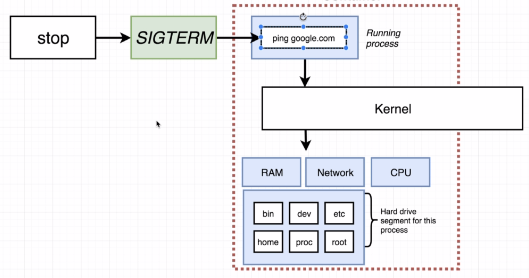


**Retrieving logs output:**

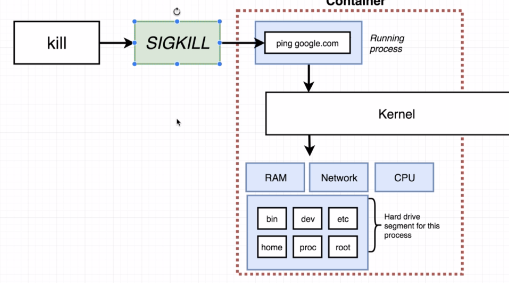
docker logs <container-id>

**Stopping container:**

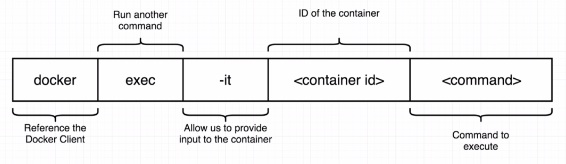
docker stop <container-id>



docker kill <container-id>



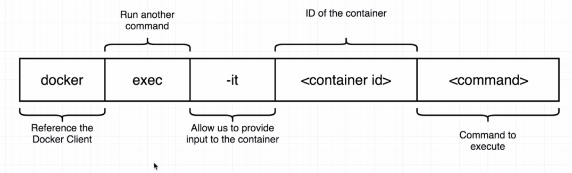
**Multi-command container:**



e.g. docker exec -it c71f0ba6267c redis-cli

-i = attach it to stdin channel

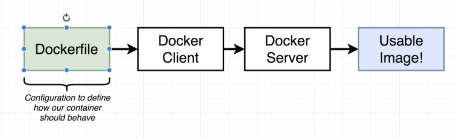
-t = to show in nice format

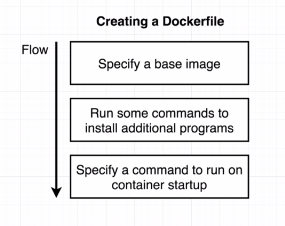


e.g. docker exec -it <container-id> sh

**Creating Docker image:**

Create docker configuration file and passed it to docker client then it sends it to docker server and docker server build a usable image.





* **Create a folder redis-image**
* **Create a Dockerfile with no extension**
* **Write below steps**

# Use an existing docker image as a base

FROM alpine

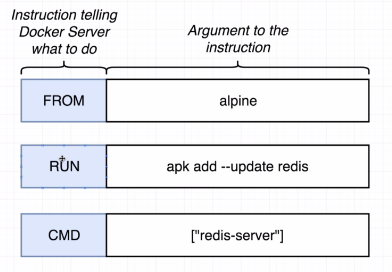
# Download and install a dependency

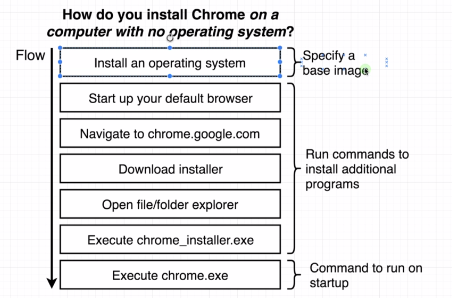
RUN apk add --update redis

#Tell the image what to dowhen it starts

# as a container

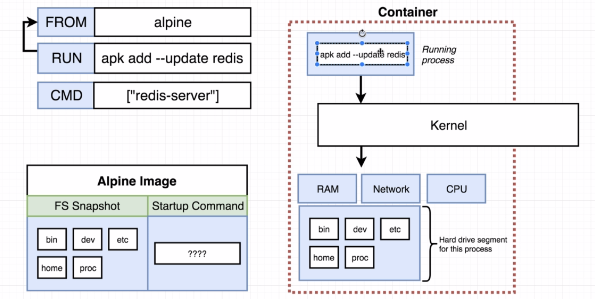
CMD ["redis-server"]

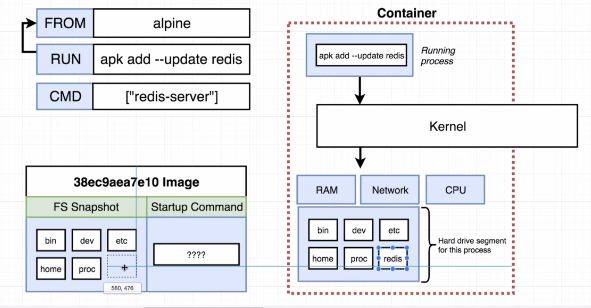
* **Now run command “docker build .”**
* **Once it successfully**
* **executes we can run this image using “docker run <id>”**

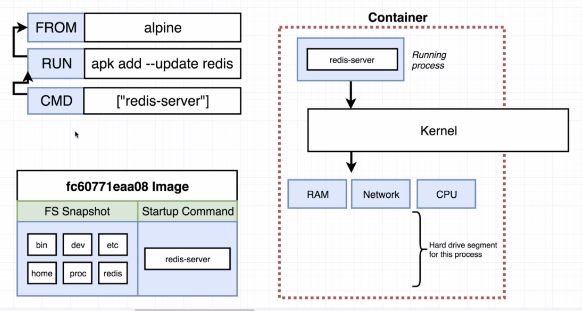


We have used alpine as a base image because it comes with a preinstalled set of programs that are useful to us.

In second line we used apk command that is package manager which is used to install redis in base image and last line indicates the command which we can use once container used.







1. From command:

Download alpine image

1. Run command

* Get image from previous step
* Create a container out of it
* Run apk command in it
* Take snapshot of that container’s FS
* Shut down that temp container
* Get image ready for next instruction

1. CMD command

* Get image from last step
* Create a container out of it
* Tell container it should run when started
* Shut down that temp container
* Get image ready for next instruction

1. No more steps
2. Output is the image generated from previous step

**Rebuilds with cache:**

Now add below command in docker file after run command and run docker build command

RUN apk add --update gcc

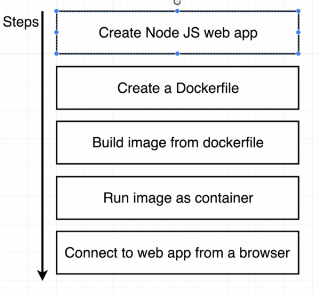
You can see using cache means only after above command it will execute the command rest it will get from cache.

**Tagging an image:**

docker build -t shaelraj/redis:latest .

<your dockerid> / repo/project name : version

Creating project using docker.



1. Create a directory called simpleweb
2. Go to that directory and open IDE or editor on that folder
3. Create package.json file with below contain

{

"dependencies":{

    "express": "\*"

},

"scripts": {

    "start": "node index.js"

}

}

1. Create index.js with below contain.

const express = require('express');

const app =express();

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

    res.send('how are you doing!!!');

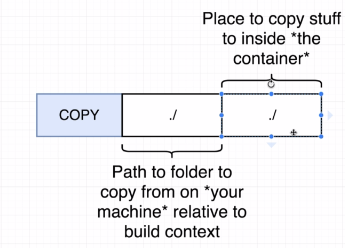
});

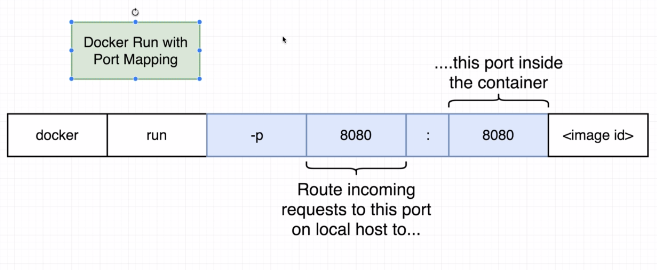
app.listen(8080,()=>{

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

});

1. Create docker file with below contain





#Base image

FROM node:lts-alpine

#install dependencies

#if not created then it will create a file

WORKDIR /usr/app/

# copy package.json from build context to container context

COPY ./package.json ./

#run the commands

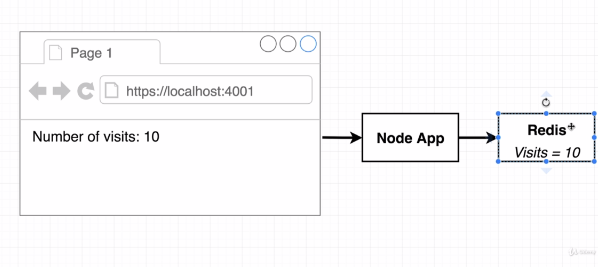
RUN npm install

#copy everything

COPY ./ ./

#Default command

CMD ["npm", "start"]



Above diagram show use case where we have node app and redis server which store no. of time a page visited.

Create directory called visits and change directory to visits.

Create package.json file inside it with below contain

{

"dependencies":{

    "express": "\*",

    "redis": "3.0.2"

},

"scripts": {

    "start": "node index.js"

}

}

Create index.js file inside it will below contain

const express = require('express');

const redis = require('redis');

// maintence

const process = require('process');

const app =express();

const client = redis.createClient({

    host:'redis-server',

    port: 6379

});

client.set('visits', 0);

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

    client.get('visits',((err, visits)=>{

        //process.exit(0);

        res.send('Number of visits is !' + visits);

        client.set('visits', parseInt(visits) +1 );

    }));

});

app.listen(8080,()=>{

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

});

Create docker file with below contain:

#Base image

FROM node:lts-alpine

#install dependencies

#if not created then it will create a file

WORKDIR /usr/app/

# copy package.json from build context to container context

COPY ./package.json ./

#run the commands

RUN npm install

#copy everything

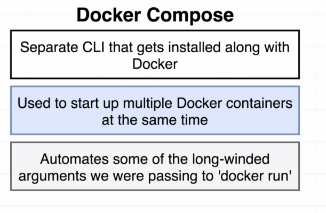
COPY ./ ./

#Default command

CMD ["npm", "start"]

When you run this application, you will get error. As both the container cannot communicate with each other so we need to make some changes so that they can talk to each other.

We can make use docker cli or docker compose. Generally, most of person use docker compose.



Creating a file called docker-compose.yml file

Put below code on that file

version: '3'

services:

  redis-server:

    image: 'redis'

  node-app:

    restart: on-failure

    build: .

    ports:

      - "8080:8080"

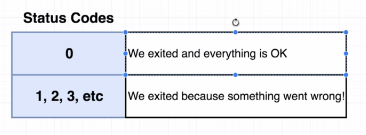
docker-compose up (it is similar to docker run myimage)

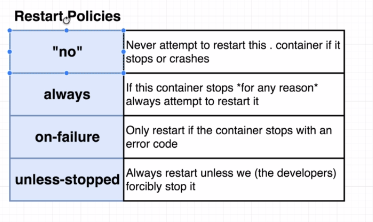
docker-compose up –build (it is similar to docker build . then docker run myimage)

launch in background:

docker-compose up -d

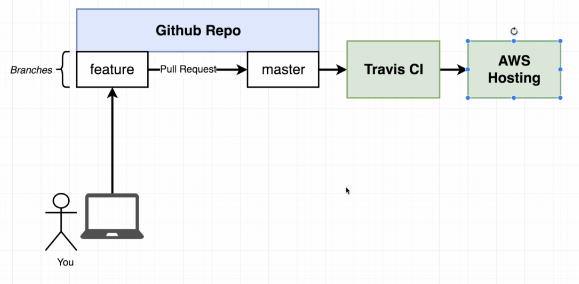
docker-compose down

**Container maintenance with compose:**

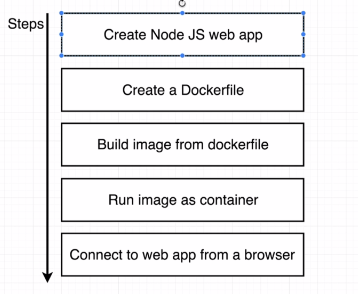


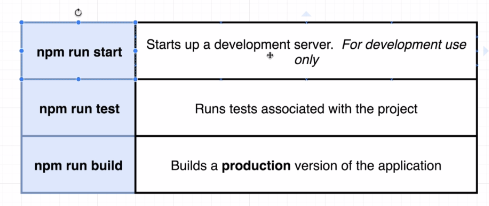
Docker-compose ps : only work on the directory where you have docker-compose file

**Creating production grade workflow:**



Docker is a tool in a normal development flow. It makes some of these tasks a lot easier.

1. Create a project using react js app generated
2. Goto the project directory.
3. We gone run npm related command.



**Creating dev docker file:**

Create a Docker.dev file with below contain

FROM node:lts-alpine

WORKDIR '/app'

COPY package.json .

RUN npm install

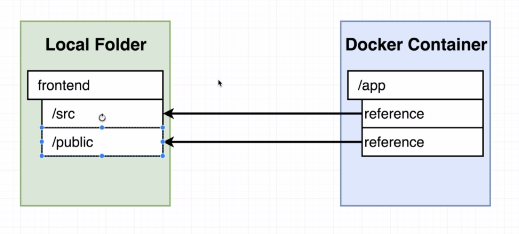
COPY . .

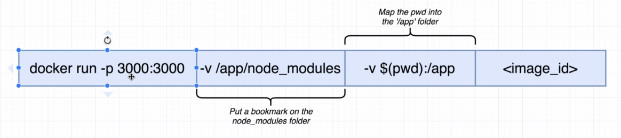
CMD ["npm", "run", "start"]

docker build -f Dockerfile.dev .

here -f indicates file using which we need to build our file.

Currently if we run the image and try to change the file locally the image doesn’t reflect the changes. To do so we need to build image again and then again re-run the image is required. We can use docker volume concept.





Bookmarking volume:

For bookmarking anything we use -v /app/node\_modules

Using docker compose file:

version: '3'

services:

  web:

    build: .

    ports:

      -"3000:3000"

    volumes:

      - /app/node\_modules

      - .:/app

When we run above docker compose file we will get error because we don’t have any dockerfile inside our folder. Instead we have dockerfile.dev so we need to do additional configuration inside our docker compose file.

version: '3'

services:

  web:

    build:

      context: .

      dockerfile: Dockerfile.dev

    ports:

      -"3000:3000"

    volumes:

      - /app/node\_modules

      - .:/app

To run test:

To run test, we can use previous concept where we are passing other commands to docker run command. And it we want any input then we have to use -it along with command.

Using docker compose:

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

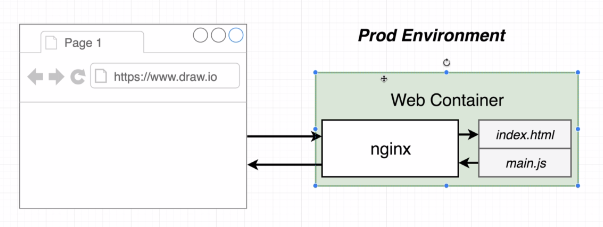
      - .:/app

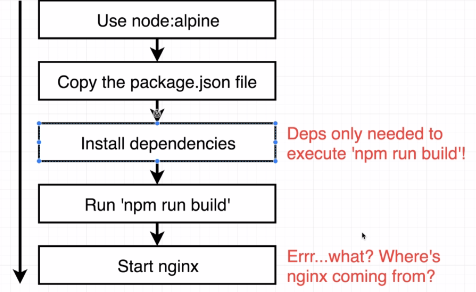
    command: ["npm", "run", "test"]

when we run above docker-compose file two container one for web and one for test are created. When we try to give some input, nothing will happen. As there are two container and terminal don’t know to which this command for.

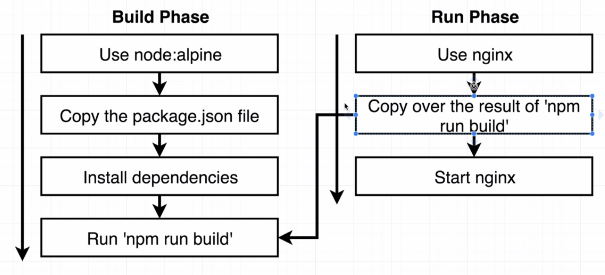
To attach terminal to specific container run docker ps command and then using below command attach the terminal to that container. But it only attaches to master process

Docker attach <container-id>





Multi step build process:



For multilayer step we need to create multistep docker file as given in below example.

FROM node:lts-alpine as builder

WORKDIR '/app'

COPY package.json .

RUN npm install

COPY . .

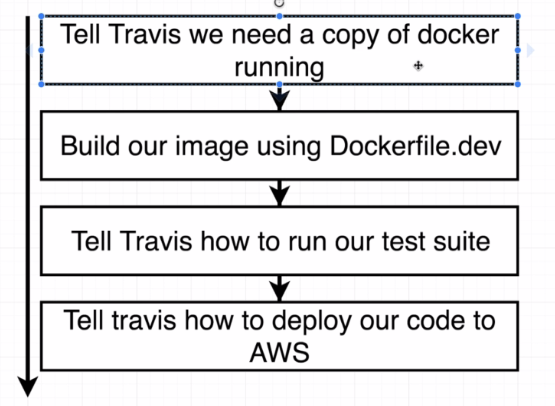
RUN npm run build

FROM nginx

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

Continuous integration and deployment with AWS.

1. Create repository in github
2. Create account in travis-ci and give access to github repository



1. Create .travis.yml file with below contain.

sudo: required

services:

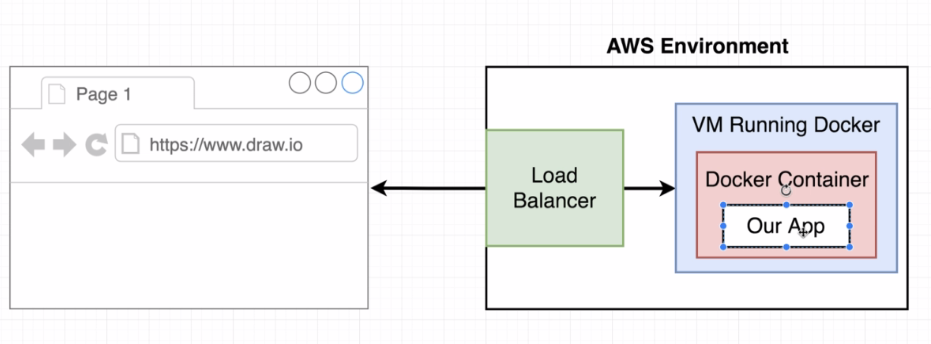
  - docker

before\_install:

  - docker build -t shaelraj/docker-react -f Dockerfile.dev .

script:

  - docker run shaelraj/docker-react npm run test -- --coverage



#deploy:

#  provider: elasticbeanstalk

#  region: "us-west-2"

#  app: "docker-react"

#  env: "Docker-env"

#  bucket\_name: "elasticbeanstalk-us-west-2-30647662547"

#  bucket\_path: "docker-react"

#  on:

#    branch: master

#  access\_key\_id: $AWS\_ACCESSS\_KEY

#  secret\_access\_key:

#    secure: "$AWS\_SECRET\_KEY"

<https://github.com/shaelraj/docker-react>

Remember, we need to delete the resources we created or you might end up paying real money for them.  To clean up the Elastic Beanstalk instance we created, do the following:

**1) Go to the Elastic Beanstalk dashboard, where you should see a page that looks like this:**

**2) On the top right-hand side click the 'Actions' button**

3) **Click on 'Delete Application' then confirm the delete**

#### Note: it might take a few minutes for the dashboard to update and show that your app is being deleted.  Be a little patient!

Single container deployment issue:

1. The app was simple no outside dependencies
2. Our image was built multiple times
3. How do we connect to a database from a container?

Multi container:

For this we are going to create a fib calculator.

