DOCKER

CRASH COURSE

PLATFORM: ▶

TUTOR: Netninja

DURATION: 13 vids in Playlist

STARTED: 19th Feb 2023

FINISHED: 22nd Feb 2023

What is Docker? #1

Docker is an Open Source platform that enables developers to build, deploy, run, update and manage containers.

Now developers often overlook docker, but it is a very important tool for any developer to know. Why?

Because it can make development of applications, either on your own or in a team much easier to manage.

By Using "Containers"

These containers help to run applications in Isolated environments on a Computer e.g

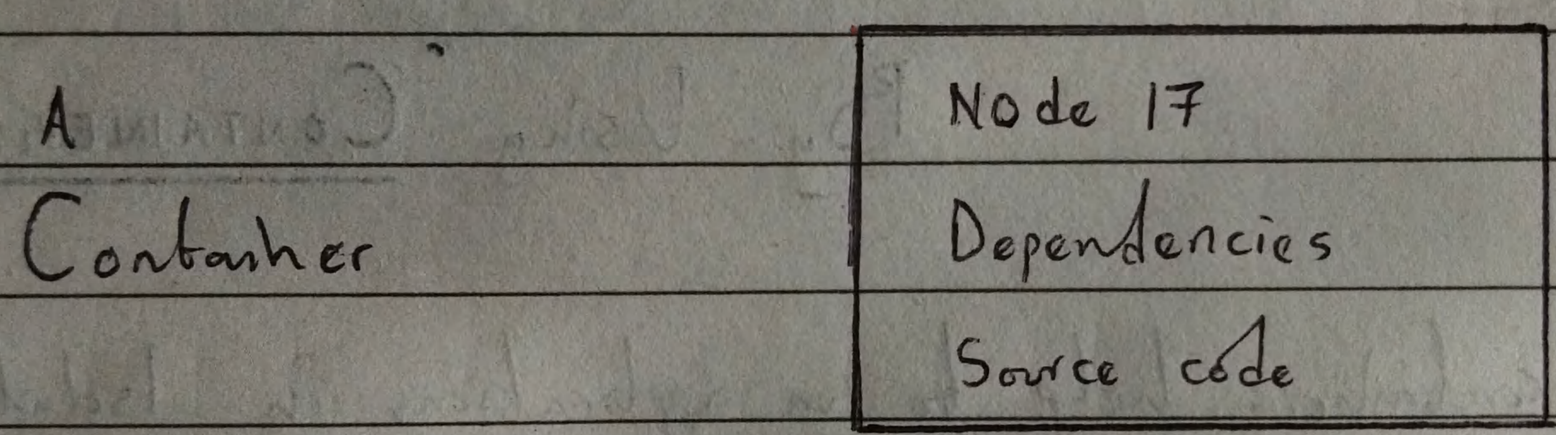
Node app	React app	Mongo db	And Any Other App you can think of...

Real Life

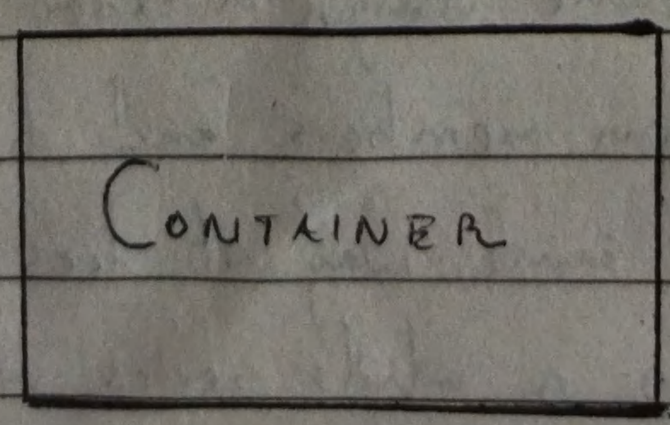
Usually before docker came along, in order to share an application you developed to other team members and have their systems run that application the same way it does on your system. They would need to undergo a whole series of installations and

of that applications dependencies and co setups to ensure that it runs on their systems. This was a very inefficient way of doing things - and that is when docker came around

Docker provided developers with containers - these contain would contain everything an application needs to run. This means all the source code, the dependencies, the correct runtime environment e.t.c



These Containers can now run an application in ISOLATION, away from any other processes on our computer - which means it wouldn't matter the dependency versions we have installed globally on our computer - Because everything the application needs in order to run is in its



This then makes it easy for me and other people in my team work with with each other sharing any application we are working on amongst ourselves

Without having to undergoe a series of dependency installations and so on. - The only thing me and everyone of my team members would need is ——— DOCKER

"Docker in short is a Tool for managing Containers."

Mind Blowing Fact

Much as these containers can run on our computers, they can also be deployed on to your production server and that way we wouldn't need to do any configurations on the server since the container has it all already.

Before You Start ...

- You must have some Web Development Experience
- We will be using Node.Js, Express & React
- Link to Code Repo

www.github.com/iamshaunjp/docker-crash-course

INSTALLING DOCKER #2

For

- Visit the below link ~~to~~ and then Download Docker
  docs.docker.com/desktop/windows/install

- Next, we want to enable WSL (Windows Subsystem for Linux)
  docs.microsoft.com/en-US/windows/wsl/install-manual

- Run Powershell as an administrator
  - Copy & Paste Step 1 code into Powershell and hit ENTER
  - Repeat for Step 3 code and hit ENTER
  - Then restart

- Download the Linux Kernel Update package by clicking on the link in Step 4
  - After it is downloaded, Run the package

- Run Powershell as an administrator
  - Copy & Past Step 5 code into Powershell

- Lastly, Install a Linux distribution on microsoft store
  - Search for ubuntu and then download anyone of ur choice
  - Install it and then Open it and then
  - Set your username & password

Images & Containers #3

When working with docker their are some terminologies and core concepts that we need to understand - and two of those things are **Images** and **Containers**

What are Images?

They are like blueprints for containers and they contain the following things stored inside them "But" They don't actually have those apps running inside them - they can only serve as a blue-print for creating containers that run applications in Isolation.

- They may Contain
  - Runtime environment (and a Lightweight Linux distribution)
  - Application Code
  - Any dependencies
  - Extra Configuration (e.g. env variables)
  - commands

- Images also have a file system too which is independent from the rest of ones in our host computer.
- Images are "READ ONLY". This means once u've created an image it cannot be changed - If you really need to change some thing about the Image then u must create a brand new Image

What are Containers ?

They are runnable instances of those Images.

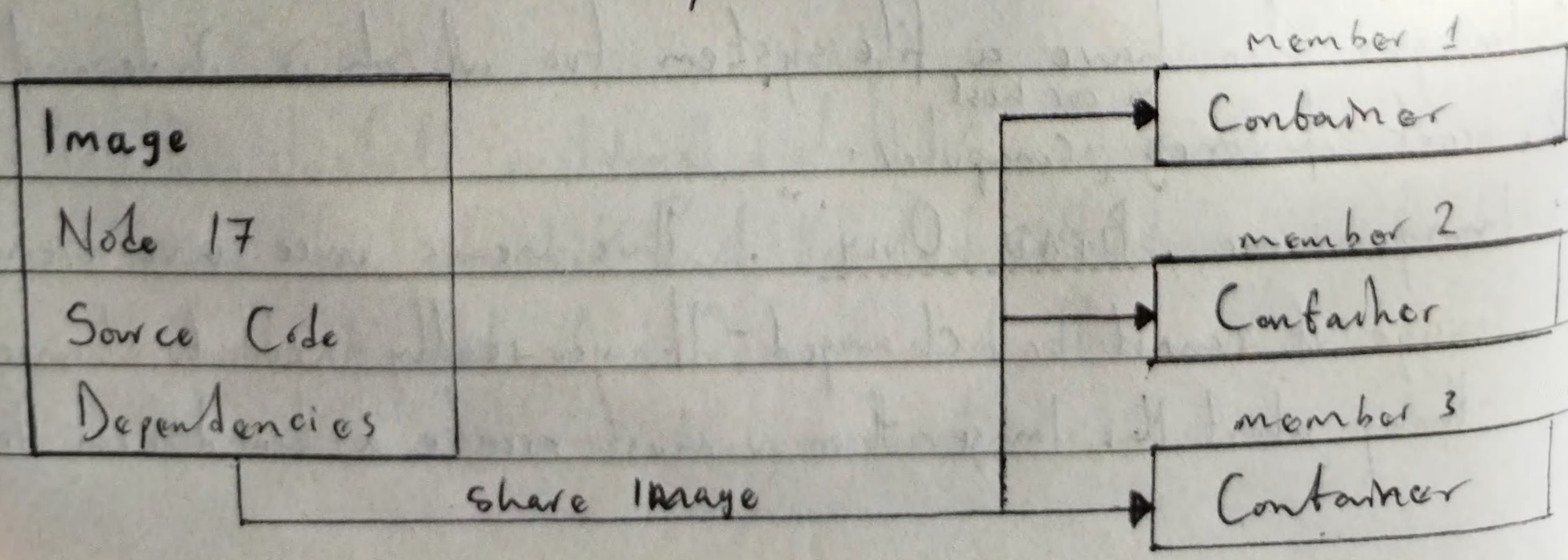
Image		Container
Runtime environment		Running instance of an
Application code	Run →	Image
Any dependencies		
Extra Configurations		Runs our application
Commands		

How ? When we run an image - It creates a runnable instance of our application (a Container)

Containers are also known as "Isolated Processes" meaning that they run independently from any other process on your Computer

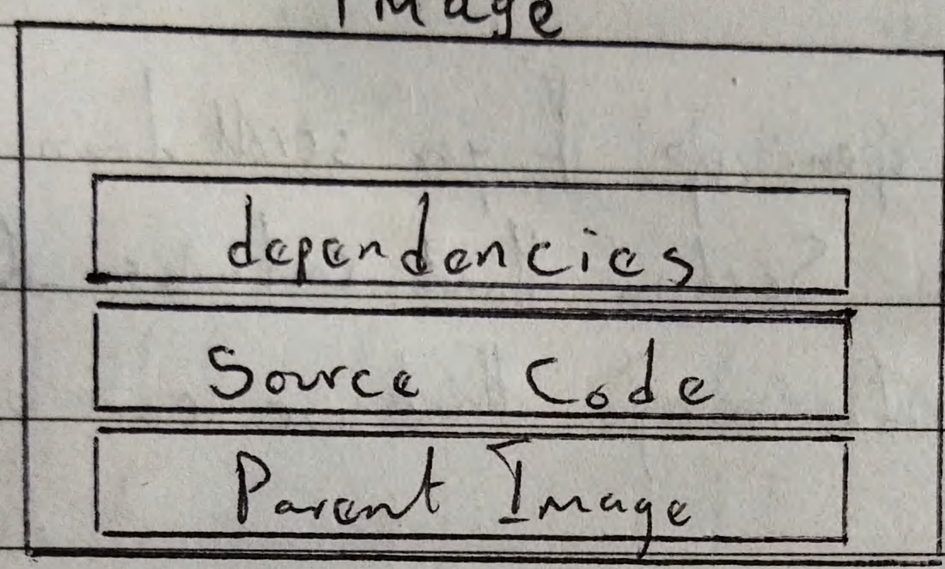
ADVANTAGE :

We could create an Image and share it with our team members



Parent Images & Docker Hub #4

Images are made up of several layers - where each layer adds something to the image **incrementally** in a Stack format



An Image must start with a "**Parent Image**" Layer as the First Layer.

The parent Image typically includes ~~the~~ a Lightweight Linux Operating System & the runtime environment of the container that we want.

So for example we could have the Parent Image of an app we want to build containing a light linux OS and then node 17 runtime environment

We ~~would~~ will be working with official docker online repo to get any docker Image of our choice
- Visit hub.docker.com

- Search for ~~a~~ node

node — node - Official Image

updated 4 days ago

Node.js is a Javascript-based platform for server-side

- When the node page opens up, if you scroll down. You would see a Supported tags Section that contains the different variations of the node image that we can use

Note: When installing an image (Parent Image) it is always good to specify the ~~one~~ version you want, if you don't - docker would automatically install the latest version

- Copy the below line - Paste in Windows Terminal - Hit Enter

```
docker pull node
```

Terminal

```
PS C:\Users\shawn> docker pull node
```

#Pasted

↳ Pls specify version

- After the node Image has installed - Open up our docker app and if we got to Images we should be able to see the Image node (it's latest version)

- Run our node Image to get a Container

Containers/Apps	Images
Images	An Image is a read only template with instructions ...
volumes	Local Hub
Dev Environments	... ↓

NAME	TAG	STATUS	CREATE	SIZE	ACTIONS
node	latest	Unused	...	...	▷ RUN

- We would be taken to the Container section, Next Open Our newly generated Containers CLI lets write some Node Commands

Containers/Apps	Search
...	Confident_bardeen ▷ CLI

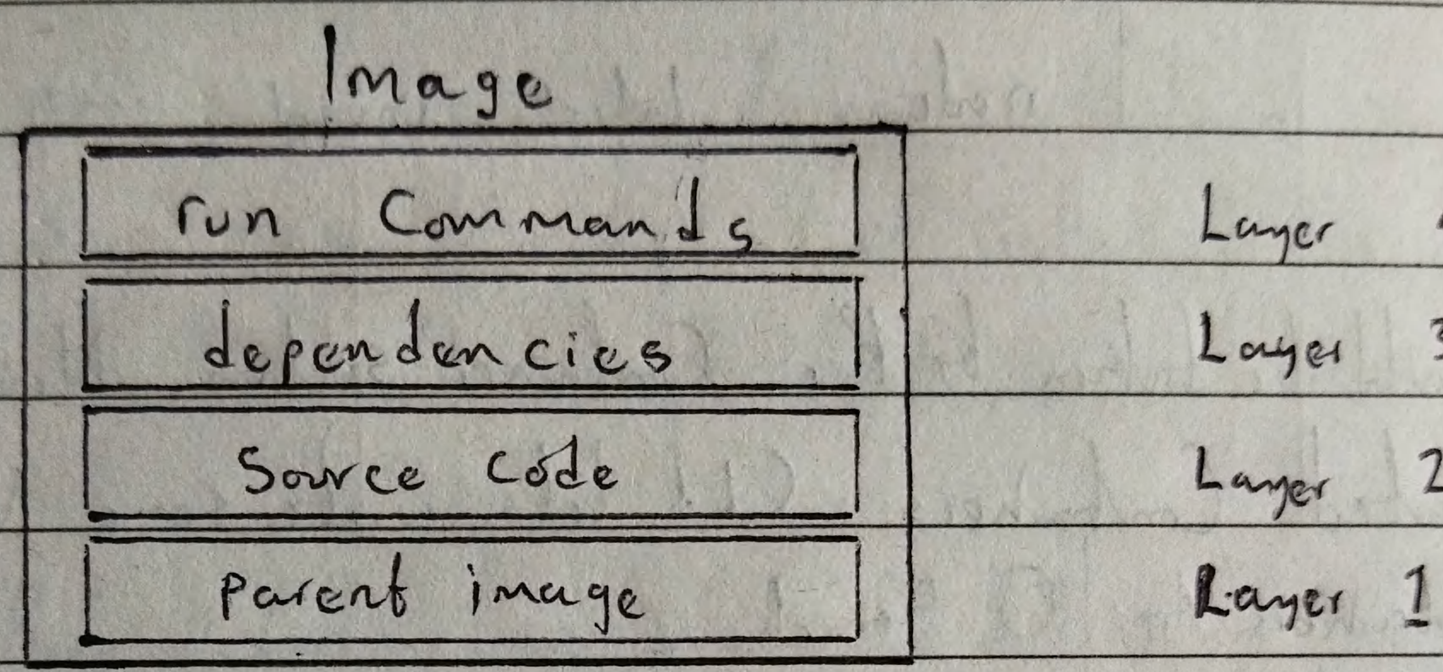
Terminal

```
# ls
bin boot dev etc home lib lib64 media mnt opt ...
# node
Welcome to Node.Js v17.4.0.
> 5 + 10
15
```

- Next Up: We will see how to create our own Image and then use a node Image as our parent Image

The DockerFile #5

Now to Create our own docker image, we would need to create A Docker File - Which is basically a set of Instructions to docker to create the different layers for an Image



≡ Setup a Simple Express Application — FIRSTLY

Terminal

```
> mkdir Docker-Crash-Course
> cd Docker-Crash-Course
> Code .                          # Open in VS Code
> mkdir api
> cd api

> npm init -y                     # Setup package.Json file
> npm install cors                # Install dependency
> npm install express             # ..  ..
> npm install --save-dev nodemon  # ..  ..
```

+ app.js

☐ ~~package~~ ≡ Create Js file and then create a simple express API

```
const express = require('express')
const cors    = require('cors')


const app = express()                    // Creates our application


app.use( cors() )


app.get('/', (req, res) => {
  res.Json([
    {
      "id": "1",
      "title": "Book Review: The Bear & The Nightingale"
    },
    {
      "id": "2",
      "title": "Game Review: Pokemon Brillian Diamond"
    },
    {
      "id": "3",
      "title": "Show Review: Alice in Boderland"
    }
  ])
})


app.listen(4000, () => {
  console.log('Listening for requests on port 4000')
})
```

package.Json

```
{
  ...
  "scripts": {
    "dev": "nodemon -L app.Js"
  },
  ...
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.17.2"
  }
}
```

Next: Install the official VS Code Docker Extension

Let's Start the Creation of our Docker File

Dockerfile # Create Dockerfile (No .ext)

Parent Image (Layer 1)

1 FROM node:17-alpine # Installs the node version 17 and alpine lightweight linux OS Image as the First layer of our IMAGE - Also it installs the Image from the docker repo if not installed already

2 WORKDIR /app # Specifies that our working directory would be a created folder in our current root directory app. This is where we will copy all our Source code to and where we will install all of our dependencies noted in our copied package.Json file

3 COPY . .

Source Code (Layer 2)

This line copies all our source code from (.) Current root directory to (.) Current Working Directory we specified for our Image in Line 2

Dependencies (Layer 3)

4 RUN npm install

This command would install all the dependencies at build time listed in the package.Json. But since we changed our working Dir.. it would look for package.Json inside □app folder and then install the dependencies in that folder.
- Which is what we want ☺

5 EXPOSE 4000

To make request to this API, we need to send requests to our Container using the specified port number
- If we are using CLI, this line would be useless to us and not needed.
- But since we will be using the Docker desktop it is pretty Important

6 CMD ["node", "app.js"]

This specifies any commands that need to be run at runtime when a Container has been generated
- Note: Not during build time

DOCKERIGNORE #6.

Now we want to actually build this image. But before we do that we want to ignore the node_modules folder when copying our source code

REASONS

- Because the node_modules folder ends up getting created during the Images build time. Thanks to our dockerfile command

  * RUN npm install

- Because the node_modules folder takes too long to copy and in general is not best practice to have it copied with the Source Code

≡ Let's Ignore the node_modules folder

.dockerignore # Create file.

node_modules # Now node_modules would be ignored during the Copying of our source code in build time

≡ Finally, Create an Image

> cd api

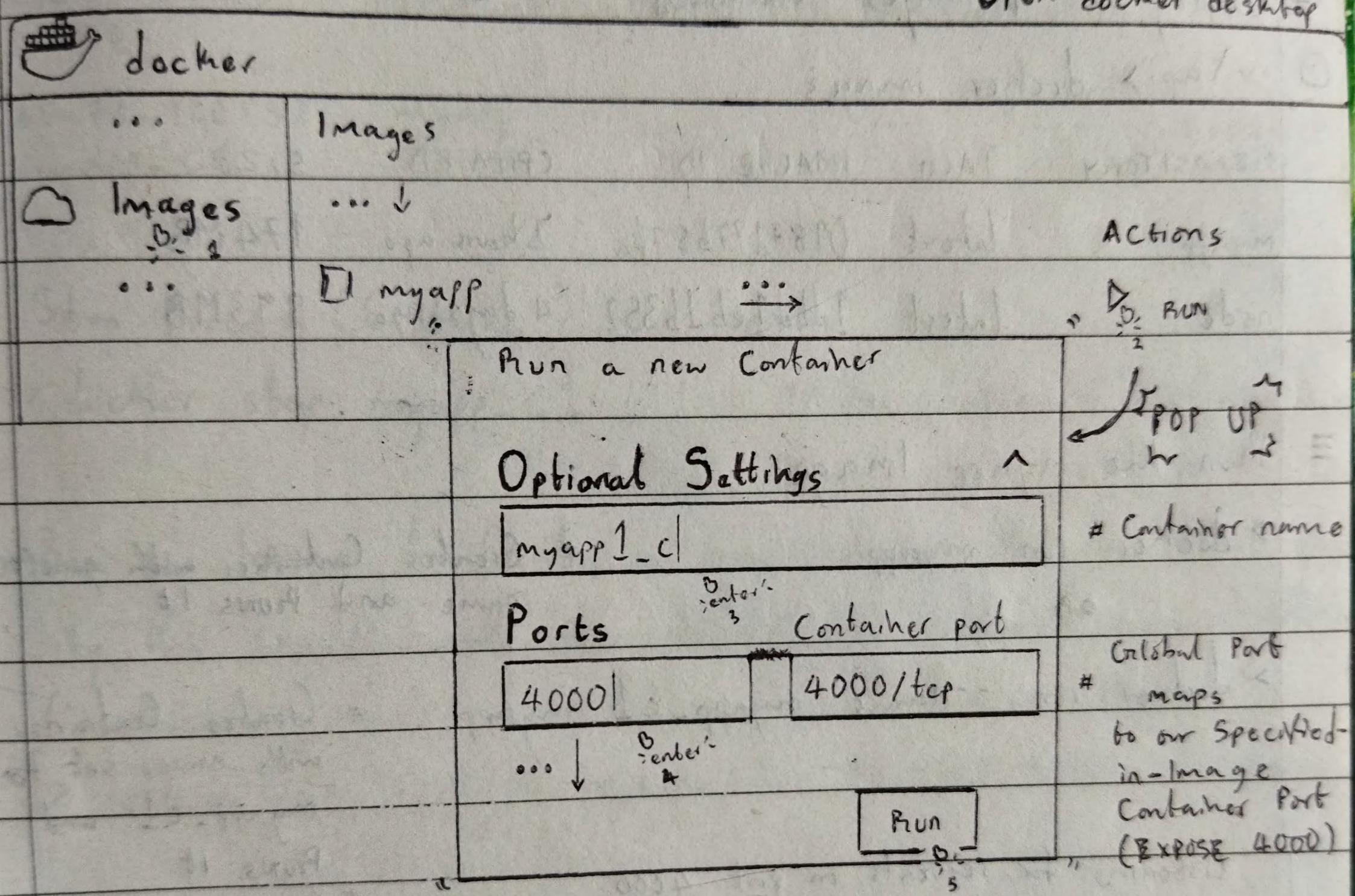
> docker build -t myapp . # -t means tag and it allows us to name the image. In our case we went with my app. The . is a relative path to our dockerfile

☑ Done: We wont see the folder after running the above Command but open your docker desktop app to see the actuall image

STARTING & STOPPING CONTAINERS #7

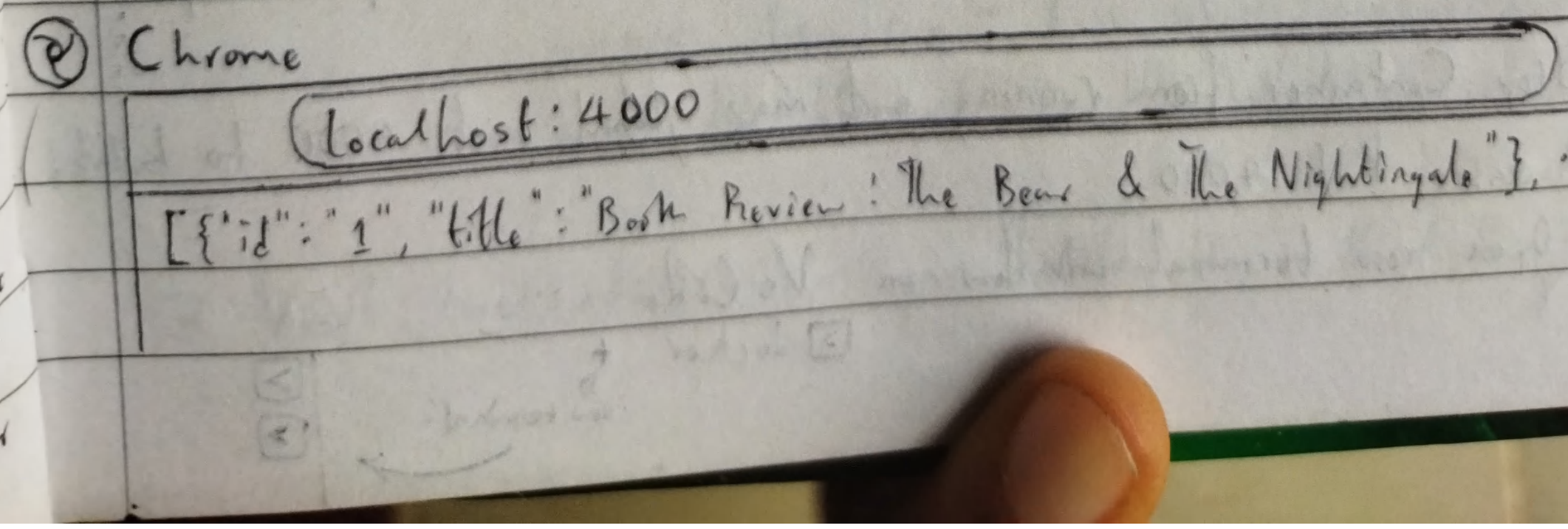
We know how to run an Image to spin-up a runnable instance of that Image (A Container) - Using the Docker Desktop App.

Open docker desktop



If we visit localhost:4000 on a browser - we should be redir-ected to our container port 4000. BUT

VISIT



We ~~A~~ want to now See how we can run an image to create a container and how to stop a running container from the CLI

Display list of Images Command ① Use VsCode Terminal

> ...\api> docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
myapp	latest	0987d77b87fa	2 hours ago	174MB
node	latest	1dbd1eb1b352	4 days ago	993MB

Run the myapp Image

> docker run myapp # Creates Container with generic name and Runs It

OR

> docker run --name myapp_c1 myapp # Creates Container with name set to myapp_c1 and Runs It

listening for requests on port 4000 # terminal blocked

NOTE: If we visit the port 4000 on our browser, We want have access to our running "Isolated process" because we need to map our global (Computer) port to our Isolated process port

Stop Container from running and map global port 4000 to Isolated Processes Port 4000.

- Open new terminal window on Vs Code

> docker + (new terminal)

≡ Show List of Running Processes (or Containers)

▷ Terminal (2nd)

```
> docker ps                    # This command shows all running processes
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS
NAME
40a73a960352    myapp    "..."      ...        ...       4000/tcp
myapp-c1
```

≡ Stop Process (Container) from running

```
> docker stop myapp-c1         # either Container id or name
```

≡ Run our Image to Create a new Container, But this time specify the Computer port to map to the Isolated process (container) port

```
> docker run --name myapp-c2 -p 4000:4000 -d myapp
```

Explanation

--name allows us to name the Container to be created

-p allows us specify the port mapping (Computer port : Isolated port)

-d allows us detach the terminal from the running process when ran - this simply means our terminal won't be blocked after running the Command

☑ Visit localhost:4000 on your ~~desktop~~ browser to see that this works

≡ Show List of all Processes

> docker ps -a #-a means all

...

≡ Show List of Running Processes

> docker ps

...

myapp_c2 ...

≡ Stop running process / container

> docker stop myapp-c2

≡ To Run an Already created / Existing Container

> docker start myapp-c2 # Already specified our port mapping and also the detach during container creation - So we have 0 worries

myapp-c2

☑ Visit localhost:4000 to see that we are still Gucci

FACT

- We can do all this from either the docker desktop or the CL
- But from henceforth we will only be using the CL

LAYER CACHING #8

So far we've seen how to build images, run images to start up a container, and also how to stop containers - And all this is fine **BUT** however their are Improvements we can make

Dockerfile ≡ Current State of Our Dockerfile

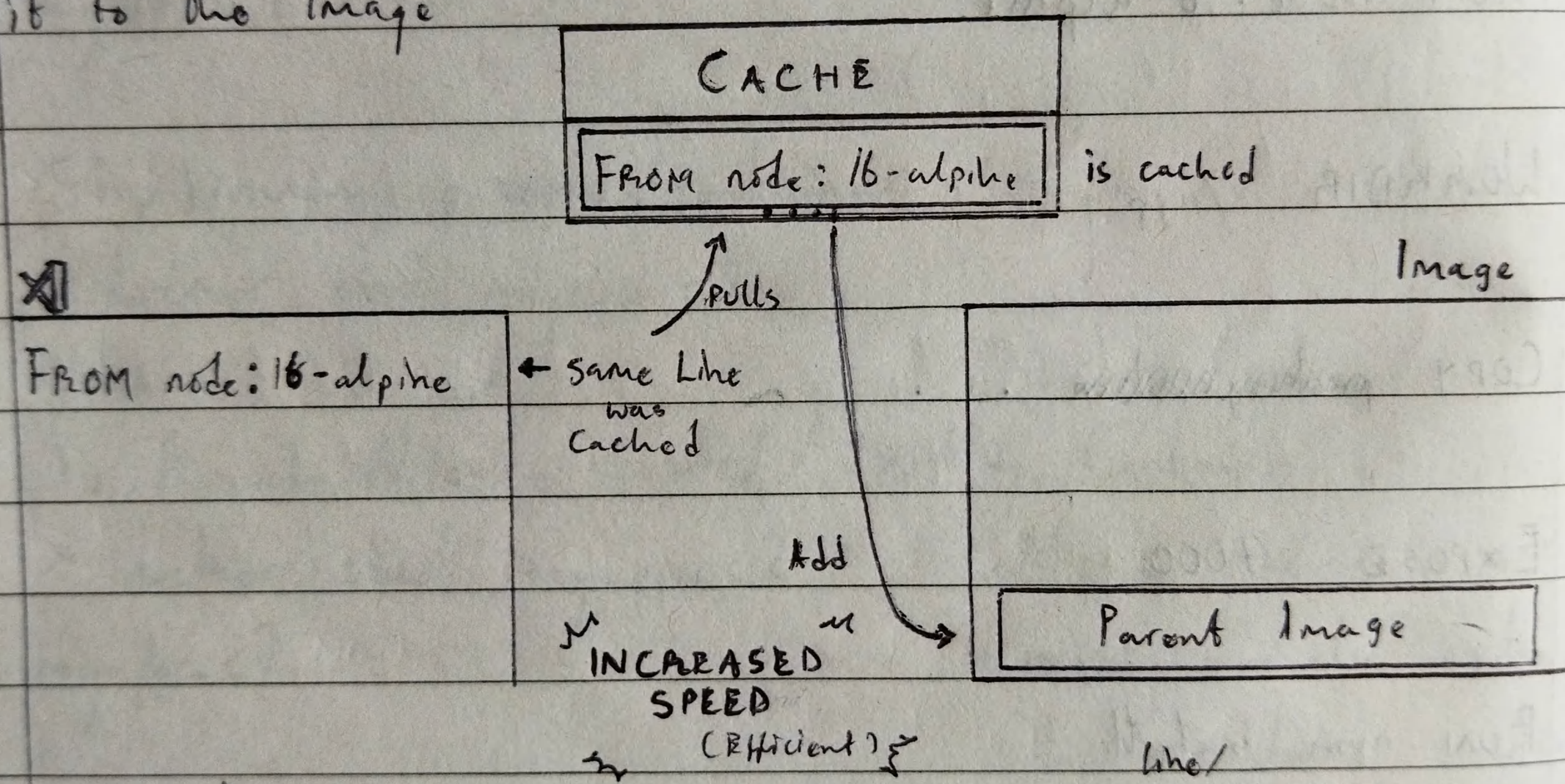
```
FROM node:16-alpine

WORKDIR /app

COPY . .

EXPOSE 4000

RUN npm install

CMD ["node", "app.js"]
```

- Docker Layer Caching

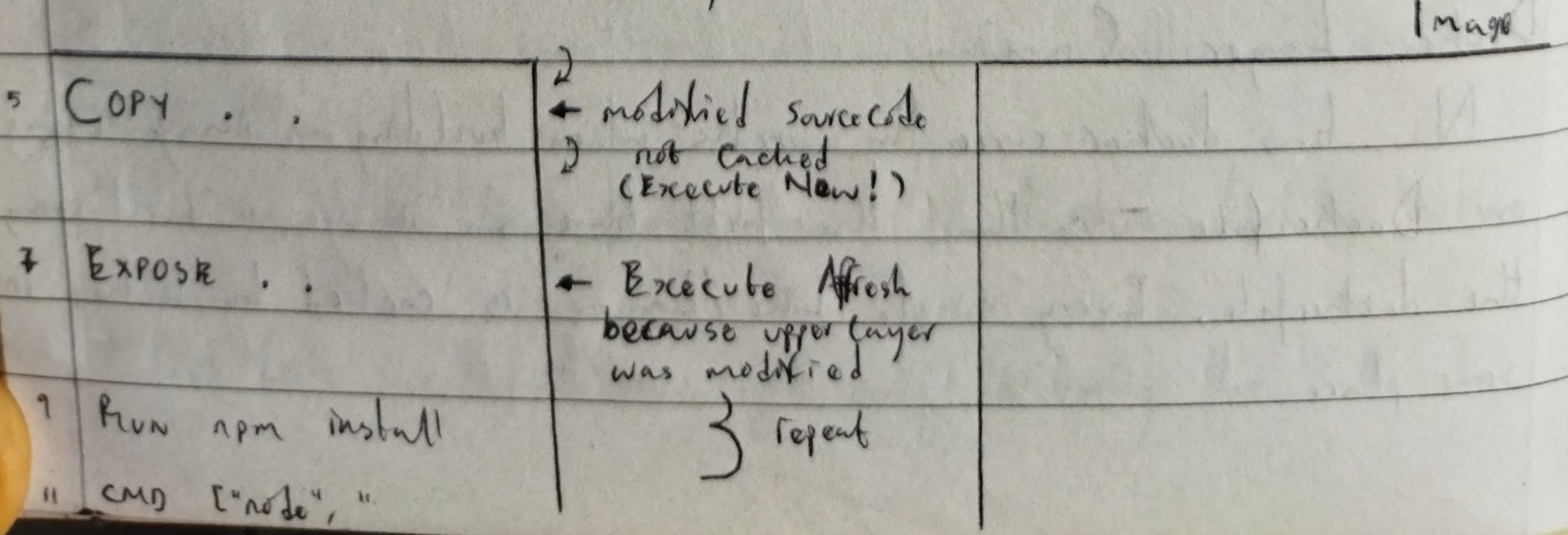
Now how docker currently works when building an image using our Dockerfile - is that the first time an image is built using the dockerfile. Every single Line/Layer is cached by docker in some place

Then the next time we try to build an image with that same Dockerfile

Docker checks the first layer in the Dockerfile to see if anything changed - If nothing did, instead of running a new execution for that line - Docker pulls the cached one and adds it to the Image



Docker repeats this process for all the line/layer that has already been cached (from top-to-bottom) until it enters a line/layer that was modified (typically the Source Code)



TO SEE PROVE
Build 2 seperate images with a Dockerfile. Then check the terminal to see Layers CACHED caches that were used.

Because Line 5 was modified and not the same as the previously cached one - Docker re-runs that command / line without grabbing the cached version

Docker repeats this for subsequent Lines whether they were modified or not. Meaning - We Run npm install again whether or not it changed. Which is not EFFICIENT at All.

Fix

- We want to run npm install before we Copy our Source Code layer into our Image. This means that we have to first off Copy the package.Json file into our Image before running our dependency installation with RUN npm install

Dockerfile

```
FROM node:16-alpine
WORKDIR /app
COPY package.Json .          # Copies Just package.Json into Image
RUN npm install              # Runs dependency installation using package.Json
COPY . .                     # Copies the entire source code to Image
EXPOSE 4000
CMD ["node", "app.Js"]
```

Now with this Improvement even If our source Code changes we wouldn't have to re-execute our dependency Installation - when it is already Cached

MANAGING IMAGES & CONTAINERS # 9

We want to now learn how to add versions (or tags) to Images and Containers and how to manage Images and Containers

② Terminal

> docker images ≡ Displays all Images

...

> docker ps ≡ Show all Running Containers

...

> docker ps -a ≡ Show all Containers

...

> docker image rm myapp ≡ Delete/Remove an Image

NOTE: This Command wont work if their exist Containers that are using this Image {myapp}. Tip: Delete Containers

> docker image rm myapp -f ≡ Delete/Remove Forcefully

...

> docker container rm myapp_c2 ≡ Delete/Remove a Container

> docker build -t myapp2 . ≡ Create Image

> docker run --name myapp2_c -p 4000:4000 myapp2 ≡ Create & Run Container

> docker stop myapp2_c ≡ Stop a Running Container

> docker start myapp2_c ≡ Start Running a Container

> docker system prune -a ≡ Delete/Removes all Images and Containers

VERSIONS

When creating an Image we can specify a version or tag name for that image

> docker build -t myapp:v1 . ≡ Creates an Image with a version/tag name

> docker Images ≡ Show all Images

REPOSITORY	TAG	IMAGE-ID	CREATED	SIZE
myapp	v1	4adb3be8a805	15 minutes ago	114 MB

> docker run --name myapp-c -p 4000:4000 myapp:v1 ≡ Runs the particular Image version myapp with tagname of "v1"

VOLUMES #10

Currently in our project, we have a tiny issue which is whenever we make a change to the source code in order to actually see that change we would have to create a new Image and run that Image to get a running container that then reflects our changes. { FACTOS }

This can be quite tedious, and a tiring way to build repetitive applications. To solve this problem docker introduced <u>volumes</u>.

WHAT ARE VOLUMES?

Volumes are a feature of docker that allows us to specify folders on our host computer that can be made available to running containers

This folder is watched so that any changes that occur in that folder is then automatically replicated in the ISOLATED PROCESS's folder to match that of the watched (mapped to) folder

Hence

We want to map our projects development folder ○ api that contains all our source code to our ○ app folder in our container ~~(which is invisible on Vs code)~~, So that when we write more code

in addition to our source code - it automatically reflects on our Container.

NOTE: This doesn't make any changes or update to our Image. Our Image is read-only. Therefore when we are done we need to create a new Image that holds the most recent source code before deploying the app container or sharing it with friends / team members

How Do We Add Volumes To Our Project?

We can do that by adding a simple -v flag to the run command. But before adding a volume we need to add a new dependency in nodemon

≡ Install nodemon / Add Installation Command to Dockerfile

Dockerfile

```
FROM node:16-alpine

RUN npm install -g nodemon          # Installs nodemon globally
                                    on our Image during build time
```

≡ Add new command to script that Uses nodemon to run our app

package.Json

```
{ ...
  "scripts": {
    ...
    "dev": "nodemon -L app.Js"      -L is needed to alter the way n
                                    # -mon works with docker (must for
```

≡ Update CMD in our Dockerfile to use our newly added Script

O Dockerfile

...

U CMD ["npm", "run", "dev"]

WHY DO WE NEED NODEMON ?

Nodemon helps us restart the server when any change is detected - so those changes can be reflected without having to manually restart the server to see them.

NOTE: Nodemon is typically used during development and it is not a needed package for production

≡ Create new Image with a tag - nodemon

> docker build -t myapp:nodemon .

...

≡ Create a Container with a Volume Setup

> docker run --name myapp-c-nodemon -p 4000:4000 --rm -v C:\Users\shawn\Documents\Tuts\docker-crash-course\api /app myapp:nodemon

absolute path to watch (our development folder) on host computer

Container folder

Tip: --rm flag means remove/delete the Container automatically when you stop the Container (from running)

A Little Problem

If we where to delete our nodes_module folder on our root computer, this means that the node_modules folder in our container would be removed as well

ROOT COMPUTER		CONTAINER (Isolated Process)
v DOCKER-CRASH-COURSE		v app
v api		
> node_modules		> node_modules
.dockerignore		.dockerignore
app.js	IN SYNC	app.js
Dockerfile	⟷	Dockerfile
package-lock.json		package-lock.json
package.json		package.json

Both Root Computer & Container are in sync. Any changes to the api folder is replicated in the Container. So if we delete the node_modules folder in the root computer. It is replicated in the Container and we don't want that to happen

Exempt node_modules from our Volume (Using Anonymous Volume)

> docker run --name myapp_c_nodemon -p 4000:4000 --rm -v C:\Users\Shawn\Documents\Tuts\docker-crash-course\api:/app -v /app/node_modules

☑ Lastly, update app.js then save to see instant updates

Docker Compose #11

So far we've seen how to make an Image, run the Image to spin-up a Container and also how to use Volumes to map our local project to a folder app in the Container.

Now to use Volumes we are required to type out that pretty long command in the terminal and sometimes we might have multiple projects that we want Containers for and we want to be able to run all those Containers **at once**

For Example

We may have a database in addition to our api express app and also a React app. And we would want to run all the images of this 3 apps to spin-up a Container and have these applications working together

Brute Force Solution

We could carry on as we have been and use a docker build & a docker run to spin up Containers for each app but that would be very tedious and tasking and inefficient.

"Say Thanks to Docker Compose"

What is Docker Compose?

It is a tool that comes with docker and - It is used for defining and running multi-container Docker applications

With Docker Compose, you can define the services that make up your application (Your Image) in a .yaml file, and then use docker compose to start and stop all running Containers

Create a Docker Compose file with a Service setup for our api

Docker-Crash-Course

> api

docker-compose.yaml # Add in main root folder

docker-compose.yaml

```
version: "3.8"                    # Docker Compose Version

services:                         # Services Usually have multiple Services / break to next line

  api:                            # api application added. It has multiple values to
    build: ./api                  # relative path to Dockerfile to be used to build image
    container_name: api-c         # Self-explanatory

    ports:
      - '4000:4000'               # Port mapping

    volumes:
      - ./api:/app                # Relative path to host container folder :
      - /app/node_modules         folder in container
                                  # Exempt node_modules from volume
```

Note: I speak in plural because docker-Compose usally runs several services (Creates Images and several Containers each for those Images)

to add to our project

Now lets say we created a new React app. Info could add that also into our docker-Compose file as a new service

Use

Docker Compose to Create And run our Service ▷ api

Terminal

```
> docker system prune          # Delete all Images & Containers

> cd ..\

... docker-Crash-Course > docker-compose up
...                          # Creates Images and runs Containers
```

Result

The above Command finds the docker-compose file and it gets all the services and then with our specified configurations in each it uses that to build Images and then immediately runs all the Images to get Containers.

In our case, since we only have 1 service their we would have 1 Image and a Container of that Image running

```
> docker images              # To see the Images if created
> docker ps                  # To see the running Containers

> docker-compose down        # Stops and deletes Containers
> docker-Compose down --rmi all # Stops & deletes Containers, and Images
> docker-Composed down --rmi all -v # deletes Containers, Images and Volumes
```

Dockerizing a React App #12

We want to now learn how to dockerize a frontend application, namely react

- Docker-Crash-Course
  - > api
  - + v myblog # Create react app => npm create-react-app myblog (Terminal)
    - ...
  - + .dockerignore # ignore node_modules => node_modules (Inside file)
    - ...
  - + Dockerfile # Add Instructions for Creating an Image
    - ...

- Dockerfile

```
FROM node:17-alpine                # React Apps require a node Parent Image

WORKDIR /app                       # Select Images Working Directory

COPY package.Json .                # Copy package.Json into Images WD

RUN npm install                    # Install all dep using package.Json copied

COPY . .                           # Copy all Sourcecode and all info Images WD

EXPOSE 3000                        # Expose Isolated port for accessing app by the Container
CMD ["npm", "start"]               # Specify Command to be ran upon Creation
```

Add React app as a new Service in our Docker Compose file

docker-compose.yaml

```
version: "3.8"

services:

  api:

    ...

+ myblog:
    build: ./myblog                    # where to find Dockerfile to build
    container_name: myblog-c
    ports:
      - '3000:3000'
    volumes:
      - ./myblog:/app
      - ./myblog:/node_modules
    stdin_open: true
    tty: true
```

NOTE:

Volumes work on Linux & mac for React Applications. But don't work on Windows because how docker works with WSL is different.

We could use a flag to by pass this

-e CHOKIDAR_USEPOLLING=true

But it only works occassionally

NOTE:

Both of this properties basically start the Container in Interactive mode. Which is what we should do with React Apps in docker

Implement Simple React App that works with our Express API

myblog > Src > App.Js

```
import { useEffect, useState } from 'react';

function App() {
  const [blog, setBlog] = useState([]);

  useEffect(() => {
    fetch('http://localhost:4000/')
      .then(res => res.Json())
      .then(data => setBlogs(data));
  }, []);

  return (
    <div className="App">
      <header className="App-header">
        <h1> all blogs </h1>
        { blogs && blogs.map(blog => (
            <div key={blog.id}> { blog.title } </div>
        ))}
      </header>
    </div>
  );
}
```

Use docker-Compose to run both Apps

Terminal

```
> docker-compose up
...
```

☐ After running, test apps to see that they both work and the React App actually fetches data from our API

Visit Port to See that our React Apps Container is Running

CHROME

localhost:3000

all blogs

Book Review: The Bear & Nightingale ABC

GAME REVIEW: Pokemon Brilliant Diamond

Show Review: Alice in Borderland

SHARING IMAGES ON DOCKER HUB #13

Just like GitHub where we share our code so other people can download and use them. On docker hub we can also share our Images so other people can download and run them

"This is typically how we share Images with Others"

A WISE MAN ONCE SAID.

STEPS

- Visit hub.docker.com
- Register or Sign in (only if you are new or logged out)
- Scroll down and click on

Create a Repository
Push Container images to a repo on Docker

...

Create Repository

thenetninjauk | v myapi

would be your username

b/ enter name

b/ click to create New Page

Cancel | Create

b/ create repo

...

CREATION SUCCESSFUL!!

thenetninjauk/myapi

Docker Commands

To push a new tag to this repository,

```
docker push thenetninjauk/myapi:tagname
```

(so we can push it to our repo)

- Now we want to build our ▢ api image only so therefore we won't be using docker Compose for this

▷ Terminal ≡ Build Only the ▢ api app Image

```
> cd api
api> docker build -t thenetninjauk/myapi .
...
```

☑ Image built

```
> docker Images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
thenetninjauk/api	latest	de667de49760	3 hours ago	119MB

≡ Before pushing to our repo Login to Docker

```
> docker login
> Enter Username : ......
> Enter Password : ........
...
```

☑ Done. We can now PUSH

≡ Push Our ▢ api app Image to Our Created Repo

```
> docker push thenetninjauk/myapi
...
```

☑ Confirm that it pushed successfully on docker hub

≡ Pull our ▢ api app Image from docker hub

```
> docker image rm thenetninjauk/myapi   # Deletes Image
...
> docker pull thenetninjauk/myapi
```

☑ DONE ☑ COURSE COMPLETED