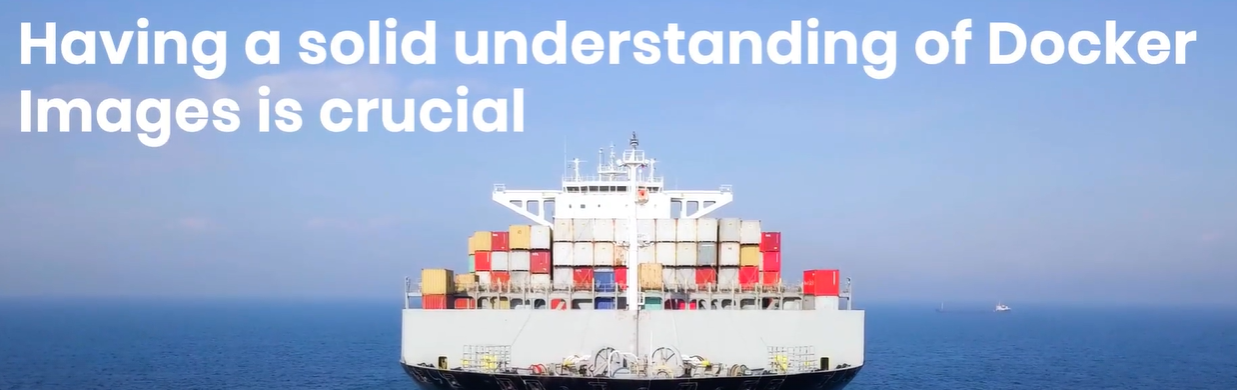
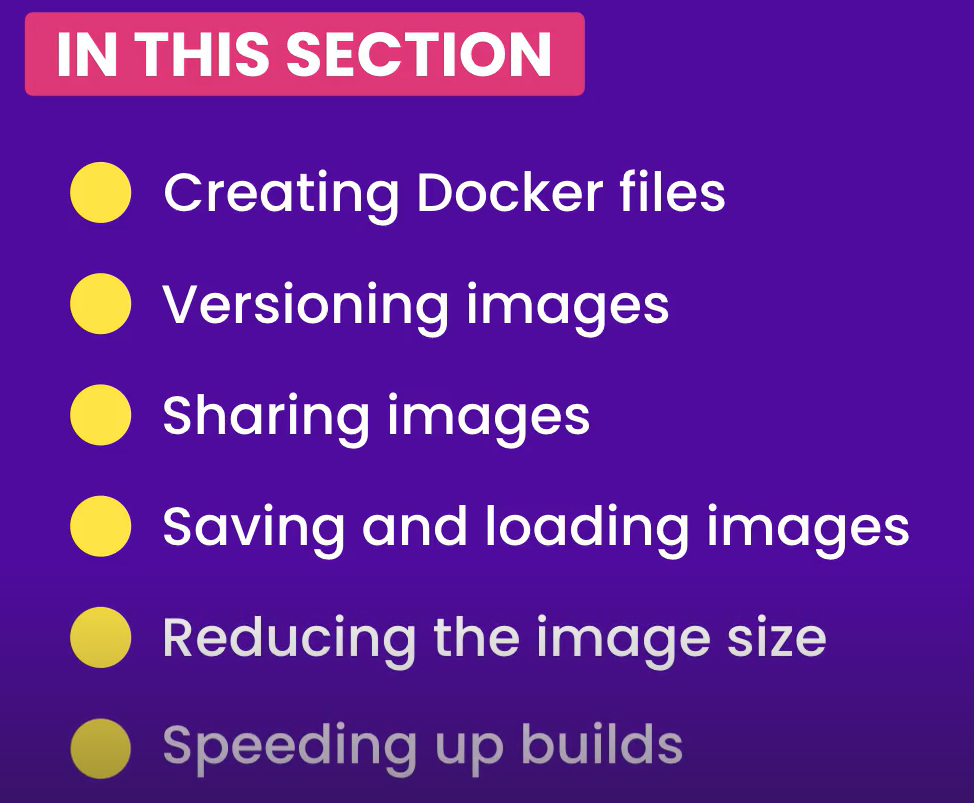


**Introduction**:

The first step in using docker to build and deploy applications is creating images.



And we will learn,



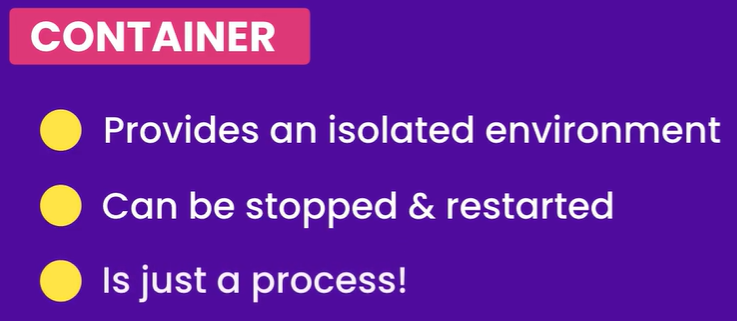
**Images and Containers**:

Before we start, let’s make sure we have right understanding of images and containers.

An image includes everything (*files and configuration setting*)an application needs to run, so it contains,

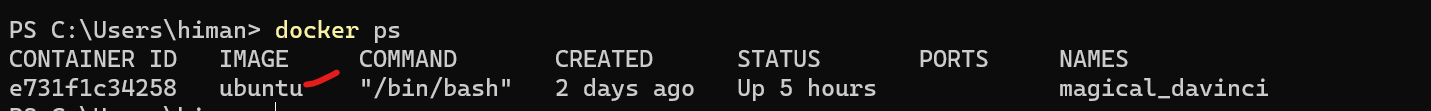


Once we have an image, we can start a container from it. A container is a *kind – of a virtual machine* and it…



*Container is just a process but a special one, since it has its own file system provided by the image*.

In the previous section we started a container from an ubuntu image.



Let us start a new container with the same image and see what happens.

docker run -it ubuntu

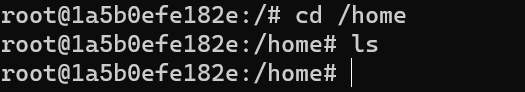


A new container has been started with a new container id.

In the previous section, inside our first container we created some directories and a deployment script in our home directory.



Back to our new container,



There is nothing here because a container gets its file system from the image, but *each container has its own write layer. So what we write in a given container is invisible to other containers*.

Of course there is a way to share data between containers(*later in the course*).

But here is the takeaway,



Each container is an isolated environment for executing an application(*an isolated universe!*). So whatever happens inside that universe is invisible to other containers.

**Sample Web Application**:

Over the next few lessons, we are going to take a frontend application built with react and package it into a docker image.

A sample react application is inside *section4-react-app* folder. There is a *package.json* file here,

{

  "name": "react-app",

  "version": "0.1.0",

  "private": true,

  "dependencies": {

    "@testing-library/jest-dom": "^5.11.4",

    "@testing-library/react": "^11.1.0",

    "@testing-library/user-event": "^12.1.10",

    "react": "^17.0.1",

    "react-dom": "^17.0.1",

    "react-scripts": "4.0.3",

    "web-vitals": "^1.0.1"

  },

  "scripts": {

    "start": "react-scripts start",

    "build": "react-scripts build",

    "test": "react-scripts test",

    "eject": "react-scripts eject"

  },

  "eslintConfig": {

    "extends": [

      "react-app",

      "react-app/jest"

    ]

  },

  "browserslist": {

    "production": [

      ">0.2%",

      "not dead",

      "not op\_mini all"

    ],

    "development": [

      "last 1 chrome version",

      "last 1 firefox version",

      "last 1 safari version"

    ]

  }

}

This package.json file is like an identification card for our application.

Focus on the dependencies object,

  "dependencies": {

    "@testing-library/jest-dom": "^5.11.4",

    "@testing-library/react": "^11.1.0",

    "@testing-library/user-event": "^12.1.10",

    "react": "^17.0.1",

    "react-dom": "^17.0.1",

    "react-scripts": "4.0.3",

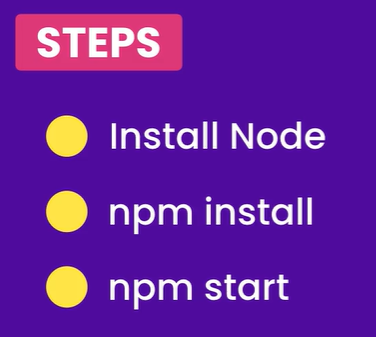
    "web-vitals": "^1.0.1"

  }

These are 3rd party libraries our application needs to run.

Now let’s say we want to take this application and run it on a brand new machine (*with nothing except OS like windows or linux*),

there are a number of steps we have to follow in that case.



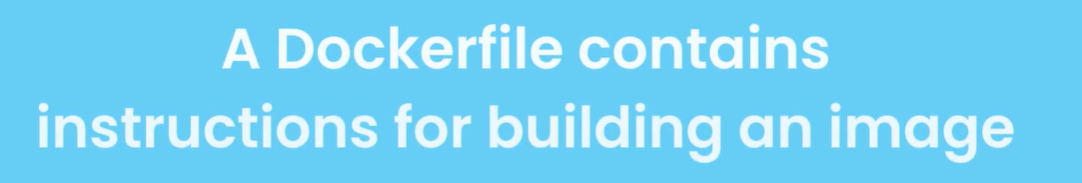
We have the same concept in other development stacks whether its c#, python, java or ruby, we always have some tool to manage the dependencies of our application and then start it.

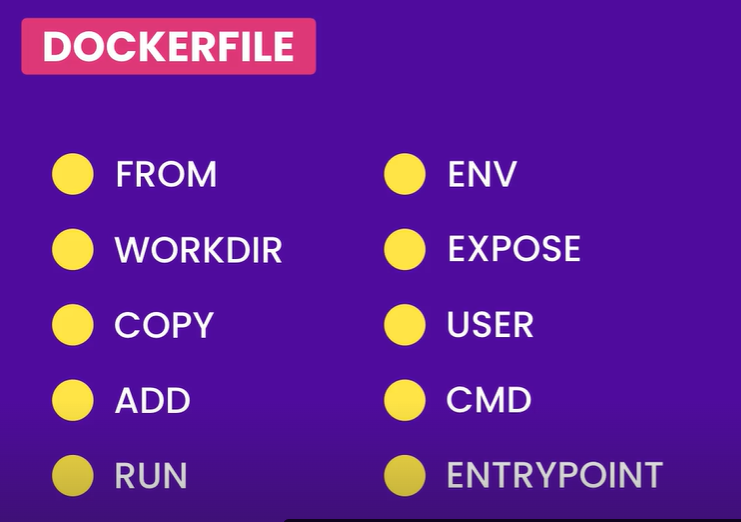
Let’s see how we can use docker, so that we don’t have to repeat all these steps every time we want to deploy this on a new machine.

Throughout the rest of this section, we are going to dockerize this application and package it into an image. Once we have that image we can deploy this application virtually anywhere.

**Dockerfile Instructions**:

The first step to dockerize an application is adding a *Dockerfile* to it.





*From* 🡪 For specifying the base image. *Base image contains a bunch of files and directories and then we build on top of it*.

*WORKDIR* 🡪 For specifying the working directory. *Once we use this command all the following commands will be executed in the current working directory*.

*COPY* & *ADD* 🡪 For copying files and directories.

*RUN* 🡪 For executing operating system commands. *All the linux commands we learned can be executed using run command*.

*ENV* 🡪 For setting environment variables.

*EXPOSE* 🡪 To tell docker that our container is starting at a given port.

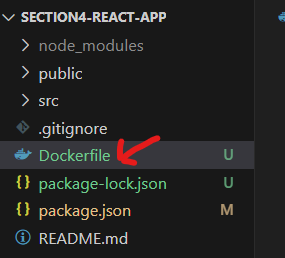
*USER* 🡪 For specifying the user that should run this application. Typically *we want to run our application using a user with limited privileges*.

*CMD* & *ENTRYPOINT* 🡪 For specifying the *command that should be executed when we start a containe*r.

Over the next few lessons we will explore each of these commands in detail.

**Choosing the right base image**:

Alright let’s start off by adding a Dockerfile in the root of our project (*make sure its Docker****f****ile not Docker****F****ile*).



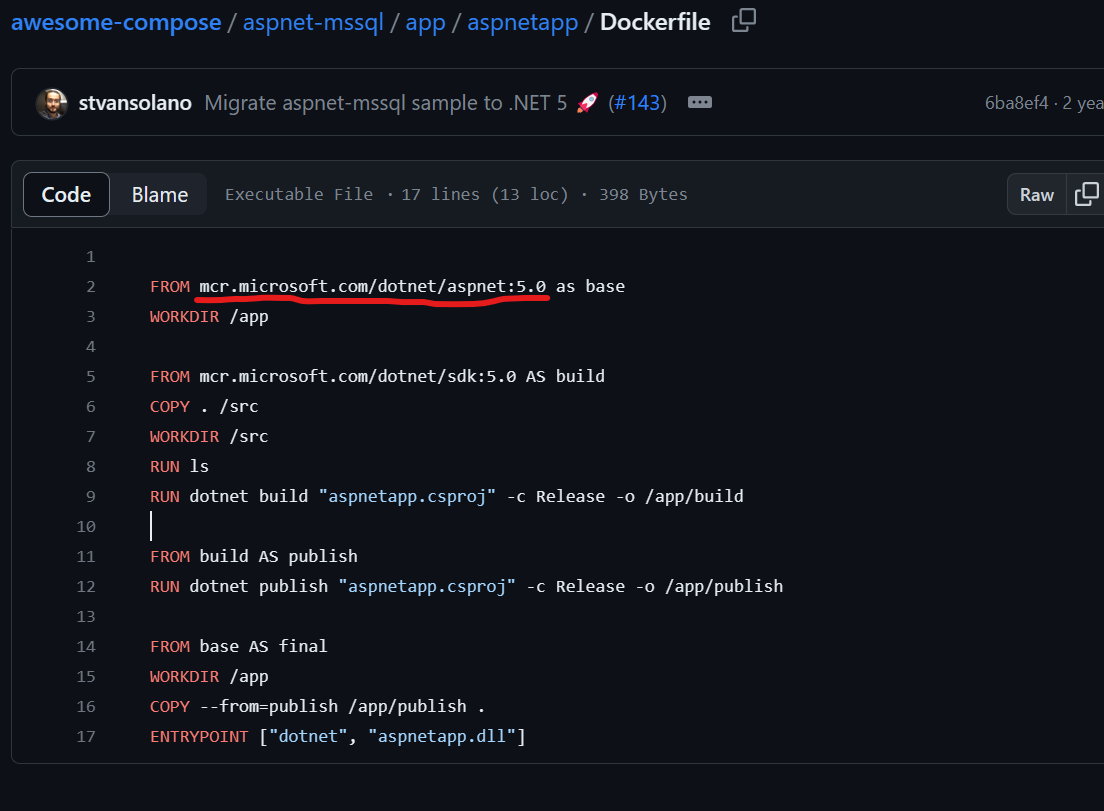
First instruction we are going to use is FROM for specifying the base image.

*The base image can be an operating system like linux or windows or it can be an OS + runtime environment*. For example if you are a c sharp developer, you can start from a *dotnet* image, python image for python developer and *node* image for a JavaScript developer.

[Samples overview | Docker Docs](https://docs.docker.com/samples/)

Here at *docs.docker.com/samples*, we can see various examples of docker files for different technology stacks.

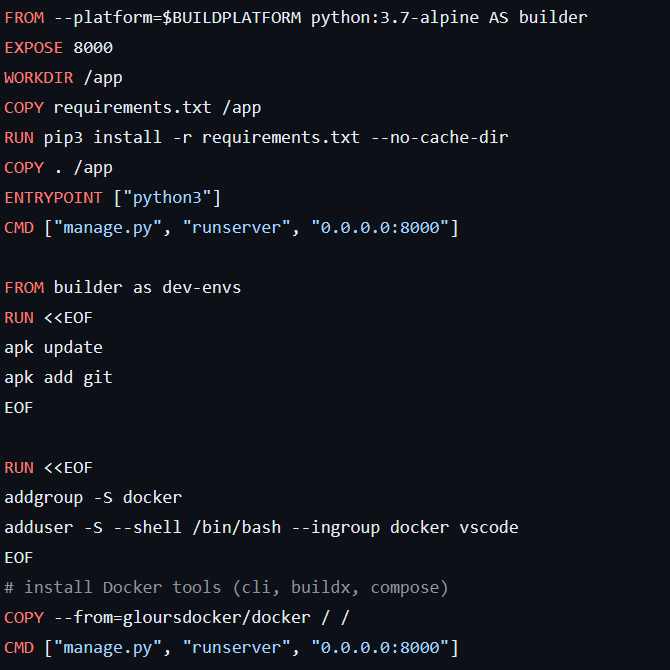
Take a look at Dockerfile for asp.net application,



Look at the *FROM* instruction, FROM *mcr.microsoft.com/dotnet/aspnet:5.0* as base, here URL starting from *mcr* means that this base image is hosted on *Microsoft container registry* not on docker hub (*default registry that docker uses*).

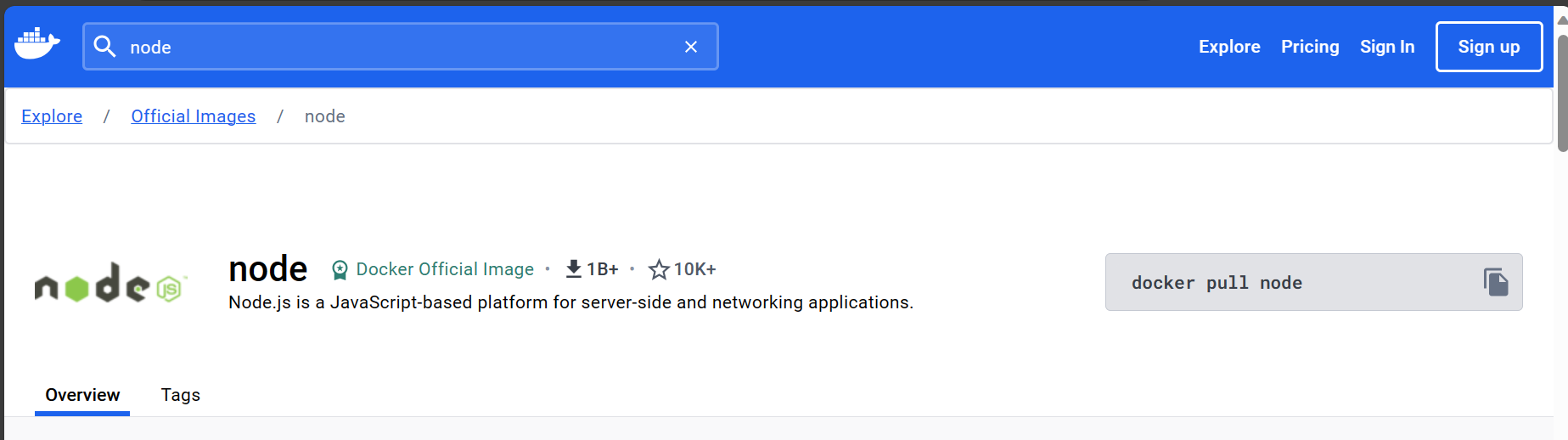
Note: For any images in any other registry then docker hub, we have to give the full URL. Don’t take URL blindly from sample Dockerfile because the version or URL can change, so always do your own research.

If you are a Django developer,



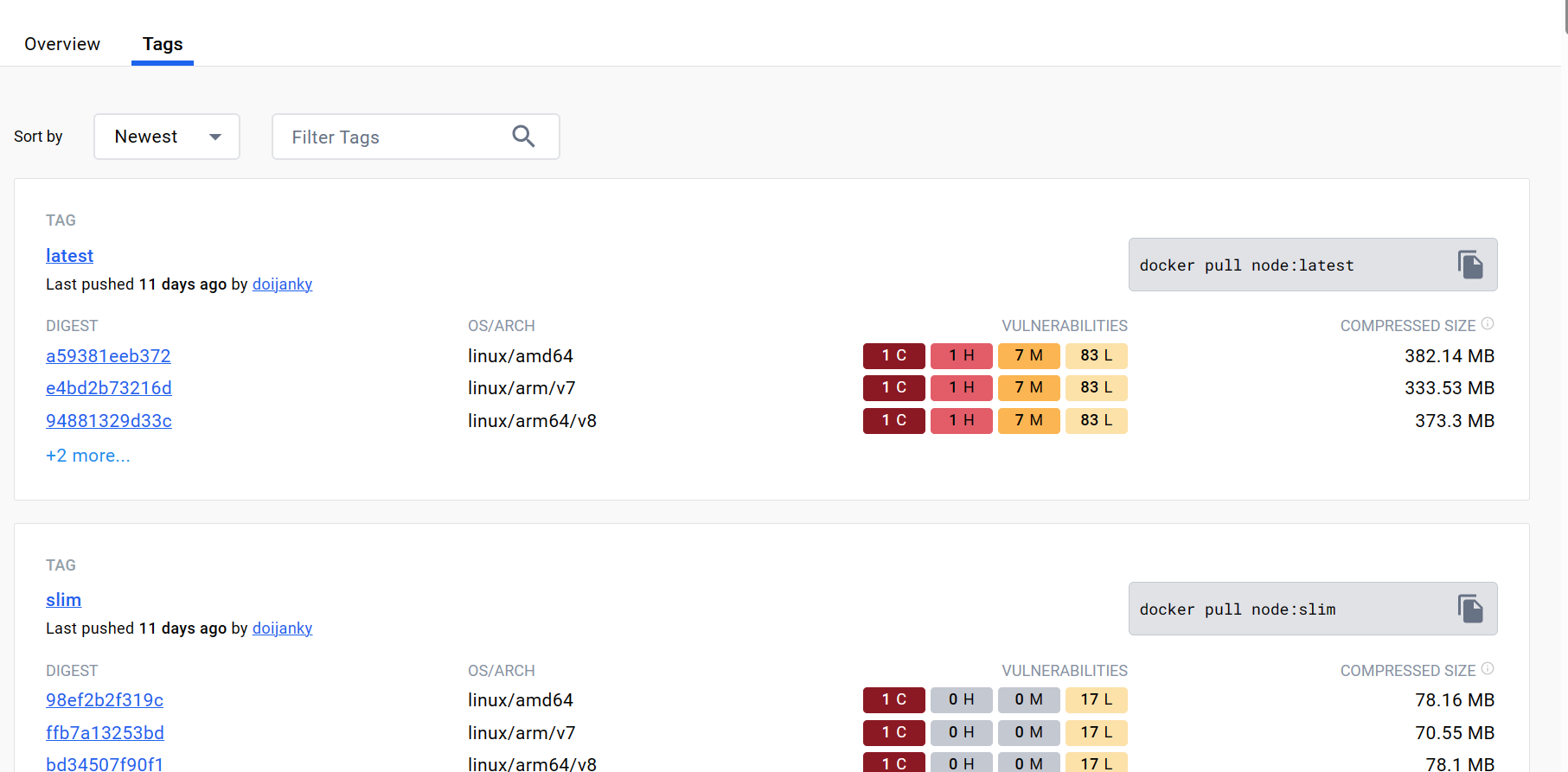
For this project, we need node because to run a react app we need NPM to install application dependencies and start the application.

So let’s go to hub.docker.com and search for node,



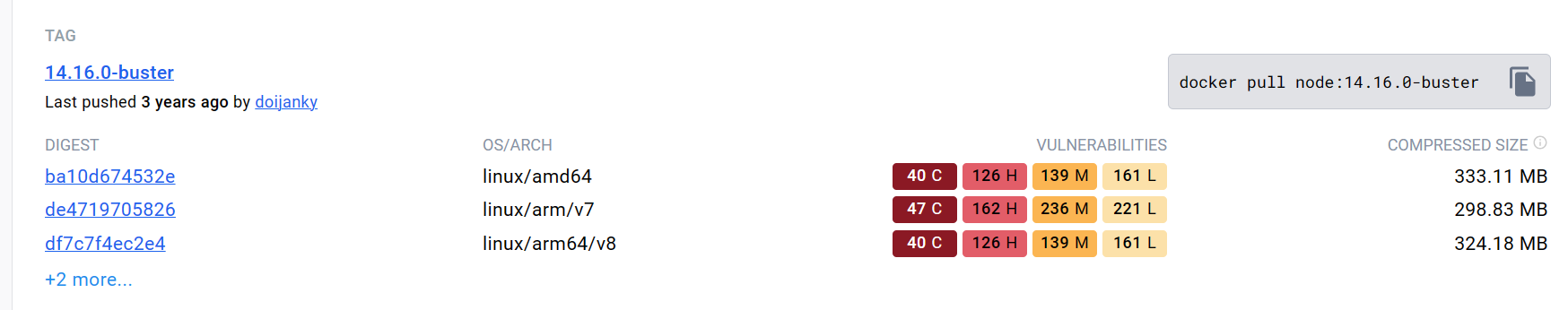
The node repository on docker hub has 100s of node images which can be a little bit confusing.

So navigate to tags,



Where we can see various node images or different versions built for different versions of linux.

For example,



We have node version 14.16.0 on top of buster(Debian linux version 10), so we have different versions of node on different versions of linux.

The image we choose really depends on our application like what version of node we want to target.

Note: Never specify :*latest* in front of node (or any technology stack) in FROM. For example,

FROM node:latest

//It is a BAD PRACTICE

Always use a specific version.

Back to docker hub, let’s say we need to build this application against node version 14, so if we search for it we find 100s of images with this version.



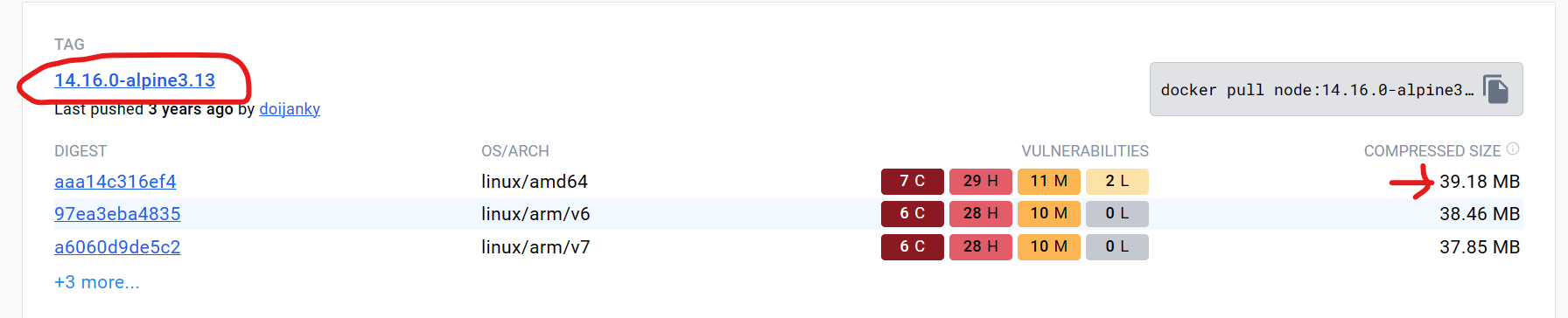
For example here, we do not have a minor build number, just a major version number. The size of this image is around 333 mb but its slightly different based on different OS and CPU architecture (*although its compressed size, it will be around 1GB after uncompressing*).

Note: If you want to run node on top of windows image, then start from a windows image and then install node on top of it. But *its not recommended because windows images are pretty large*(*over 2GB*).

So depending on our CPU architecture we should pull an image and docker will automatically download the right docker image for our CPU architecture.

In our case, we want to use smaller images because we need to make our builds and deployments faster.

So search for *alpine* on this page, since these images are really small.



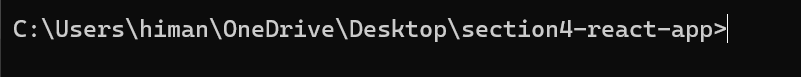
Its almost 10 times smaller than other version.

Copy its name and paste it inside our Dockerfile,

FROM node:14.16.0-alpine3.13

That’s the first step…

Now inside our project directory,



We are going to build an image by typing this command here,

docker build -t react-app .

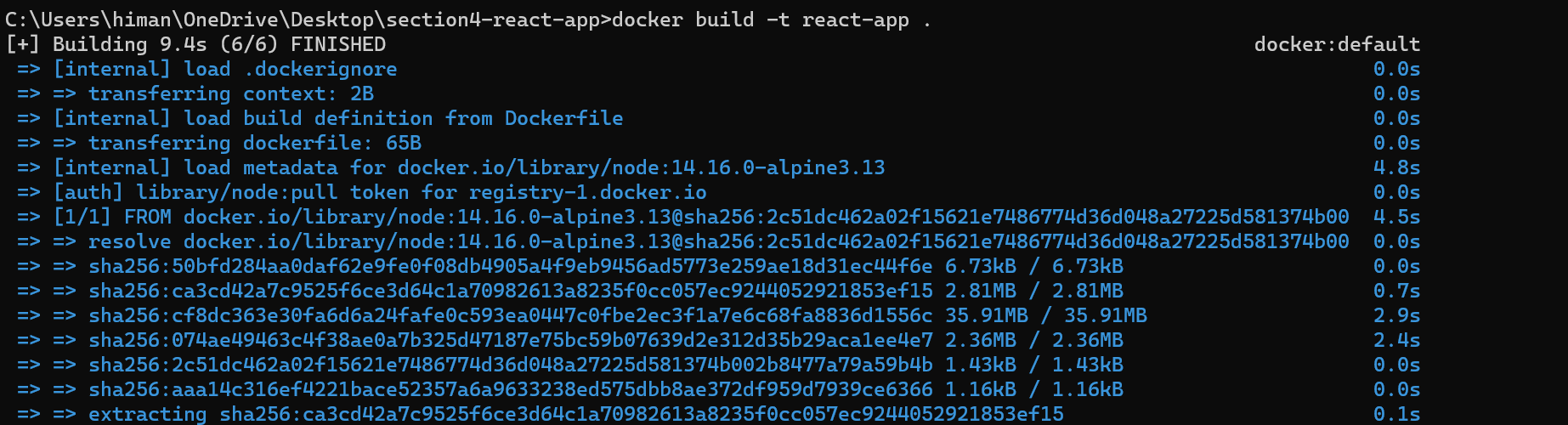
*Command breakdown*:

docker

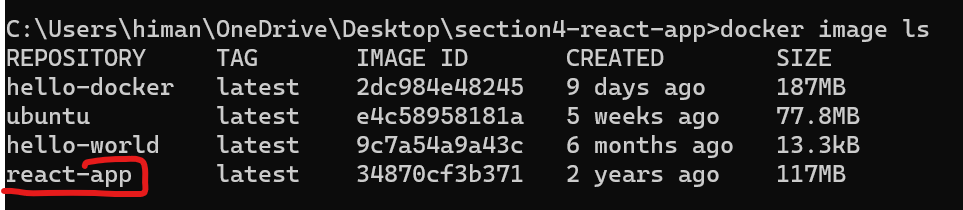
build -t : for tagging the image

react-app : name of the image, anything xyz…

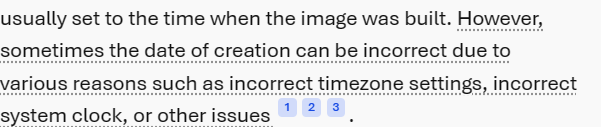
. : we type a period to tell docker that it can find Dockerfile in the current directory.



This image is successfully built now, so now we can look at all the images on this machine using docker images or docker image ls command.

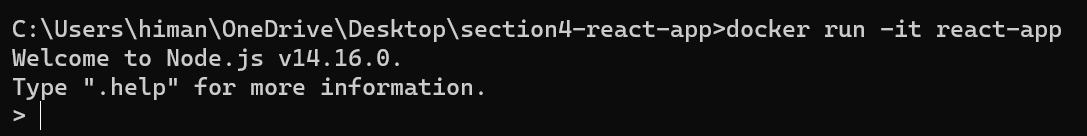


The created at is showing 2 years ago which us not right,

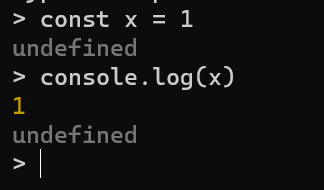


Now let’s start a container with this image and see what happens,

docker run -it react-app

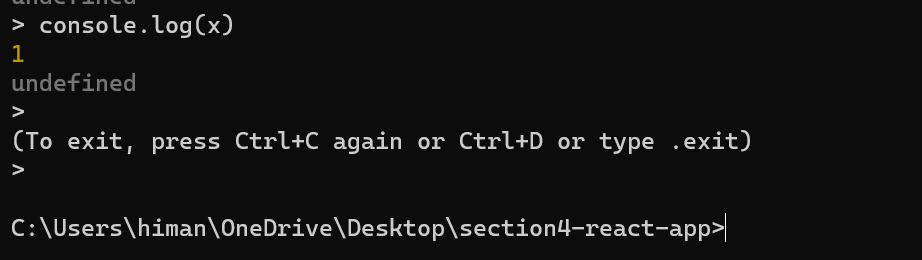


As soon as we run this command, we are inside a node environment. So here we can write JavaScript code and node will execute it.



For example, here we defined a constant x and then logged its value on the console.

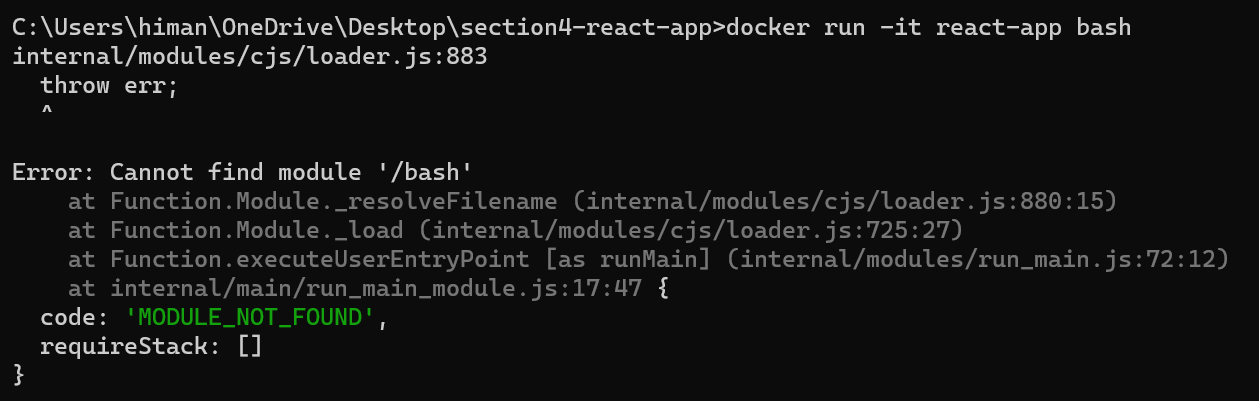
But we do not want this, we need to run bash, so that we can look at the file system. So press ctrl + C 2 times to exit.



The container is stopped now. So if need to start this container with *bash* specify the same from command line.

docker run -it react-app bash

But we get an error saying that this module is not found.

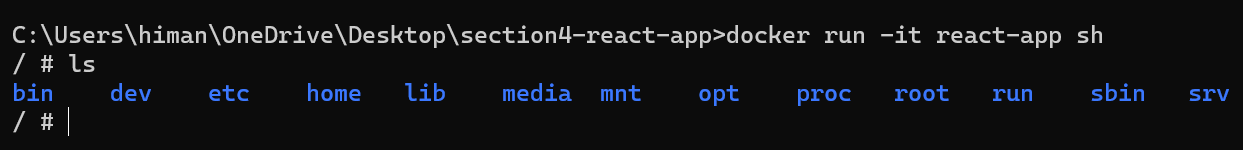


It’s because *our alpine linux does not come with bash*.

But it comes with the basic *shell* (*sh*).

docker run -it react-app sh

Now we are doing a shell session inside this container,



If we type node --version,



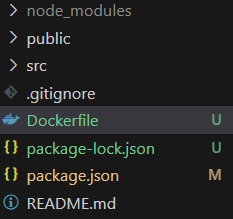
In this image we do not have our application files, we only have alpine linux and node. So in the next lesson we will learn how to copy application files into this image.

**Copying Files and directories**:

Now we have a base image, next step is to copy the application files into the image. For that we have two instructions called *COPY* and *ADD*.

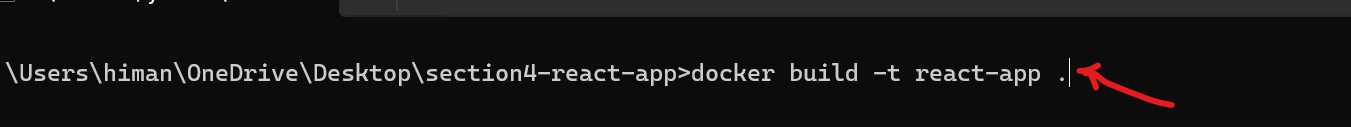
Both commands have almost same syntax but ADD has a couple of additional features.

Let’s start with COPY , *with this we can copy one or more files or directories from the current directory* (*our project directory*) *into the image*.

 🡨 current directory

We cannot copy anything outside this directory.

Here is the reason when we execute the build command, look at the last argument (*period .*)



A period means current directory, so when we execute this command docker client sends the *content of this directory* to the docker engine. This is called *build context*.

**“***Docker client sends the build context to docker engine and then docker engine will start executing Dockerfile instructions one by one, therefore at that point docker engine does not have access to any files or directories outside of the current directory***”**.

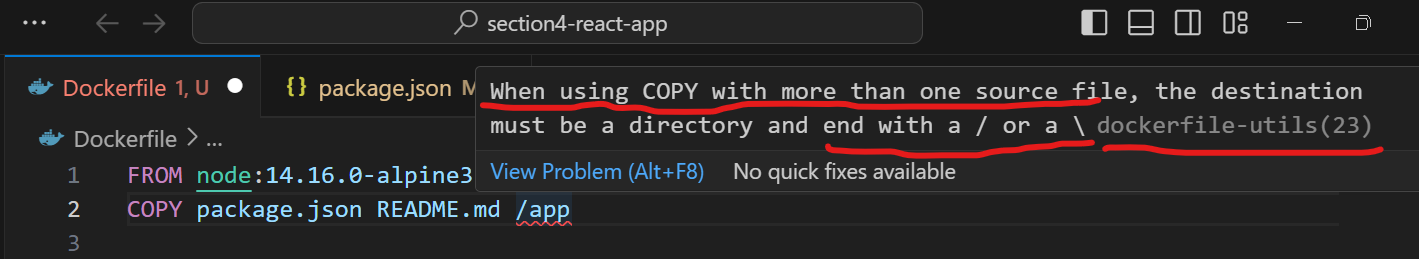
Here we can copy one or more files like package.json into /app into the image.

FROM node:14.16.0-alpine3.13

COPY package.json /app 🡪*Like this*

If /app directory does not exist Docker will automatically create it.

We can also copy more than one file,



So in the end we can add a / like this,

FROM node:14.16.0-alpine3.13

COPY package.json README.md /app/ 🡪 *like this*

We can also use a pattern,

FROM node:14.16.0-alpine3.13

COPY package\*.json README.md /app/ 🡪 *all the file names that start with package*

and end with json

In this directory we have two files that fit this pattern, which is *package.json* and *package-lock.json*.

Note: package.json includes the list of dependencies and versions but the actual versions that might be installed on this machine might be a little bit different than them. So *package-lock.json keeps track of the exact version of these dependencies installed on this machine*.

If we want to copy everything the current directory into the app directory,

FROM node:14.16.0-alpine3.13

COPY . /app/ 🡪 *Like this (it’s an absolute path because it starts with* ***/****)*

This is all about source files and directories. Now let’s talk about *destination*.

We can also set the relative path if we set the WORKDIR instruction (*working directory*) first.

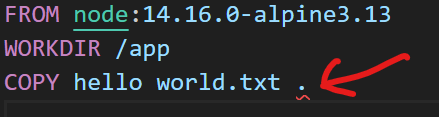
FROM node:14.16.0-alpine3.13

WORKDIR /app 🡪 *like this*

COPY . . 🡪 *second period means current WORKDIR*

All the instructions that come after will be executed inside this working directory.

If *a file name has space in its name and we try to COPY it*,

🡨 we get an error immediately.

This is where we use another form of COPY instruction, means *we can pass an array of strings and each string represents an argument of the COPY instruction*.

FROM node:14.16.0-alpine3.13

WORKDIR /app

COPY ["hello world.txt", "."] 🡪 *like this*

Here have wrapped our *source item* (*hello world.txt*) in double quotes as well as the next argument.

To keep things simple, we will copy everything in our context directory (*current working directory*) into the WORKDIR of the image.

FROM node:14.16.0-alpine3.13

WORKDIR /app

COPY . .

We also have ADD instruction, it has the exact same syntax as COPY but it also has two additional features.

🡪 We can add a file from a URL.

ADD http://.../file.json . 🡪 *if we have access to this URL we can add it*

🡪 We can add a compress (.*zip or .rar*) file and ADD will automatically uncompress this into a directory.

ADD file.zip . 🡪 *here*

We can use either COPY or ADD instruction but the best practice is to use COPY because its more straightforward and there is no magic around it.

*Use ADD only if you need to use any of these additional features like adding a file from a URL or want to uncompress a compressed file*.

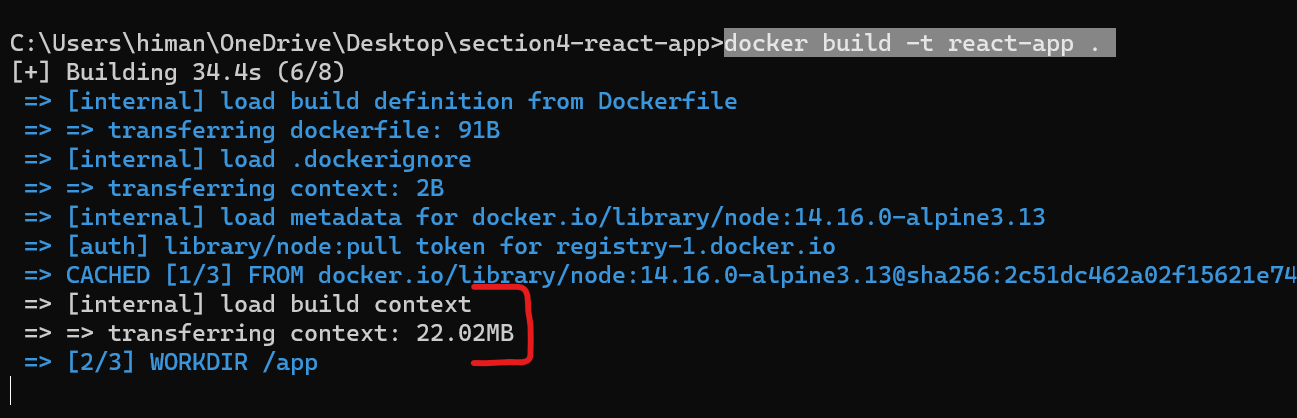
So we will rebuild our image with these instructions for now,

FROM node:14.16.0-alpine3.13

WORKDIR /app

COPY . .

Notice this line,

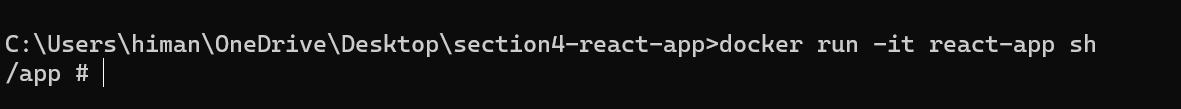


**Transferring context**. All the files and directories of our current directory are being sent to docker engine.

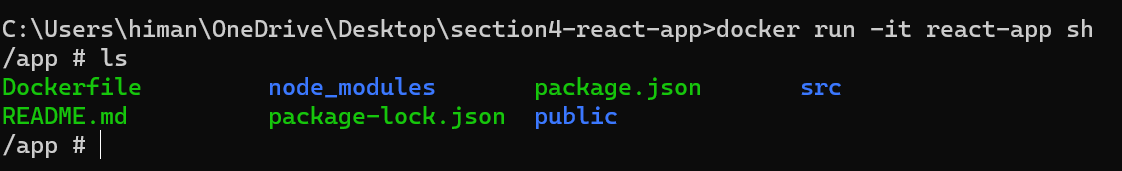
After the image is built, let’s start a container with this image.

docker run -it react-app sh

We start with shell so that we can look at the file system.



We are inside the app directory by default because in our Dockerfile, we set this directory as current working directory.



Here we can see all the files and directories in our project including node\_modules.

**Excluding Files and Directories**:

In the last lesson we saw that *when we build our application, Docker client takes everything in the project directory which we called the build context or the context directory and sent it to docker engine or docker daemon*.

For an application with no features our build context was about 244MB. That’s the result of node\_modules directory.

There is a problem here, later in the course when we talk about deployment we will see that our docker client will talk to a docker engine on a different machine.

So that means *whatever we have in the build context has to be transferred over the network and if we have a large build context with a million files in it, all these files have to be sent to the docker engine on the remote machine*.

We don’t want to transfer these node\_modules directory because all these dependencies are defined in package.json already, so we can simply exclude this directory and copy everything else then restore these dependencies on the target image.

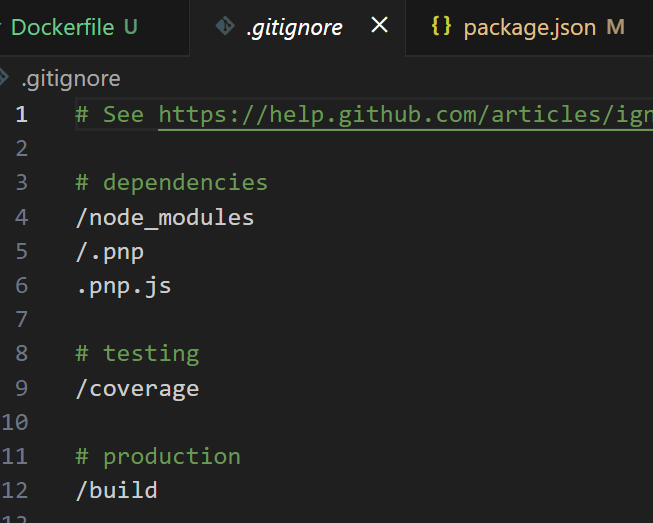
This has two benefits:

🡪 Our *build context is smaller*, so we transfer less data over the network.

🡪 *Build is going to be faster*, so we do not have to wait for all these files to be transferred to docker engine.

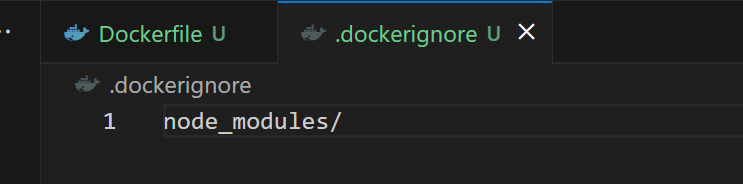
Question is how do we exclude this node\_modules directory?

We know about the concept of .*gitignore* file,



We have similar concept in Docker. So we can create a file in the root of the project called .*dockerignore*.

Here we can list files and directories that need to be excluded.

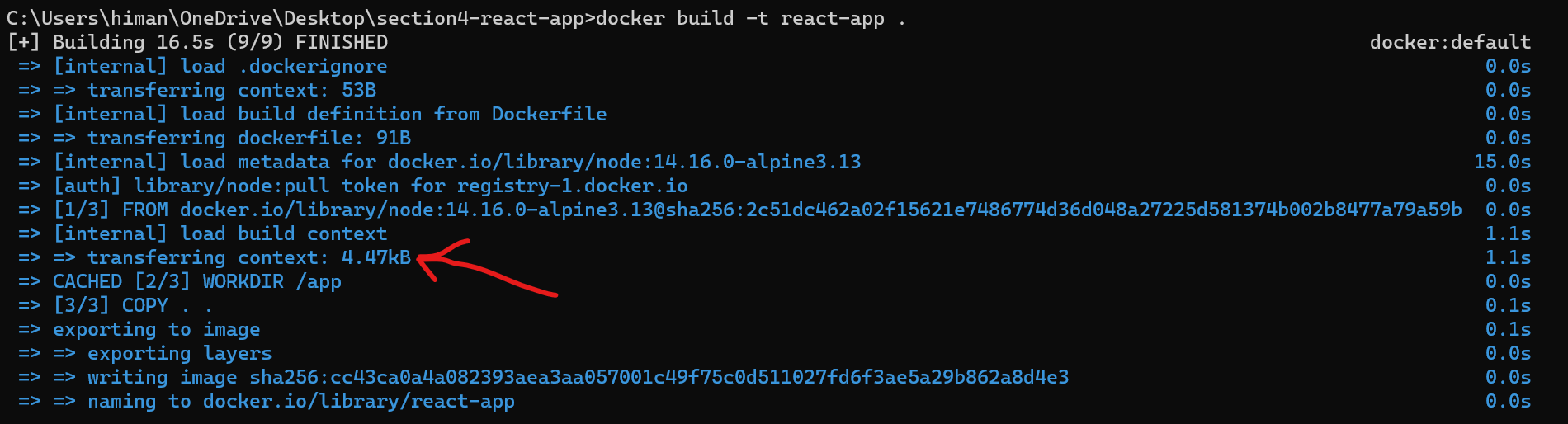


Now when we build a new image, docker will no longer transfer this large, gigantic directory to docker engine.

Let’s run the build command one more time,

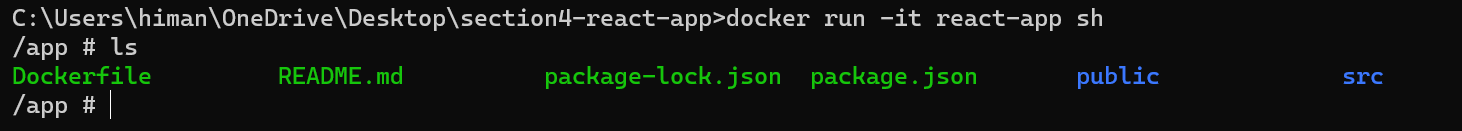
docker build -t react-app .

Now look at our build context, its only 5 kb.



That’s a huge improvement. There is a problem though, if we start a new container and look at the file system of that container, we are not going to see the node modules directory, because we excluded it.

docker run -it react-app sh



We only have two directories here public and src, node\_modules is not here.

And now we have to run npm install inside our /app to install our dependencies.

**Running Commands**:

Next step is to install our project dependencies using npm and this is where we use RUN instruction.

With RUN *we can execute any commands that we normally execute in a terminal session*.

RUN npm install

Here we can run npm install and we can also run any of the linux or windows commands.

For example, we want to run python on this image as well, we can use apt to install python,

RUN apt install python

Note: If we run apt install python, we will get an error because alpine linux does not have apt package manager by default. It has another package manager called ***apk***. So *be aware of these differences depending on type of linux or windows you are using*.

With these updated instructions,

FROM node:14.16.0-alpine3.13

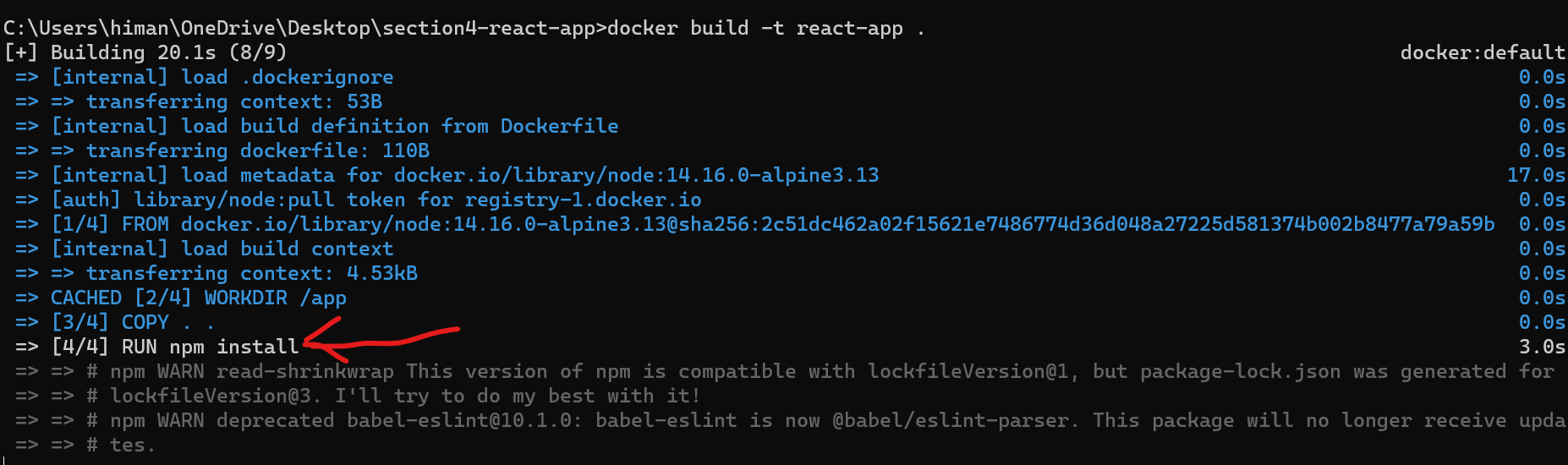
WORKDIR /app

COPY . .

RUN npm install

let’s rebuild the image,

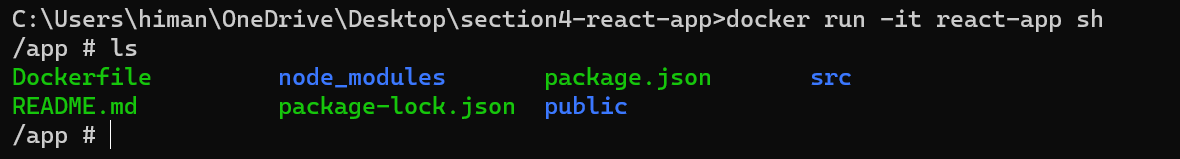
docker build -t react-app .



Docker engine is running npm install to download and install these dependencies.

Once its complete and image is ready, we start a new container with this image and run shell,

docker run -it react-app sh



Here we have node\_modules directory now.

**Setting Environment Variables**:

Sometimes we need to set some environment variables. For example let’s say this frontend application needs to talk to a backend or an API, so *often we set the URL of the API using an environment variable*.

We can use ENV instruction. So we can set API\_URL to <http://api.myapp.com/> or whatever.

FROM node:14.16.0-alpine3.13

WORKDIR /app

COPY . .

RUN npm install

ENV API\_URL = <http://api.myapp.com/> 🡪 *here*

So let’s rebuild this image and see the environment variable in our container.

docker build -t react-app .

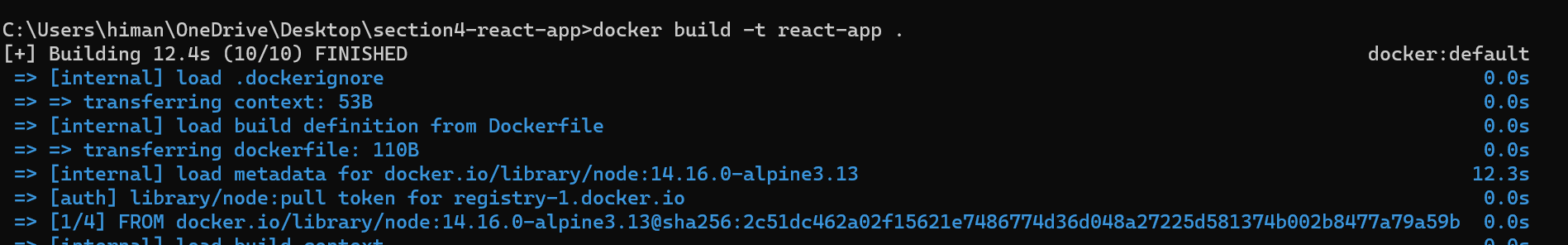
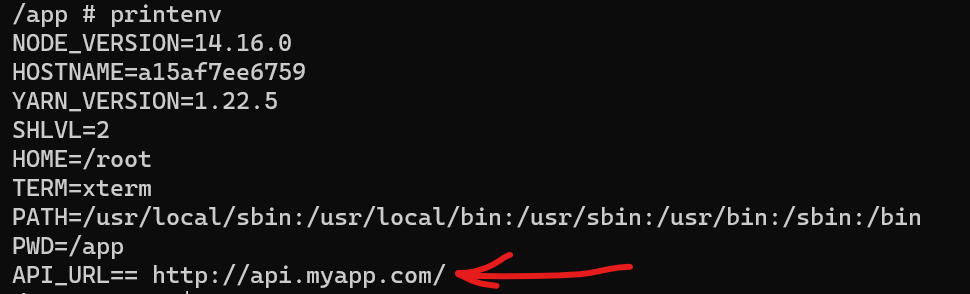


Image is built, now start a container with this image,

docker run -it react-app sh

To view/inspect environment variable:

🡪 printenv command

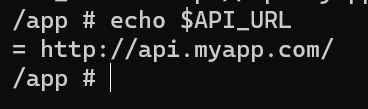


Double == sign are showing in env variable??

*Maybe equal sign for defining value of the env variable is not needed in current versions or* ***there should be no gap between variable and value****! Correct it later with next change…*

🡪 Use echo command,

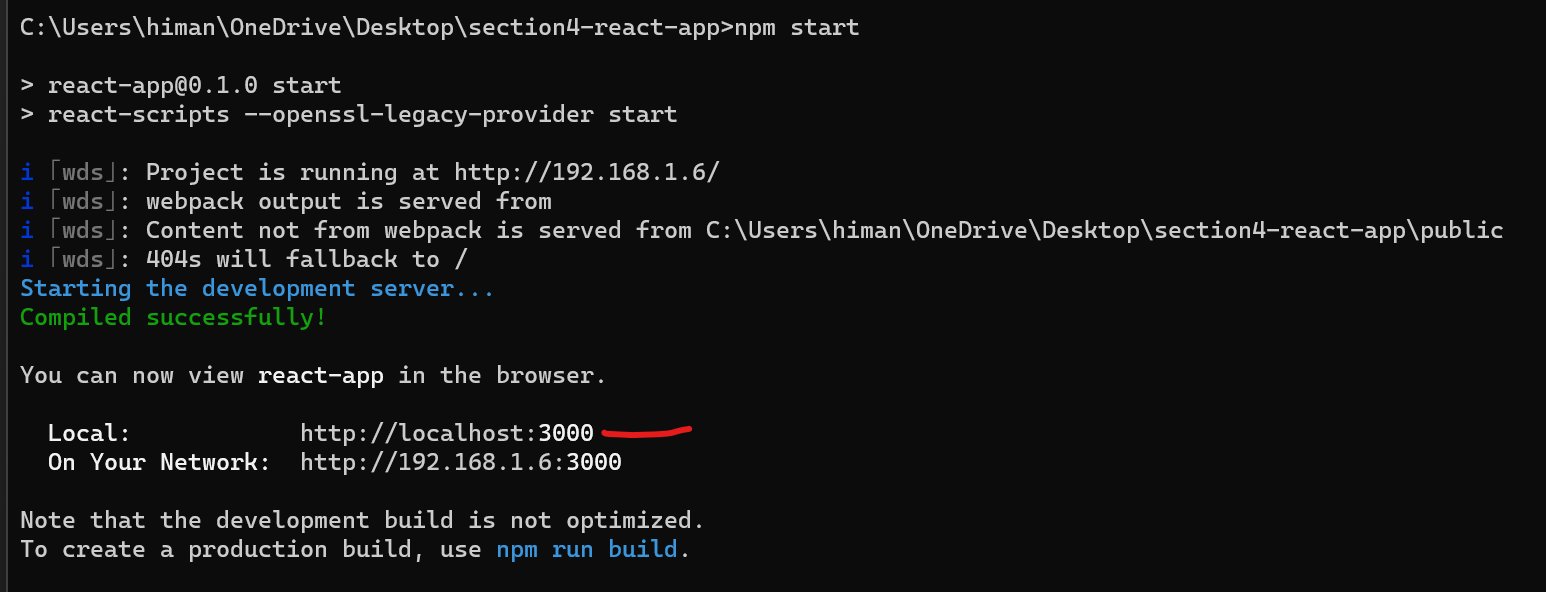
echo $API\_URL



Now whenever we start this container, this environment variable is automatically set for us.

**Exposing Ports**:

Let’s take a quick break from our Dockerfile and start this application outside of docker the way (*run npm start from project directory*).



This starts a development server on port 3000 but in future when we run this application inside a docker container, *this port 3000 will be open inside the container not the host*.

This is something important we need to understand. So *on the same machine we can have multiple containers running the same image and all these containers will be listening to port 3000. But the port 3000 on the host is not going to automatically mapped to these containers*.

In the next section, we will learn how to map a port on the host to a port on these containers.

But for now we will go to our Dockerfile and use EXPOSE instruction to tell what port this container will be listening on, so we write 3000 here.

FROM node:14.16.0-alpine3.13

WORKDIR /app

COPY . .

RUN npm install

ENV API\_URL=http://api.myapp.com/

EXPOSE 3000 🡪 *here*

Note: I just want to clarify something, the EXPOSE command does not automatically publish the port on the host, its just a form of documentation to tell us that this container will eventually listen on port 3000.

So later when we properly run this application inside a docker container, we know that *we should map a port on the host to port 3000 of the container*.

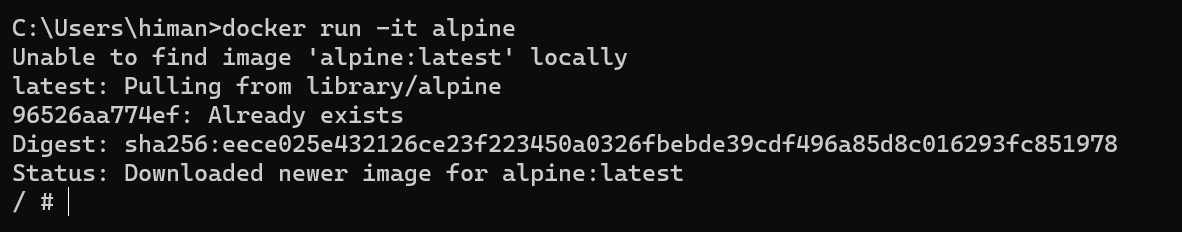
**Setting the User**:

Let’s talk about users, so *by default Docker runs our application with the root user which has the highest privileges which can open up security holes in our system*.

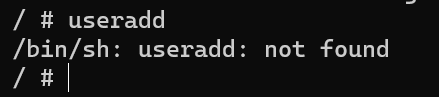
To run our application we should create a regular user with limited privileges.

But before doing that in a Dockerfile, let’s open up a shell session on alpine linux and play with a few commands.

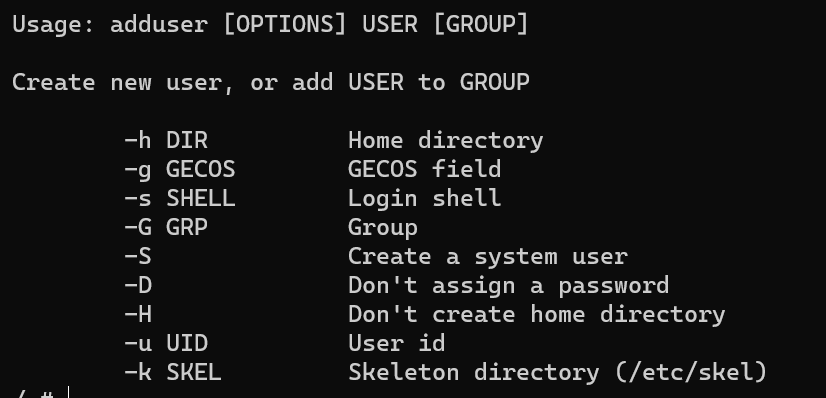
docker run -it alpine



Here we start a shell session, now in this image we do not have useradd command.



Instead we have adduser



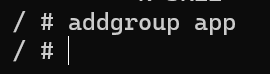
Here we will use two options. First is -G (*for setting primary group of the user*) and the other is -S (*for creating a system user*)

We will *use a system user because this user is not a real user, its just for running an application*.

Before creating this user, we need to create a group so that we can add this user to that group and for that we will use addgroup command.

So we create group called *app*,

addgroup app



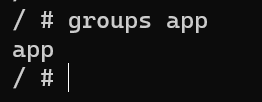
The we use adduser along with *-S* argument to create a system user, *-G* for setting the primary group which is *app* and finally we specify the name of the user as *app*.

adduser -S -G app app

Note: Here we specify the same username as that of group name because it’s a common best practice in linux. *Whenever we create a new user we create a primary group for that user with the same name*.

Now we will verify if this user is in the right group so we use,

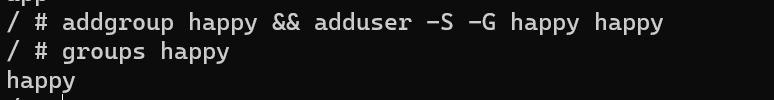
groups app to specify all the groups of the *app* user.



We can combine *addgroup* and *adduser* commands together using && sign and create a user with a primary group simultaneously (*a single command to perform two tasks*).

addgroup happy && adduser -S -G happy happy

For example we create a group called happy with the same username.



This above command is the one that we will run in our Dockerfile

We run this command by using RUN instruction.

FROM node:14.16.0-alpine3.13

WORKDIR /app

COPY . .

RUN npm install

ENV API\_URL=http://api.myapp.com/

EXPOSE 3000

RUN addgroup app && adduser -S -G app app 🡪 *here*

So we will create a group and a user called *app*.

Once we do that we can set the user by using USER command,

FROM node:14.16.0-alpine3.13

WORKDIR /app

COPY . .

RUN npm install

ENV API\_URL=http://api.myapp.com/

EXPOSE 3000

RUN addgroup app && adduser -S -G app app

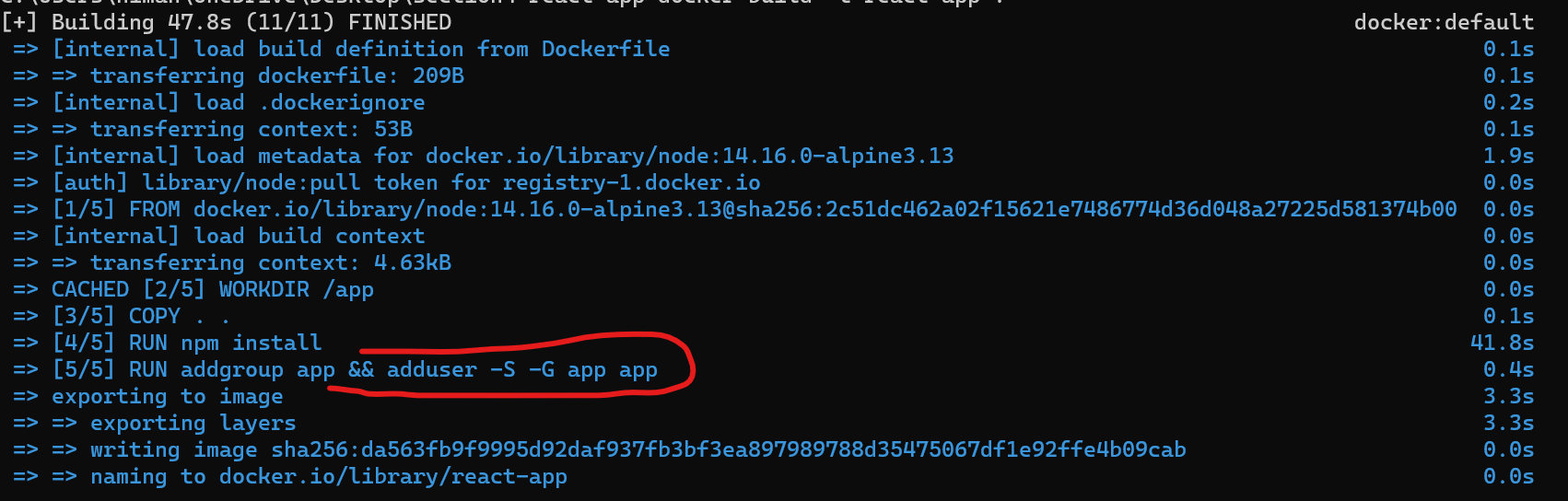
USER app 🡪 *Here*

So all the following commands will be executed by using our newly created *app* user.

Now we save the changes and rebuild our image.

docker build -t react-app .

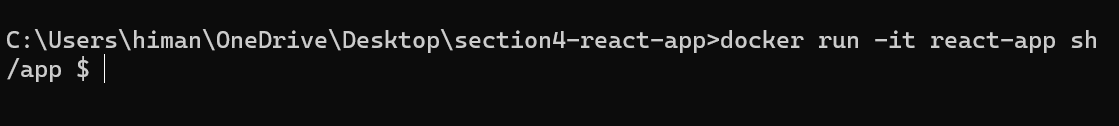
Notice the 5th step,



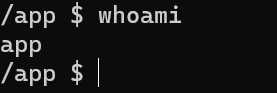
That is our RUN command for creating a new group and a new user.

Now let’s start a new container and make sure the current user is the app user not the root user.

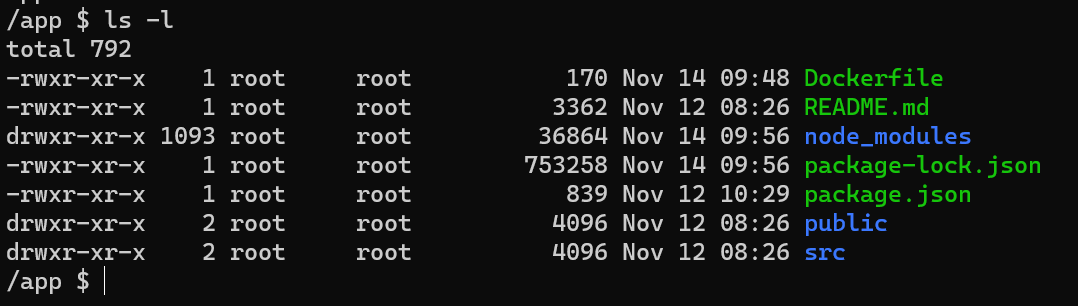
docker run -it react-app sh



To see the current user, we need to use whoami command.



Let’s look at the files and permissions in our app directory using ls -l command,



All these files as we can see are owned by the root user. As we can see root user can read / write any of these files and directories but the current user running this session is the *app* user (*it falls in* ***others*** *group and not able to write any of the files*).

In contrast if we execute this application using root user, a hacker could potentially rewrite something in our application.

**Defining Entrypoints**:

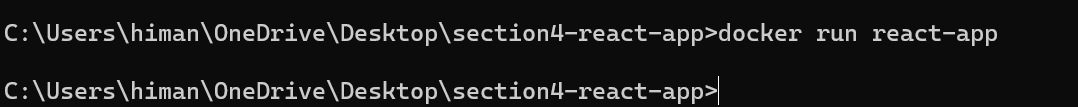
Our Dockerfile is almost ready, now how do we start our application?

As we have seen already inside project directory we can start the application by running npm start. So this is the command we should execute when starting a container.

Note: Here we don’t start the container with interactive mode (using -it) because we do not want to interact with this container, so we do not need to run a shell session. We just want to start he application.

So here after docker run, we just type the image name (*react-app*), let’s see what happens when we type,

docker run react-app

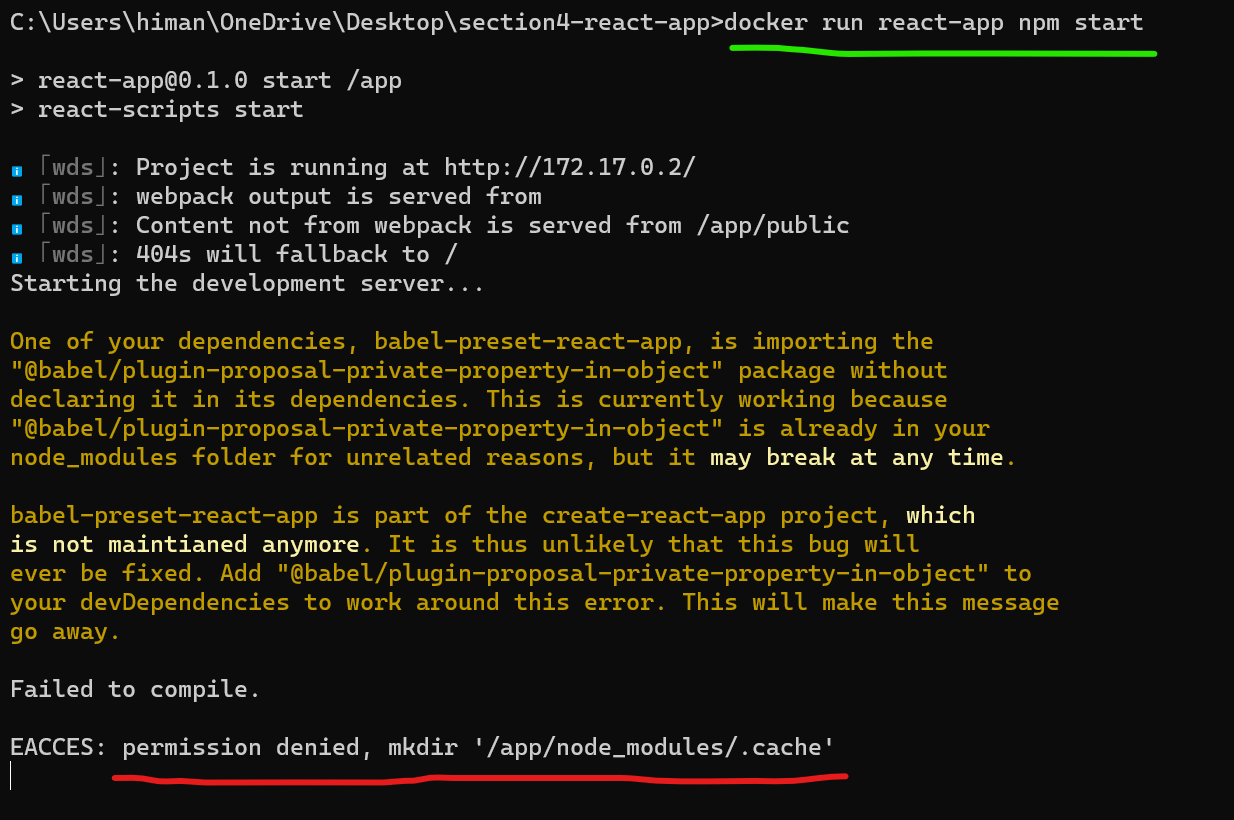


Our container started and then stopped automatically because we did not specify a command or a program to execute. So this is where we are going to type npm start.

docker run react-app npm start

**personal side note**: *needed to remove –openssl-legacy-provider*

"//": "react-scripts --openssl-legacy-provider start",



We get a permission error.

To understand the reason behind it, look at the Dockerfile.

FROM node:14.16.0-alpine3.13

WORKDIR /app

COPY . .

RUN npm install

ENV API\_URL=http://api.myapp.com/

EXPOSE 3000

RUN addgroup app && adduser -S -G app app 🡪 *here*

USER app 🡪 *and here*

In our Dockerfile, We set the user at the end. So we executed all the previous instructions (*WORKDIR to EXPOSE*) as the root user but then we switched to regular user with limited privileges.

So move these two lines on the top,

FROM node:14.16.0-alpine3.13

RUN addgroup app && adduser -S -G app app

USER app

WORKDIR /app

COPY . .

RUN npm install

ENV API\_URL=http://api.myapp.com/

EXPOSE 3000

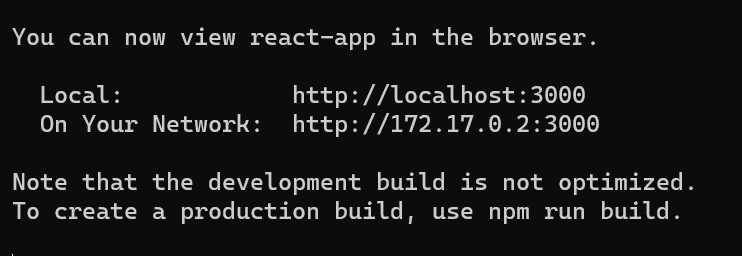
So first we set the user, then we create working directory, copy files and so on.

Now we have to rebuild our image and start a new container.

docker build -t react-app .

And start the new container,

docker run react-app npm start



Our webserver started on port 3000. But as we know this is port 3000 of the container, not localhost (*if we go to localhost:3000 on our machine, we can’t access this app*).

In next section, we will learn how to map a port from the host to a port on the container.

With the current command that we use to start the container, there is a problem. We don’t have to specify npm start every time we run a container.

This is where we use the *CMD* instruction.

So back to the Dockerfile at the end, *we supply a default command to be executed*.

FROM node:14.16.0-alpine3.13

RUN addgroup app && adduser -S -G app app

USER app

WORKDIR /app

COPY . .

RUN npm install

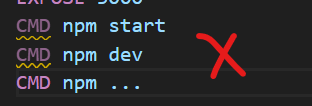
ENV API\_URL=http://api.myapp.com/

EXPOSE 3000

CMD npm start  *🡪 here*

Now we will rebuild the image () then we can start the container by simply running docker run react-app.

Note: Remember, just because the command CMD instruction is for supplying the default command, it doesn’t make sense to have multiple command instructions in a Dockerfile.



If you have multiple command instructions, only the last one will take effect. Beware of that.

Note: With both *RUN* and *CMD* instructions we can execute commands but there is a difference between them. RUN instruction is a ***build time instruction*** (*executed at the time of building the image like installing npm dependencies*) while in contrast the CMD instruction is a ***runtime instruction*** (*executed when starting the container*).

Furthermore CMD instruction has two forms ***Shell form*** and ***execute form*** (*takes an array of strings*).

#Shell form

CMD npm start

#Execute form

CMD ["npm", "start"]

If we shell form, Docker will run this command in a separate shell which is why its called a shell form. On linux this shell is */bin/sh* (*original shell program*), on windows its *cmd* (*command prompt*).

Common practice is to use the *execute form* because with this *we can execute the command directly and there is no need to spin up another shell process and also it makes it easier & faster to clean up resources when we stop containers.*

We have another instruction called *ENTRYPOINT* which is very similar to CMD and it also has shell & execute form.

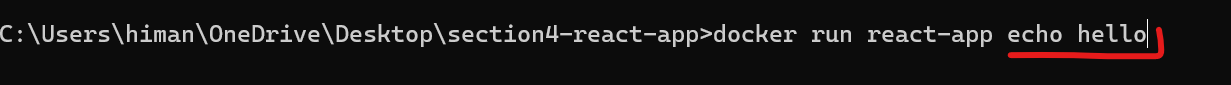
#shell form

ENTRYPOINT npm start

#execute form

ENTRYPOINT [ "npm", "start" ]

Note: The difference between CMD and ENTRYPOINT instructions is that comparatively when we use CMD it is easier to override CMD instructions when starting a container. For example,



Here we have supplied an echo command which will override CMD written in Dockerfile.

In contrast, when we use ENTRYPOINT, it can be only overridden if we supply --*entrypoint* option in the terminal.



In practical terms, we often want to use ENTRYPOINT when we know for sure that this is the command or program that should be executed whenever we start a container but *with CMD there is a flexibility in usage and we can always override this useful in ad hoc situations*.

**Speeding up the builds**:

It is noticeable that our builds are fairly slow and every time we make a small change, we have to rebuild and wait almost 1-2 minutes.

So let’s see how can we optimize our builds.

First thing we need to understand is the *concept of layers in docker*.



*A docker image is basically a collection of layers. We can think of a layer as a small file system which only includes modified files*.

So when Docker tries to build an image for us, it executes each of these instructions and create a new layer. That layer only includes the files that were modified as a result of that instruction.

From the first instruction,

FROM node:14.16.0-alpine3.13 🡪 *node and linux files layer*

Docker takes the node image and put it in the layer (*more accurately node image itself has several layers as well*). For now let’s just imagine we have a single layer which consist of all the node and Linux files.

Then Docker will execute second instruction,

RUN addgroup app && adduser -S -G app app 🡪 *user & group file layer*

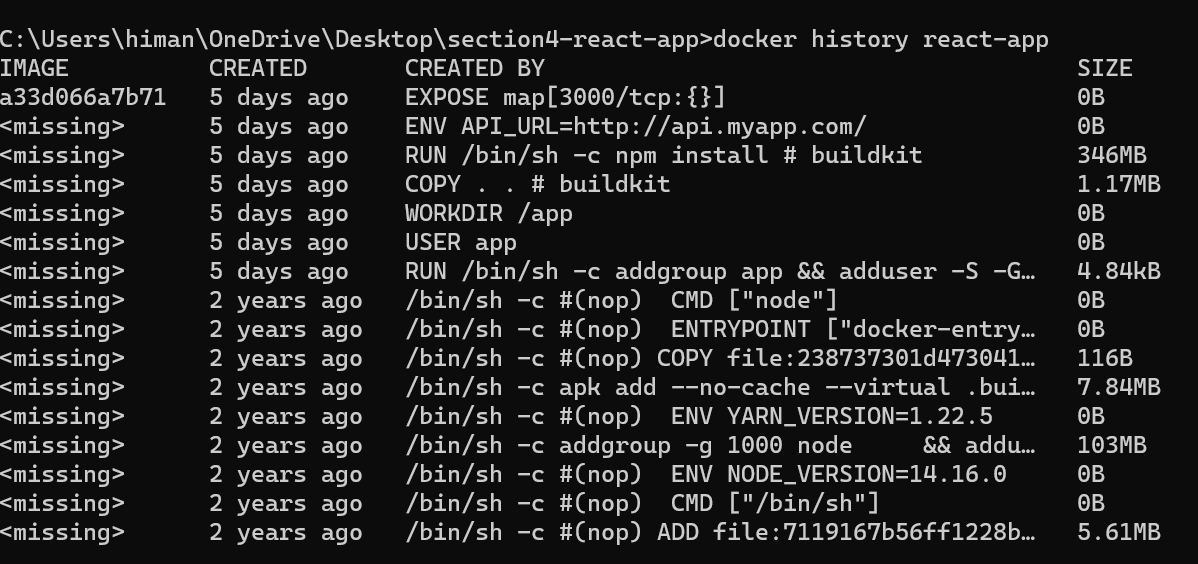
To add a group and a user. Once again it will create a new layer because as we saw in previous section when we add a new user in a group, something is written in file system (*couple of files are modified*). So this layer will only include modified files.

Similarly docker will execute rest of the instructions and create several layers. Let’s have a look at these layers of our image.

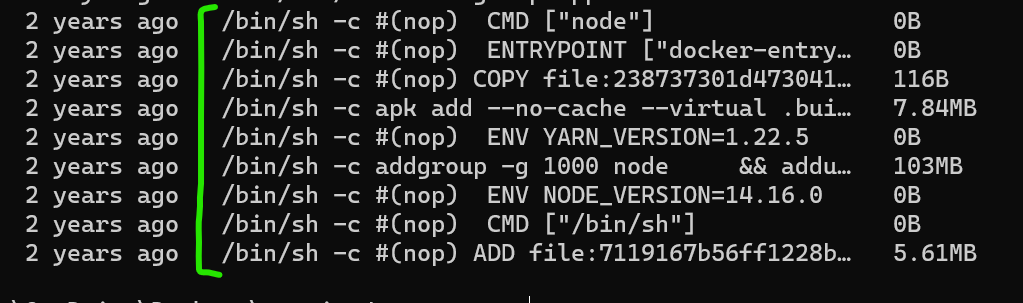
docker history react-app

For now we will just look at *CREATED\_BY* and *SIZE* columns.

*In Created By we can see the instruction which created a new layer and in size column we can see size of that layer*.



We have to look this table from bottom – top.

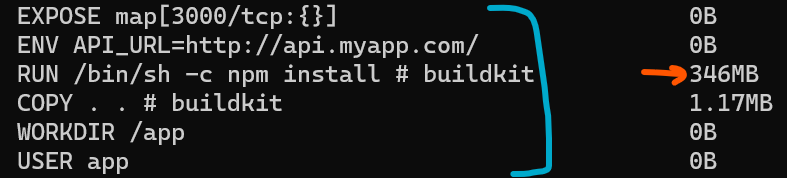


In the bottom we have layers that come from node and linux. These layers are from FROM instruction which brings in several layers of node and linux and then we build on top of those layers.



Then we have RUN instruction for adding a new user. Size is around 4kb because only a couple of files were modified as a result of this command.

Notice this command is executed inside */bin/sh* since we used this instruction in the shell form (*if we used* ***execute*** *form, this command would be executed directly*).



After that other instructions are executed. Including npm install which has a size of 346MB.

*So an image is basically a collection of these layers*.

Now Docker has an optimization mechanism built into it. So next time we ask docker to build an image, it’s going to look at the first instruction and see if the instruction is changed or not. If it’s not changed, it’s not going to rebuild this layer instead it will be reused from the cache.

After that docker will look at the second instruction,

RUN addgroup app && adduser -S -G app app

And once again if nothing is changed Docker will reuse this layer from the cache. In contrast if change something here…

RUN addgroup app2 && adduser -S -G app app

🡪 *we added a new group*

Then docker will rebuild this layer.

Look at this instruction,

COPY . .

This is a special instruction because Docker cannot tell if something is changed or not, so it has to look at the content of files as well which means if we make a tiny change in our application docker can’t use this layer from cache and has to rebuild it.

This is where problem happens, once this layer is rebuilt. All the following layers have to be rebuilt as well.

As a result Docker can’t reuse *RUN npm install* layer from its cache and has to install all npm dependencies and we have to wait a full minute to install all the dependencies.

So *for optimization we have to separate installation of third party dependencies from copying our application files*.

Instead of copying all the files in one go, first we need to copy the files needed for installing third party dependencies which are *package.json* and *package-lock.json*. These are the only two files we need for running npm install.

FROM node:14.16.0-alpine3.13

RUN addgroup app && adduser -S -G app app

USER app

WORKDIR /app

COPY package\*.json . 🡪 *First copy both .json files*

RUN npm install 🡪 *Install npm dependencies*

COPY . . 🡪 *Then copy rest of the application files*

RUN npm install

ENV API\_URL=http://api.myapp.com/

EXPOSE 3000

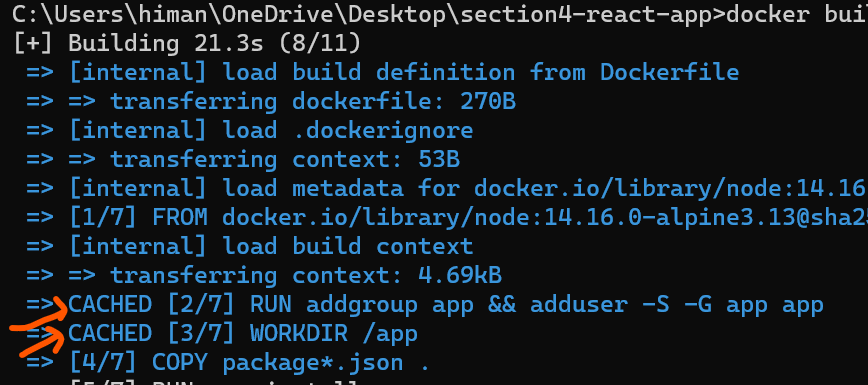
CMD ["npm", "start"]

With this new setup if we have not changed any of the application dependencies docker is going to reuse this layer from its cache because package.json file is not modified.

Similarly Docker is not going to re-install all the npm dependencies (*RUN npm install*)because this instruction is not changed, but if we RUN npm update then this layer will be rebuilt.

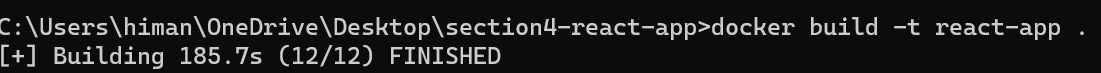
Let’s rebuild the image,

docker build -t react-app .



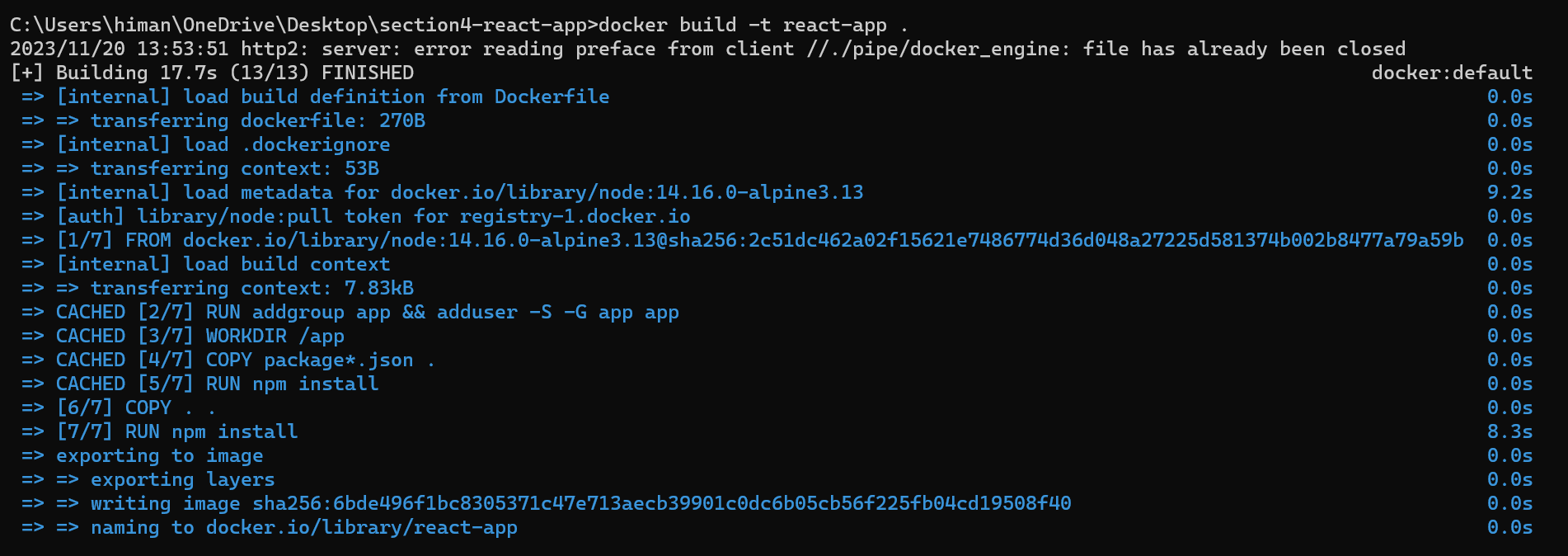
Notice CACHED because docker is reusing the layer cache for these instructions.

This time it took 185 seconds,

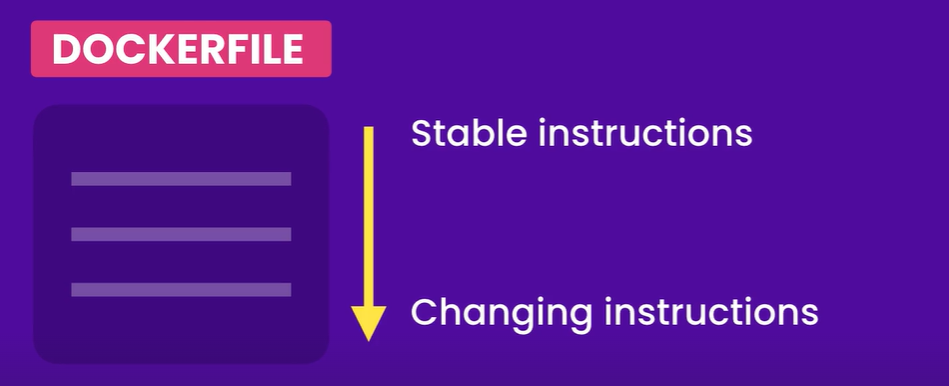


Because it’s the first time we are using new configurations.

Now let’s change something in our README.md file so that we can see how fast our build will be created due to optimization.



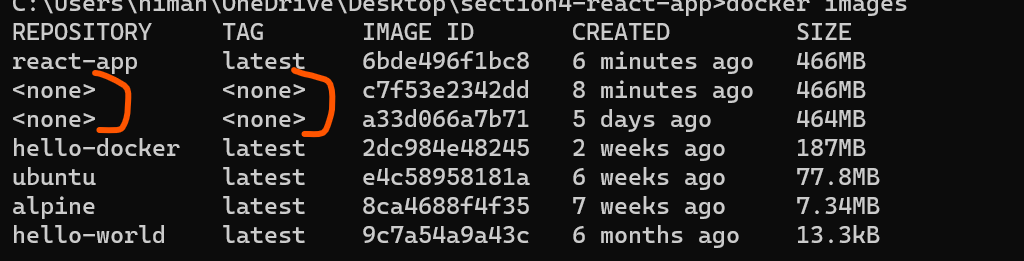
This time it took only 17 seconds to create the build.



***“****In order to optimize our build we should organize our Dockerfile such that instructions that don’t change too frequently should be on top and instructions for files that change frequently should be down at the bottom****”***.

**Removing images**:

If we look at the images by using docker images command, we can see some images have no name and no tag.

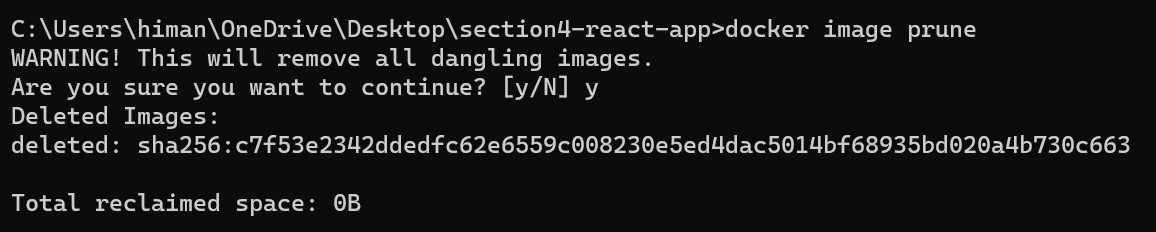


These are what we call *dangling images* or loose images.

These are basically layers that have no relationship with a tagged image. *So as we are changing our docker file and rebuilding our image, docker was creating these layers and at some point these layers lost their relationship with our react-app image*.

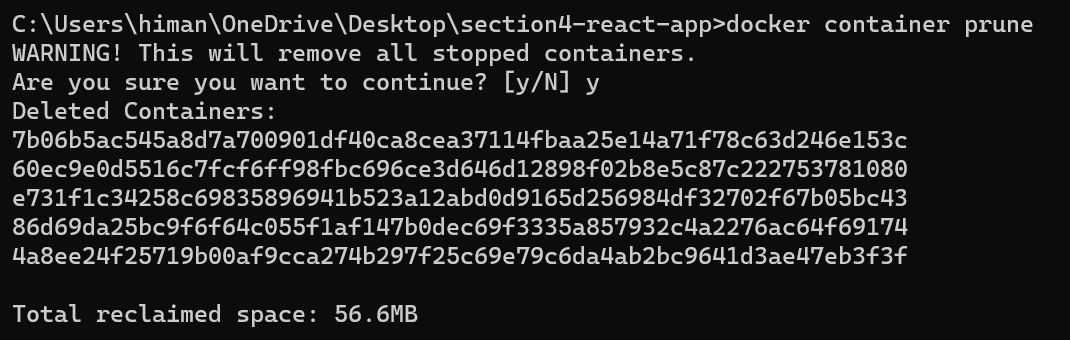
So as we work with docker we see these dangling images popping up all the time. To get rid of them we have to use *prune* command.

docker image prune

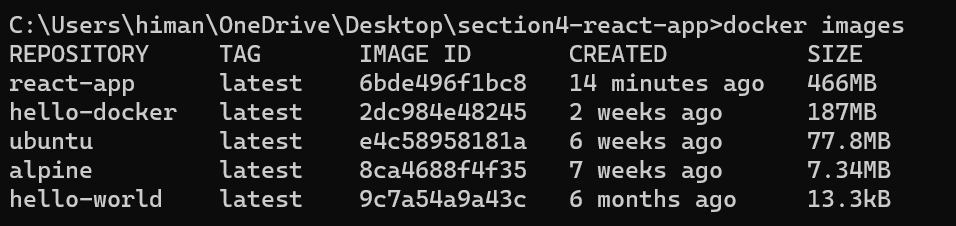


Sometimes there may be some stopped containers that might be using our images we can use prune command for removing dangling containers as well.

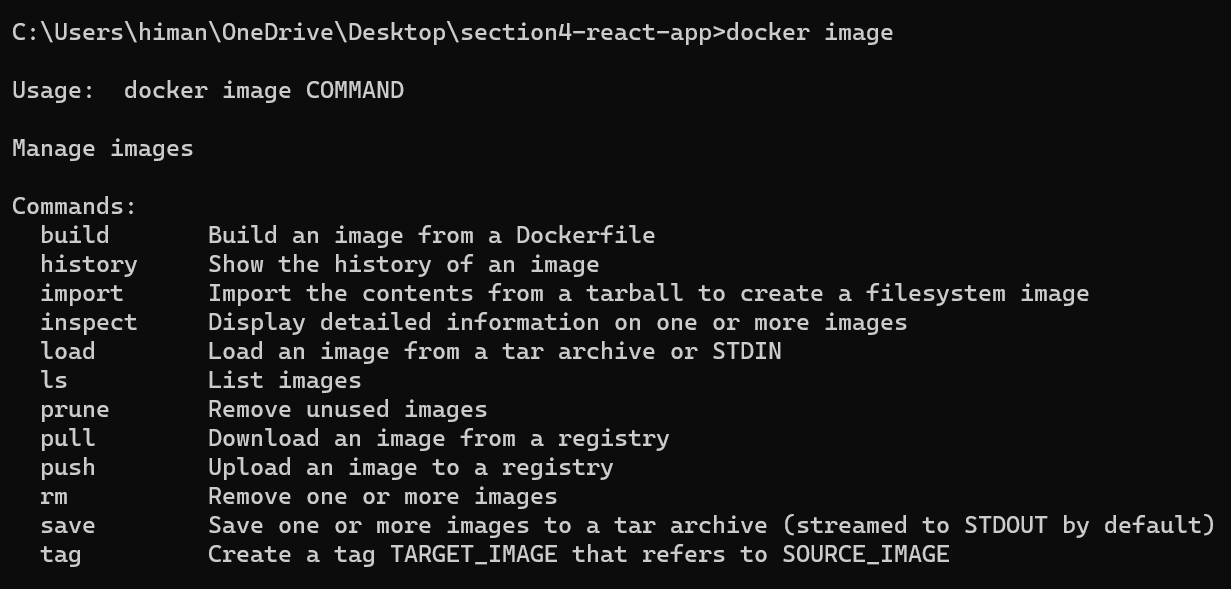
docker container prune



Now we only see proper images we have created or pulled from docker hub.

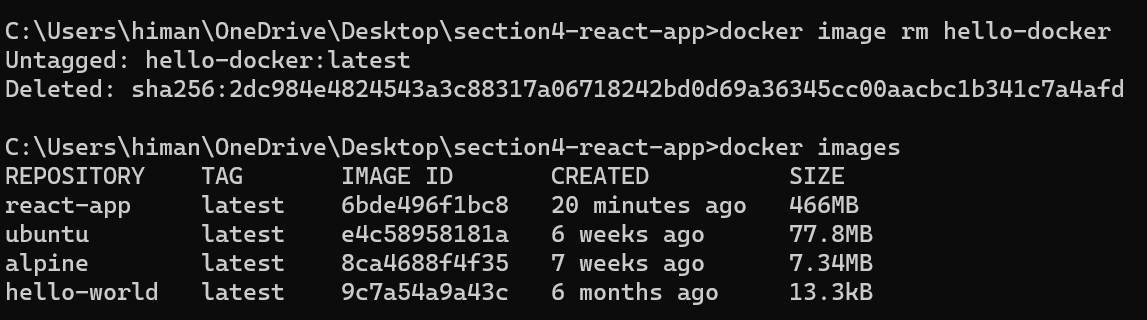


Note: All the image related operations begin from docker image.



Let’s delete *hello-docker* image using rm command.

We can reference an image by its name or image - id.



**Tagging images**:

Whenever we build a image or pull it from docker hub, by default docker uses the latest tag. Now this latest tag is just a label and there is nothing special about it (*so it does not mean that this is the latest version of your image*).

In production we should always use explicit tags to identify which version we are currently running in each environment (*test / staging / production*).

Let’s create a clean build first,

docker build -t react-app .

If we look at docker images,



We have 4 images and all of them have the latest tag.

There are two ways to tag an image.

🡪 ***Tag an image while building it***

docker build -t react-app**:1** .

The command is almost same as before but there is just one simple variation. After image name we have a colon followed by version number (*it can be plain number like 1, 2, 3 or 1.2, 3.1.2 etc*).

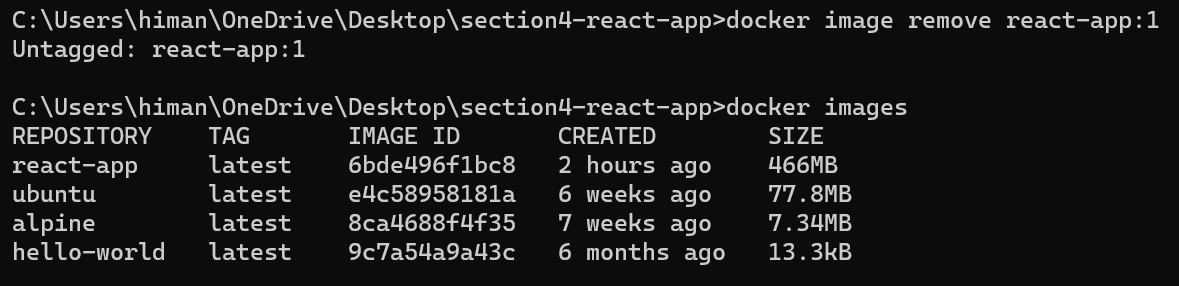
After running this command, we see images one more time,



Here we have an interesting situation, the react-app image we have, has two tags 1 and latest. Same image with multiple tags (*look at the image id*). So a image can have multiple tags.

What if we made a mistake and want to remove a particular tag. We use remove command.

docker image remove react-app:1

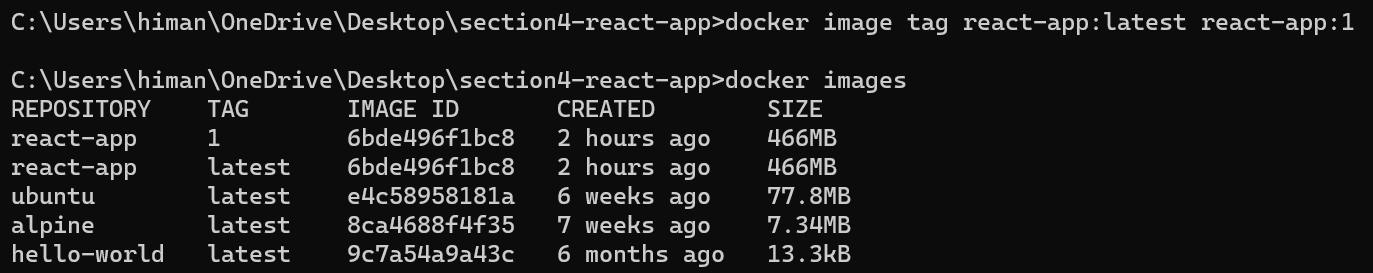


🡪 ***Tag an image after building it***

We use the tag command.

docker image tag react-app:latest react-app:1

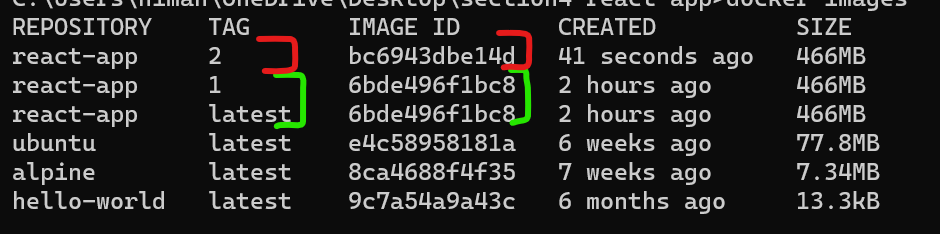
Here we give react-app a new tag.



Let’s make a couple of changes in README file and build a new image with tag #2.

docker build -t react-app:2 .

Now let’s look at the images,

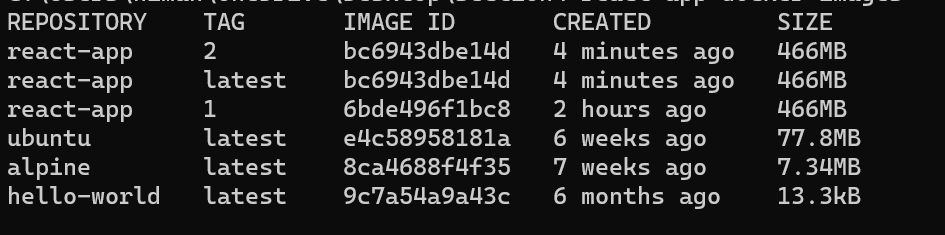


Notice the latest tag of the react-app is pointing to the tag #1 because both these tags have the same identifier (*older image*).

Here is the takeaway, the latest tag does not necessarily reference the latest image, we have to explicitly apply to the latest image.

So we can make our tag#2 image as our latest image,

docker image tag bc6 react-app:latest

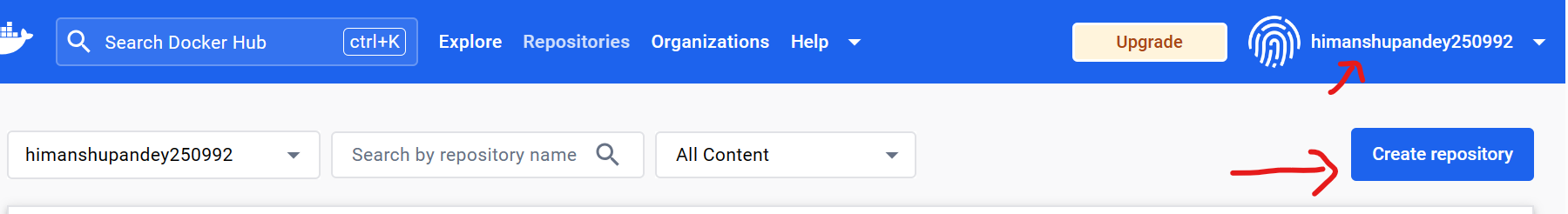


So now the latest tag is pointing to tag#2 of our image.

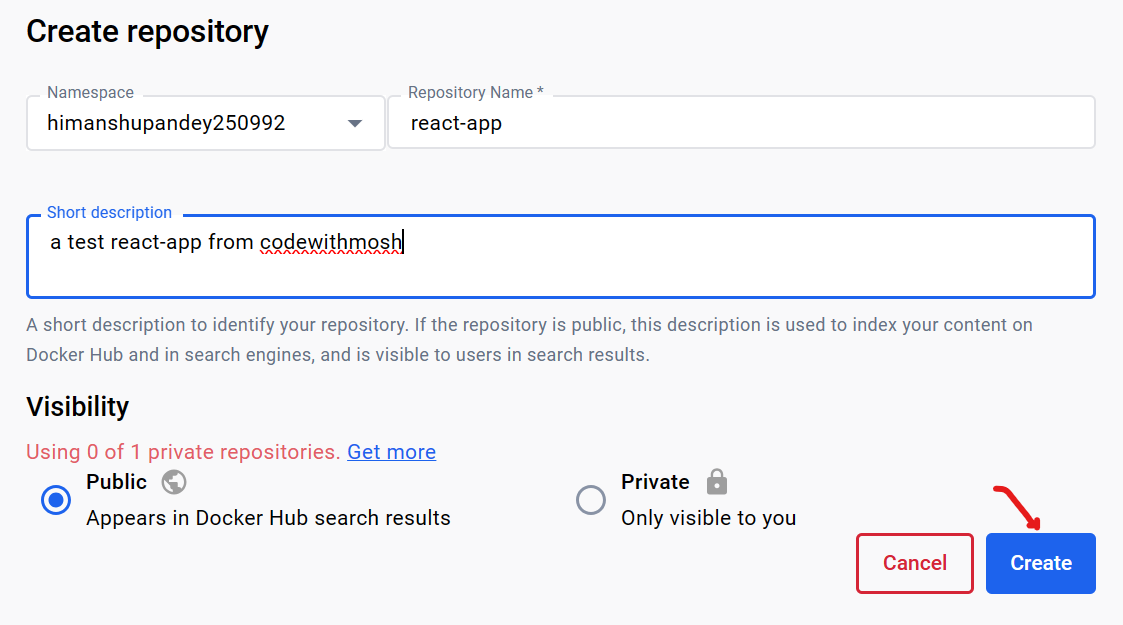
**Sharing images**:

Let us see how we can share our images with others.

Head over to hub.docker.com and login with docker account.

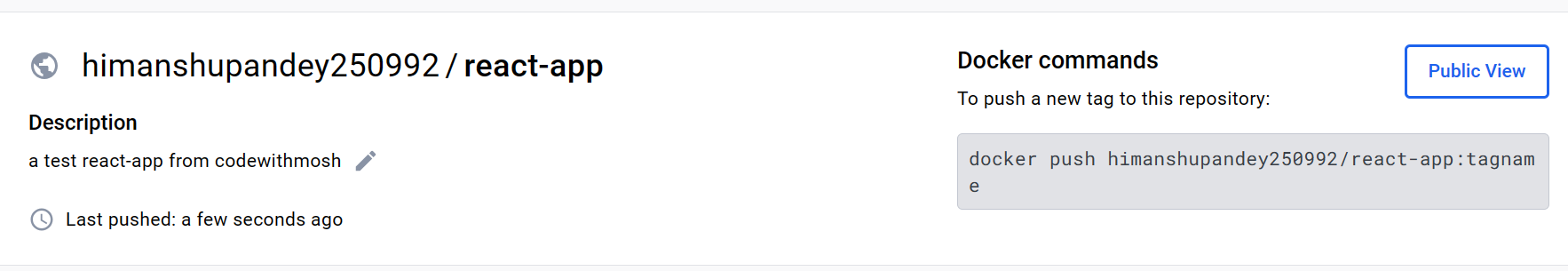


Go to create repository which is similar to creating a GitHub repository. Here we can have multiple images with different tags.



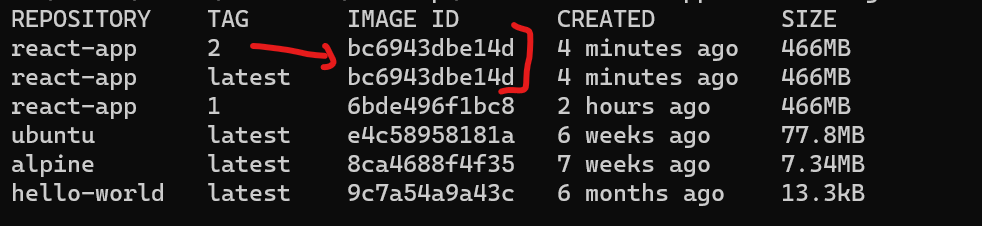
Note: Alternatively we can connect our account with GitHub. So every time we do a push, docker hub automatically pulls the latest code and builds a new image.

After creating we have a repository,



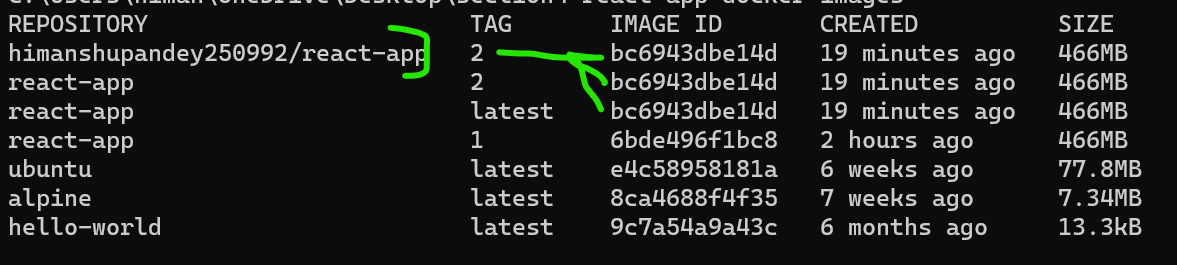
Now to push our image to this repository, we have to give it this tag himanshupandey250992/react-app (*username/name of repo*)

Let’s say we want to publish latest version of our image,



docker image tag bc6 himanshupandey250992/react-app:2

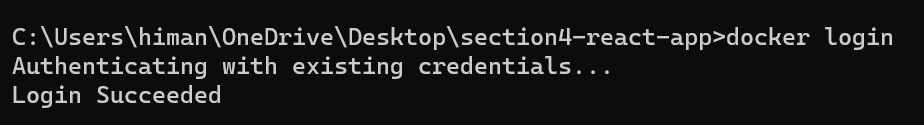
Now our images with id bc6 has three tags,



And we are ready to push this image,

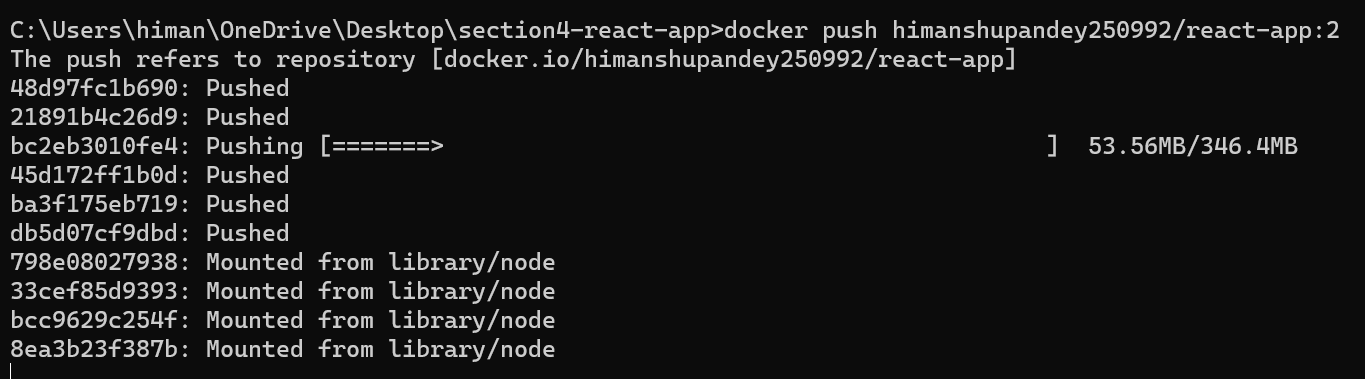
So first we have to login,

docker login



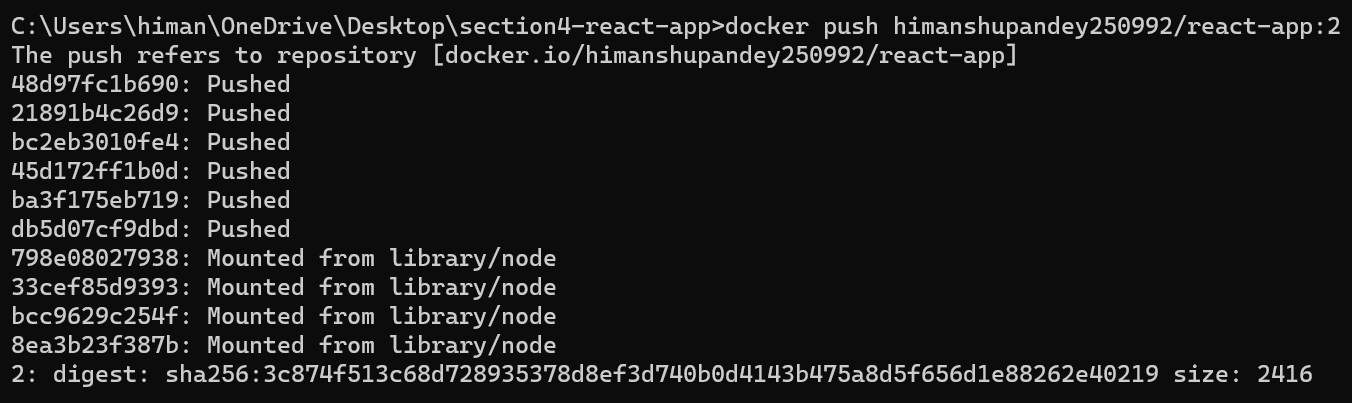
Now we will push the image,

docker push himanshupandey250992/react-app:2

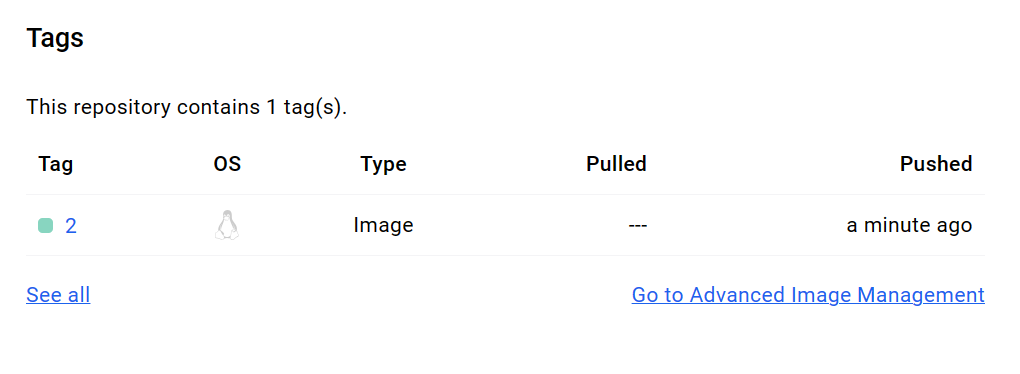


Now docker is pushing each layer of our image. So first time we push this image, it’s going to take a little while because one of our layer which include all npm dependencies is fairly large.

So once we push this image, our future pushes will be faster, assuming we have not changed our application dependencies.



Our image is pushed, so let’s refresh our repo page.

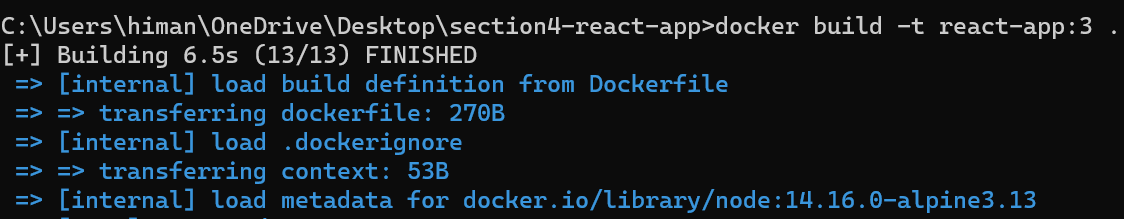


So in this repository we have one tag #2 which is based on linux.

Back to the project, let’s make a small change to README file and do another push.

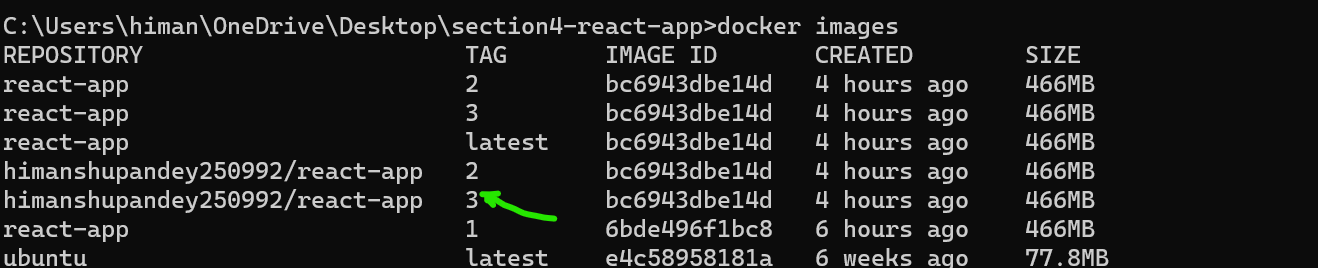
First build the image,

docker build -t react-app:3 .



Now we need to give this build an extra tag that starts with himanshupandey250992 (*username*) / react-app (*repo name*)

docker image tag react-app:3 himanshupandey250992/react-app:3



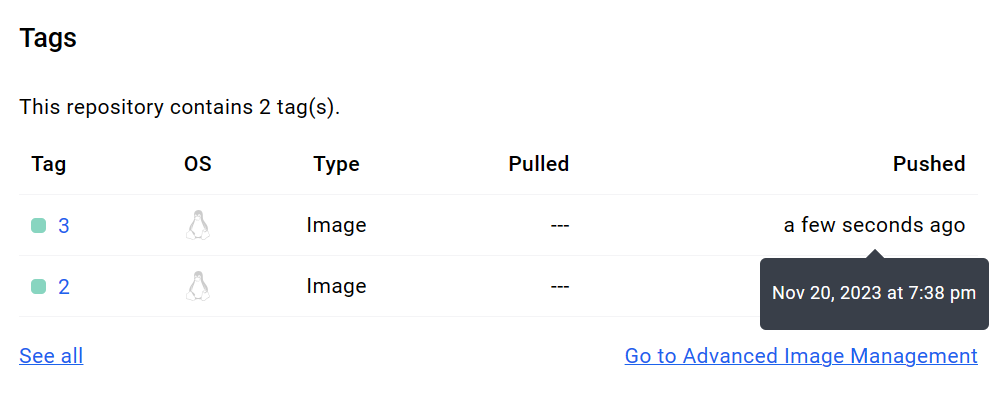
Let’s push this image now,

docker push himanshupandey250992/react-app:3

Look some of these layers already exist, so this push is lot faster than previous push.



Now in this repository we have two tags and since we have these images on docker hub, we can pull these on any machine that runs docker.



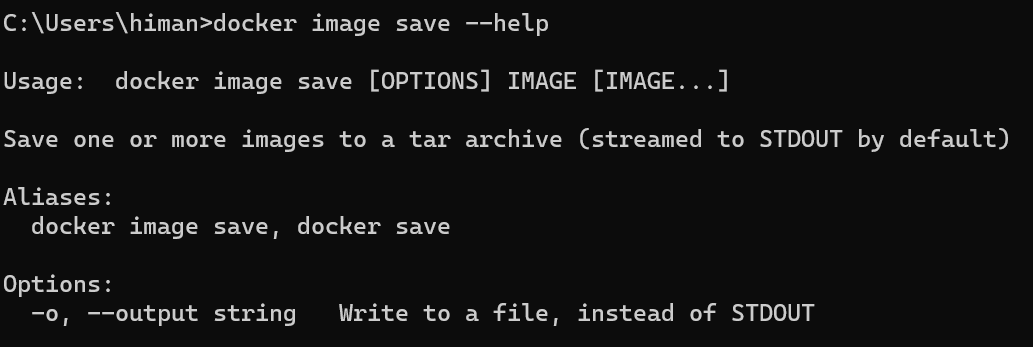
**Saving and Loading Images**:

Let’s say we have an image on this machine and we want to put it another machine but without going through docker hub.

In this case we can *save* this image as a compressed file and load it on the other machine.

Note: If you want to learn about any command then use --help after the end of the command. For example to know about *save* command.

docker image save --help



So here is the syntax,

**Usage**: *docker image save [OPTIONS] IMAGE [IMAGE...]*

After save we have to use one of the options like -o which is the short form for output.

**Options**:

-o, --output string Write to a file, instead of STDOUT

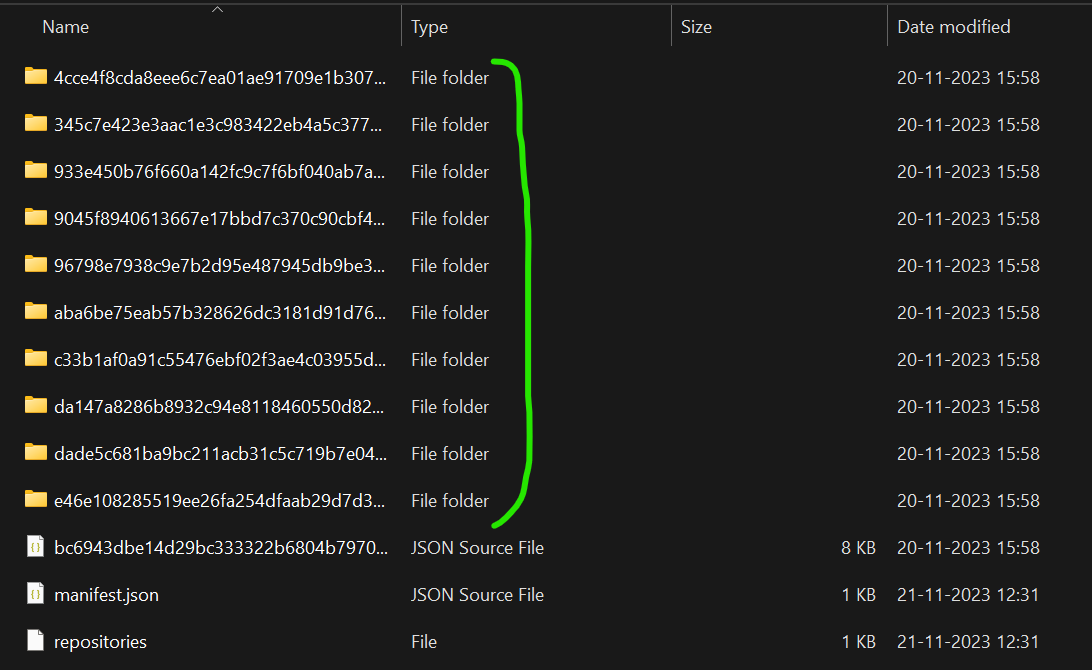
Here in options we can specify a file (*string*)that we need to write to then we identify the image (*IMAGE [IMAGE…]*).

So following this syntax we can write a command,

docker image save -o react-app.tar react-app:3

Here we write a file to save in the current directory called react-app.tar (*a tar file in linux is like a zip file in windows*) and then we specify our image that we want to compress (*react-app:3*).

It takes some time to create the .tar file. If we open this file we can see some folders inside this.



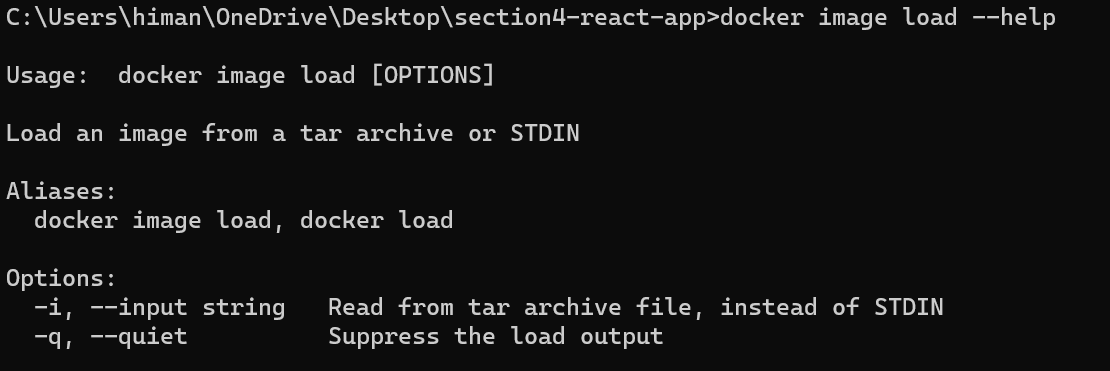
Each of these folders represent a layer. If we look inside these layers/folders we can see a json file, a tar file (*contains all files of that layer*), if we open it we can see code or files inside that layer.



Similarly we can find a layer for linux, layer for node and so on.

So we have *save* command for saving an image into a file and we also have *load* command. Let’s look at its syntax.

docker image load --help



**Usage**: *docker image load [OPTIONS]*

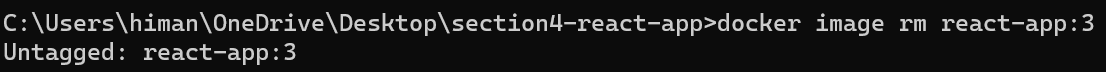
So we use docker image load followed by options.

We can -i or --input to specify the file name and we can use -q or --quiet to suppress the load output.

To demonstrate we will first delete the image from our machine and then use load command to load image file.

To remove,

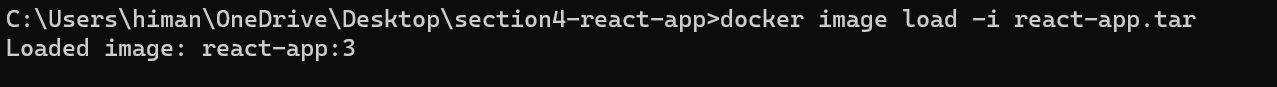
docker image rm react-app:3



This react-app version 3 image is removed from our system.

Let’s load it from the .tar file we just created,

docker image load -i react-app.tar



Our image is loaded now.