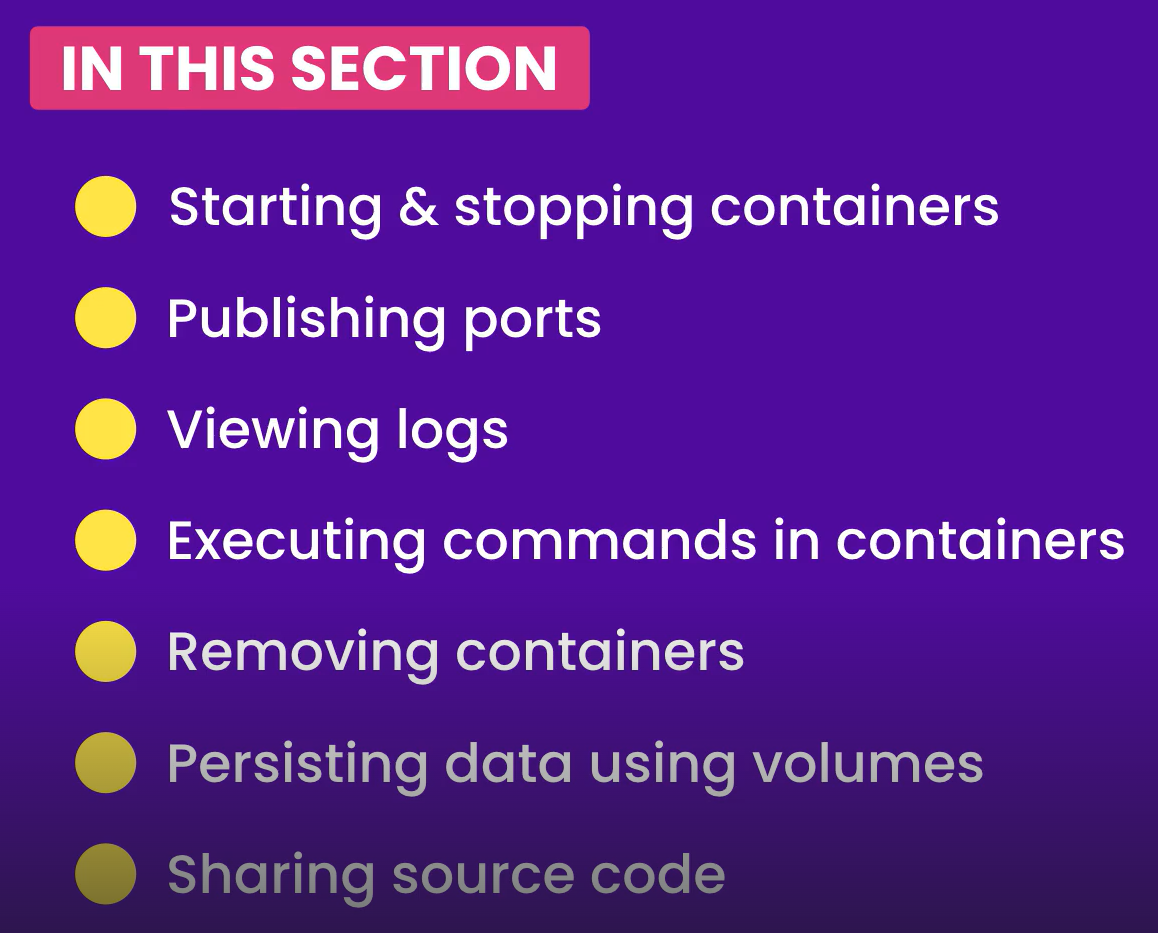


**Introduction**:

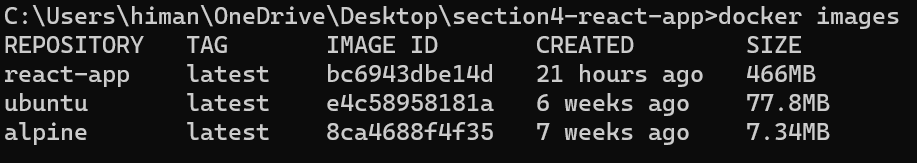
In this section we will learn about containers in more detail…



**Starting Containers**:

We briefly talked about container commands throughout the course.

And we have cleaned up unused images.



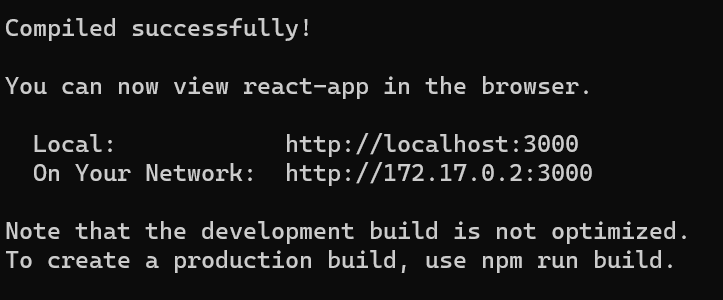
To look at the running containers,

docker ps



Let’s run a new container using our react-app image,

docker run react-app

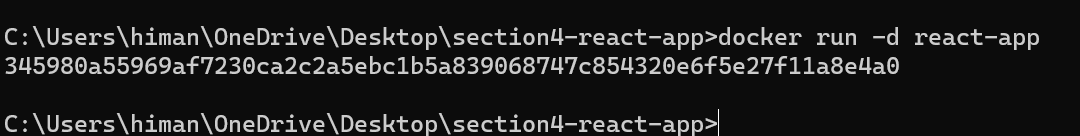


And it started our development web server.

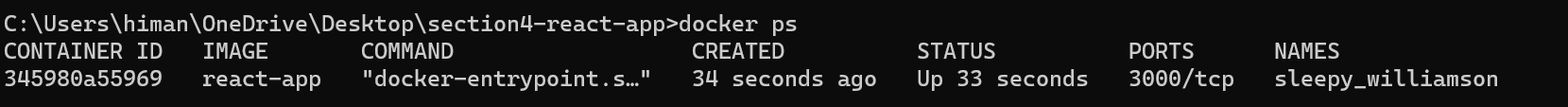
But there is a problem, we cannot type any additional commands on terminal window. If we press ctrl + C to get out of this our container stops.

Let us see a cool technique, we will use the *run* command again but with a new option -*d* (*short for detached*). This way we can *run the container in detached mode or in the background*.

docker run -d react-app



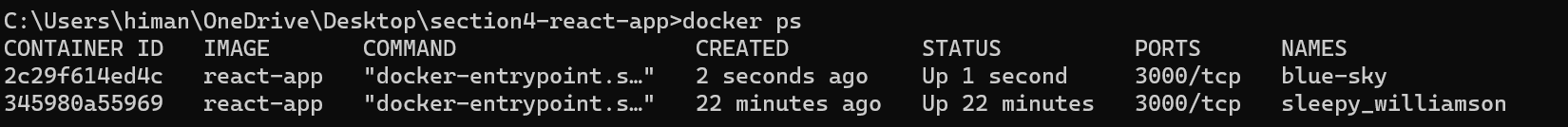
Our container / process is still running,



Note: If we look at the far right column, we can see a NAMES column. Docker automatically associates each container with a random name, so in the future when we want to reference a container, we can either use its ID or NAME *but we can also give our container a name when starting them*.

Let’s start another container in detached mode, this time we will use --name to give it a name.

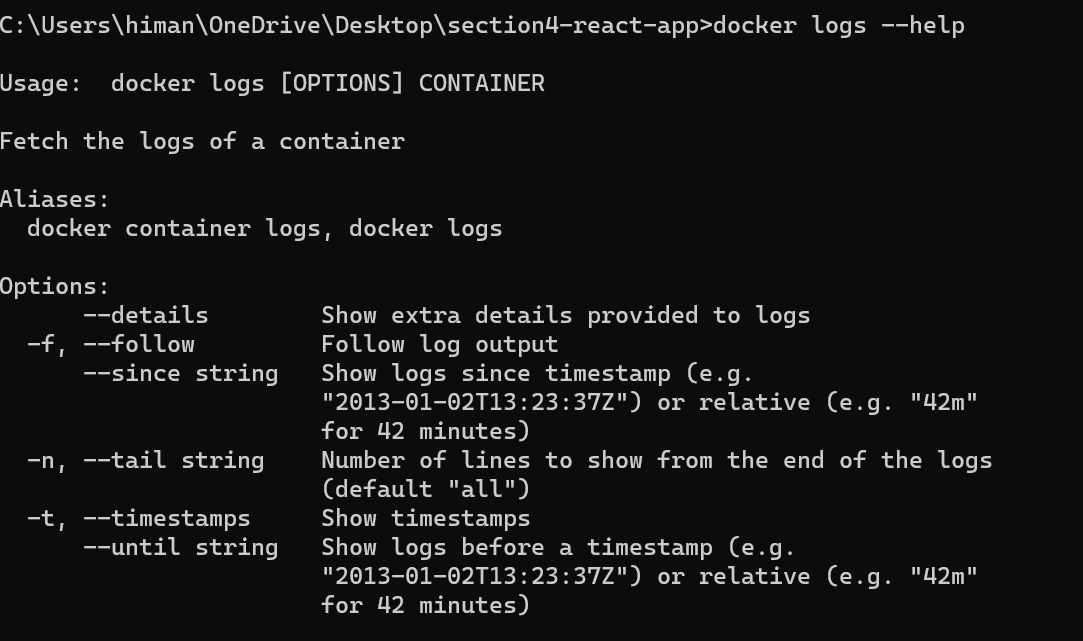
docker run -d --name blue-sky react-app



Now we have two containers and one of them is *blue-sky*.

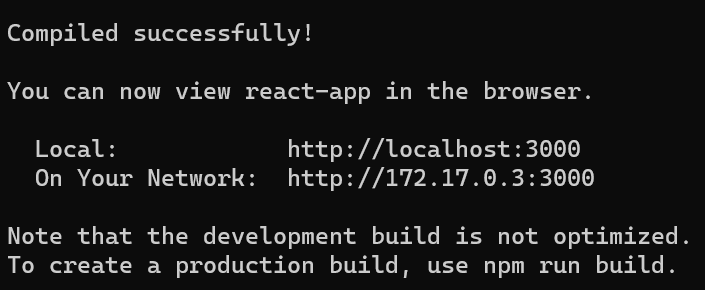
**Viewing the logs**:

So now we have two containers running in the background, but we don’t know what is currently going on inside these containers (*something might go wrong / server might generate an error*).



This is where we need to use *logs* command. Let’ look at 2c2 container-id logs.

docker logs 2c2



We can see the output of our webserver which is exactly the same output that we saw when we started this container.

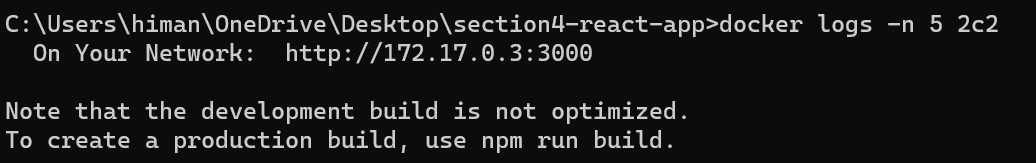
Couple of options with logs command,

🡪 -f or --follow (*this can be used when our container is continuously producing output*).

docker logs -f 2c2

🡪 -n or --tail (*with this we can specify number of lines to show, useful when we have a really long log*)

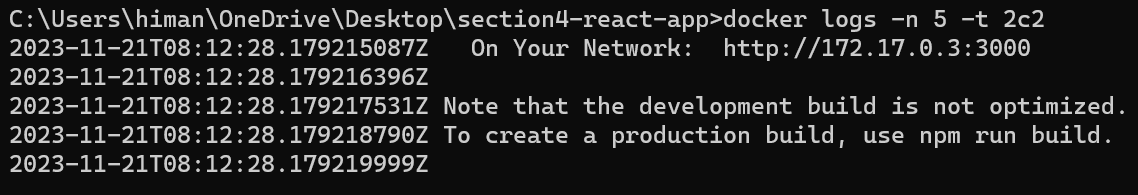
docker logs -n **5** 2c2



With this command we can see only the last 5 lines of the log.

🡪 -t or --timestamps (*if we want to look at the timestamps*)

docker logs -n 5 -t 2c2

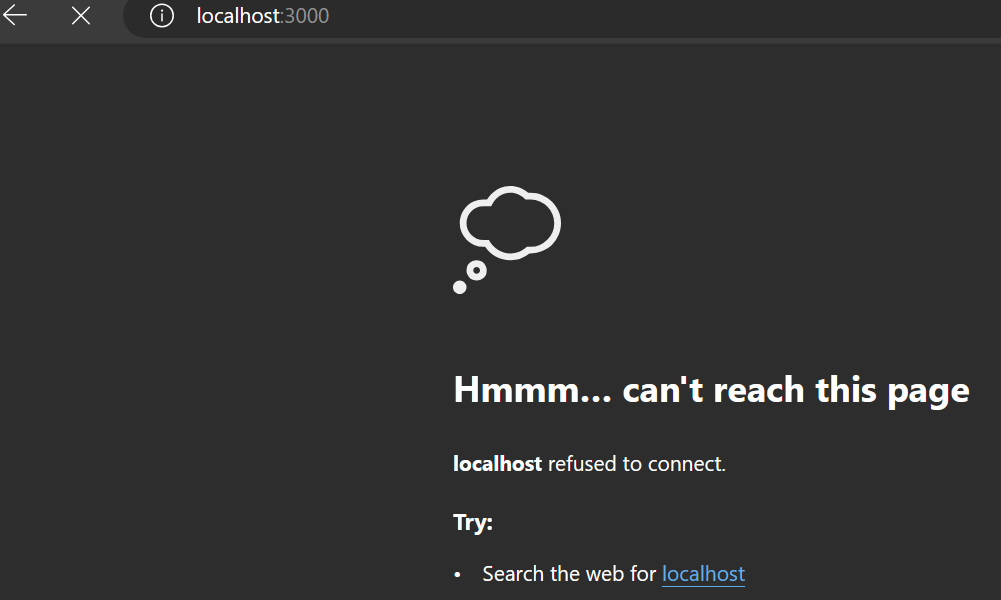


Last 5 lines of log along with timestamp.

*If you encounter any issues while running an application inside a docker container, look at the logs*.

**Publishing Ports**:

Currently we have two containers running our react application. But if we go to localhost: 3000 we cannot access this application.

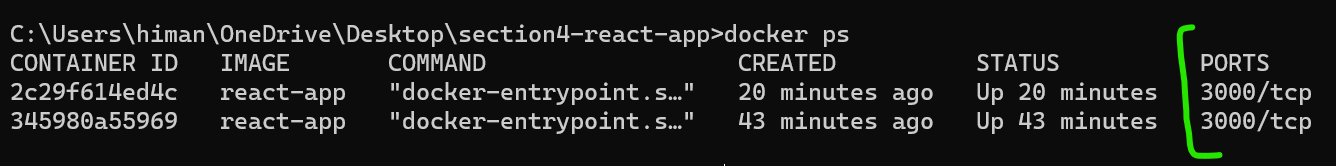


It is because port 3000 is published on the container not on the host.

So *on the same machine we have multiple containers and each of these containers is listening on port 3000 but the host itself is not listening on this port*.

There is no way to send traffic into the **localhost** :3000

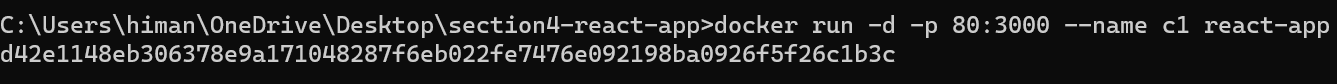
This is where we need to *publish a port*.



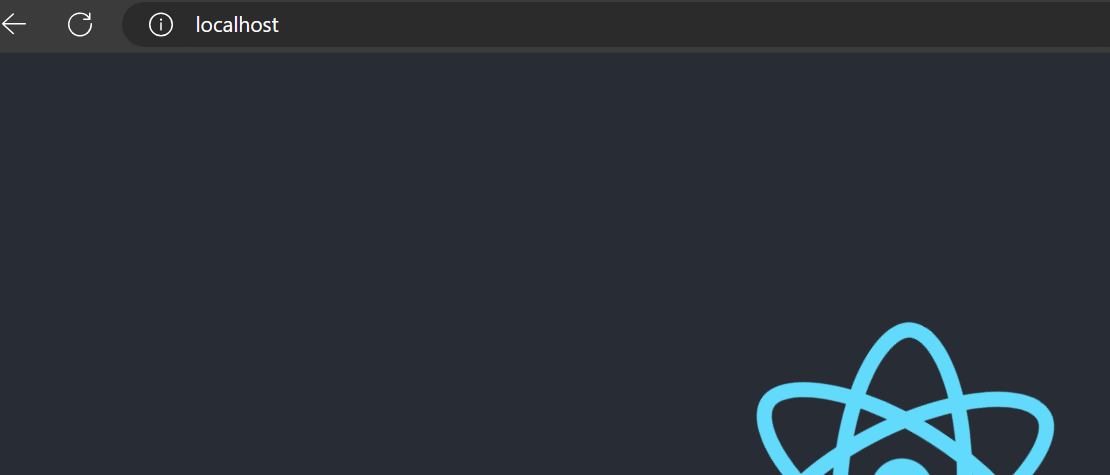
Here we have a column called PORTS and in this we can see ports with their mapping. Here both these containers are listening to port 3000.

Now we will start a new container and publish a port at the same time.

docker run -d -p **80:3000** --name c1 react-app



Here we published a port on the host (*here we used port 80 of host but we can use any*) and then we specified name of our container(*to identify our container*) along with image name.



At localhost:80 we can see our react application.

Let’s look at running containers again,



Notice the port mapping for our c1 container. *Port 80 of the host is mapped to port 3000 of the container*.(->)

**Executing commands in running containers**:

We learned that when we start a container it executes default commands that we specified in our Dockerfile.

USER app

WORKDIR /app

COPY package\*.json .

RUN npm install

COPY . .

RUN npm install

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

EXPOSE 3000

CMD ["npm", "start"] 🡪 *This one*

What if we want to execute a command in a running container later on. Let’s say *we want to troubleshoot something and want to look at the file system of that container*.

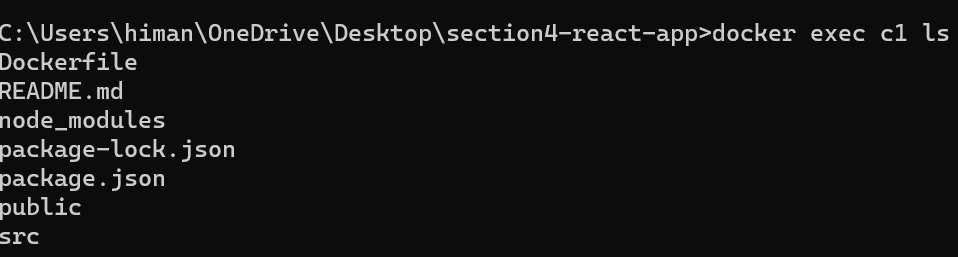
For doing so we have a *exec* command and with this we can execute a command in a running container.

Note: The difference between docker run and docker exec is that with *run* we can *start a new container and run a command* whereas with *exec* we *execute a command in an already running container*.

With exec we can run any operating system commands like linux or windows commands.

docker exec c1 ls (*here c1 is the name of our container*)

Here we used ls command to see the content of our *app* directory.



The reason we are in app directory is because in our Dockerfile we set the WORKDIR to /app.

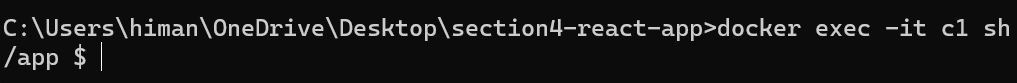
WORKDIR /app 🡪 *Here*

COPY package\*.json .

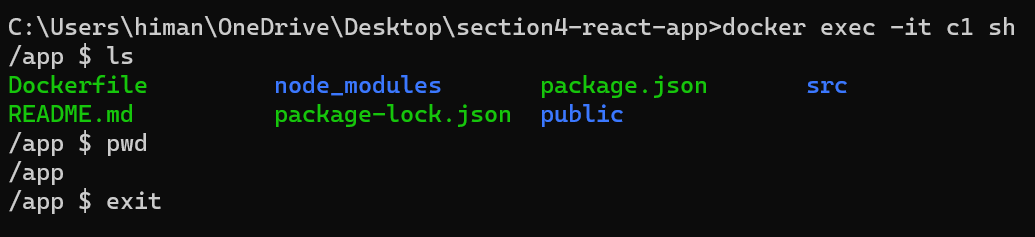
RUN npm install

Using the same command we can even open up a shell session.

docker exec -it c1 sh (*-it for interactive mode*)



Now we have a shell session inside this container.



When we are done we can type *exit* and it does not affect the state of our container (*it’s still running*).



So by using *exec* command we can execute any command on a running container.

**Stopping and Starting containers**:

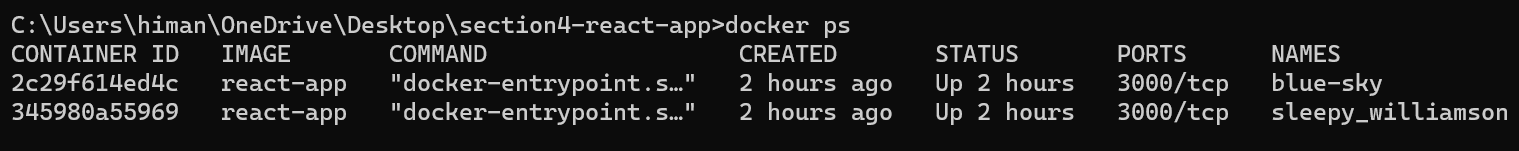
Earlier in the course we learned that a container is like a lightweight virtual machine, so it can be stopped and then restart it.

Let’s *stop* our c1 container,

docker stop c1



Let’s verify,



So if we go to browser now and refresh our localhost page, our application is not available now because the *container that was serving our application is no longer running*.

Let’s restart it using *start* command,

docker start c1

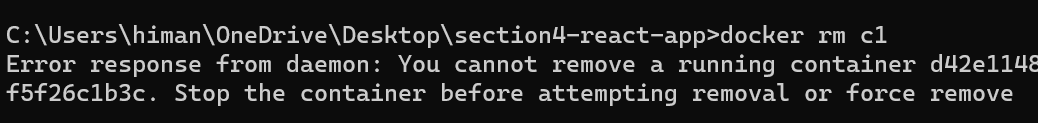


Note: The difference between docker run and docker start is that with *run* we *start a* ***new*** *container* while with *start* *we* ***restart a stopped*** *container*.

Now our application is being served again.

**Removing Containers**:

There are two ways to remove a container we can either type docker container rm c1 or docker rm c1.

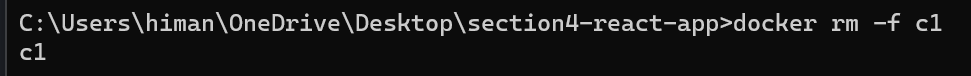


We get an error saying ‘*You cannot remove a running container*’.

So here we have two options. *One option is to stop the running container and then remove it while the other option is to use* ***Force*** *option*.

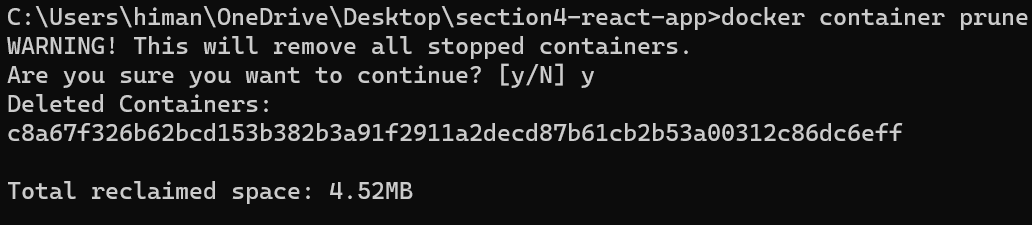
Using ***force*** option:

docker rm -f c1



Here we also have *prune* command and with this we can remove all the stopped containers in one go.

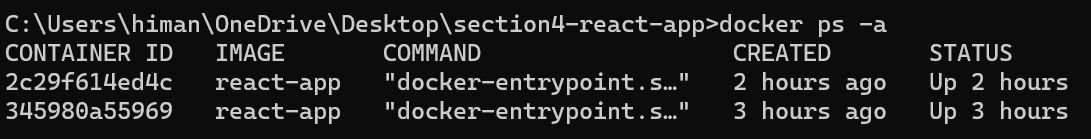
docker container prune



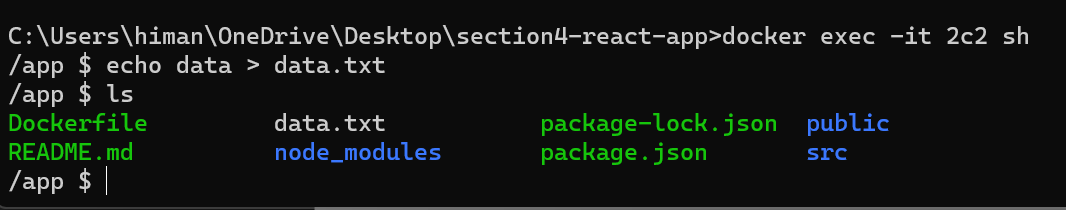
**Containers File Systems:**

Earlier in the course we learned that each container has its own file system that is invisible to other containers.

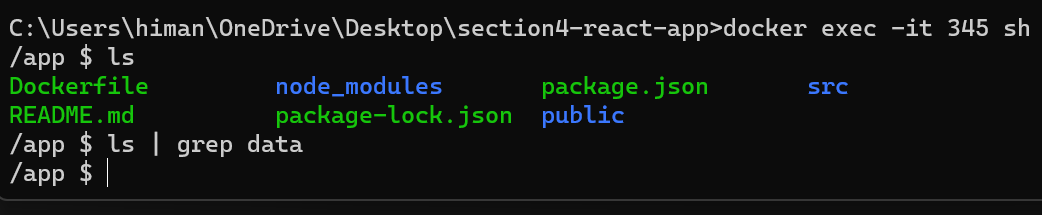
In this machine we have two containers,



Start a shell session on the first container and write something to a file in /app directory.



Then start a new shell session on the second container and see if the file is there or not.



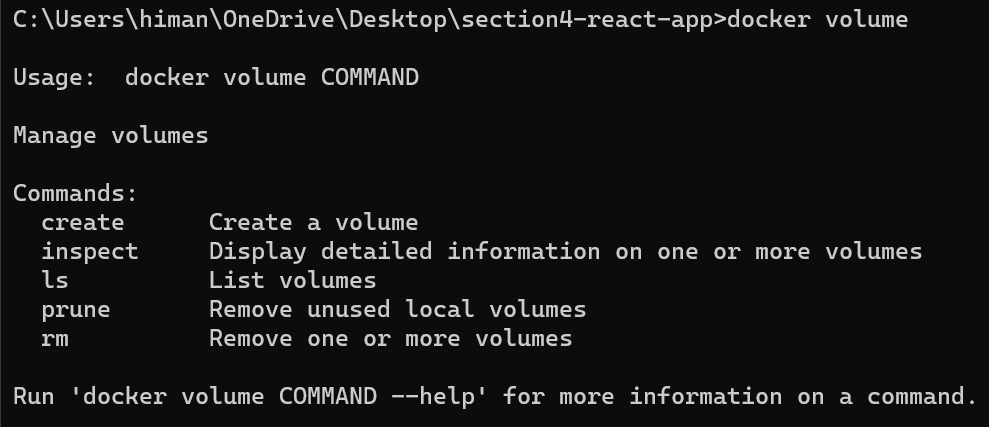
data.txt file is not here because each container has its own file system that is invisible to other containers and that means if we delete this container its file system will also go with it and we will lose our data.

So *we should never store our data in a container file system that’s what volumes are for*.

**Persisting data using volumes**:

**“***A volume is a storage outside of containers, it can be a directory on the host or somewhere on the cloud***”**

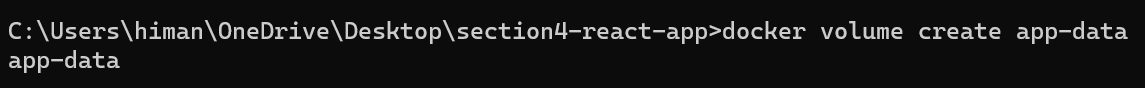
We have this command called docker volume which has some sub commands.



So we can create volumes, inspect, list, prune or remove them.

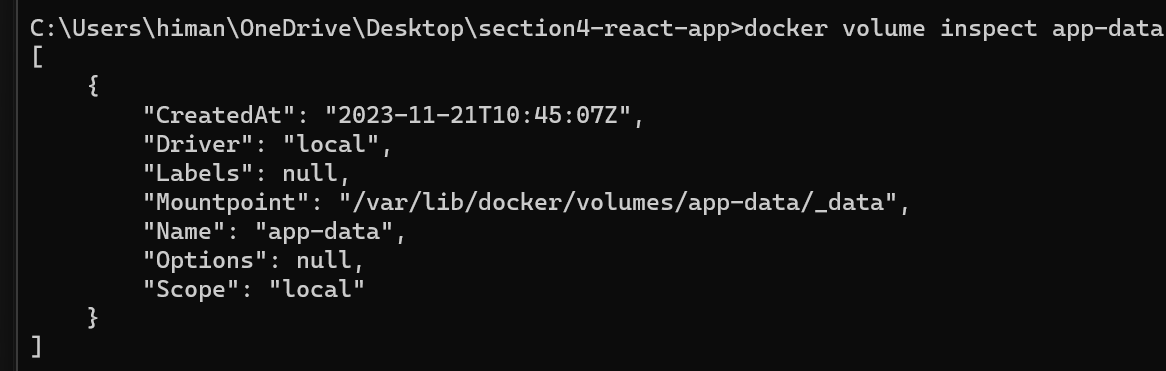
Let’s create a new volume called *app-data*,

docker volume create app-data



Now inspect it,

docker volume inspect app-data



Here we have a bunch of properties.

Note: *Driver : “local”* means this is a directory on the host (*default value*). We also have drivers for creating volumes on the cloud so if you use a cloud platform, you need to do your own research and *find a driver for creating a volume in that cloud platform*.

Here *"Mountpoint": "/var/lib/docker/volumes/app-data/\_data"* means this where the directory is created on the host.

Now we have a volume, let’s see how can we start a container and give it this volume for persisting data.

docker run -d -p 4000:3000 -v app-data:/app/data react-app

Break up of this command:

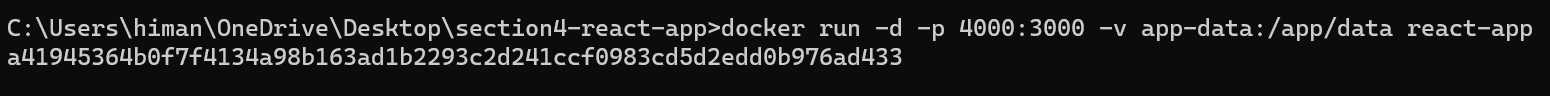
🡪 Run docker in detached mode.

🡪 Map port 4000 of host to port 3000 of the container.

🡪 We use a new option -v for volume and map *app-data* to a directory in the file system of the container (*here we type a* ***:*** *colon followed by an absolute path in the file system of the container*)

Note: We don’t have to explicitly create this volume before running this command, so if we write a new volume here, Docker will automatically create it.

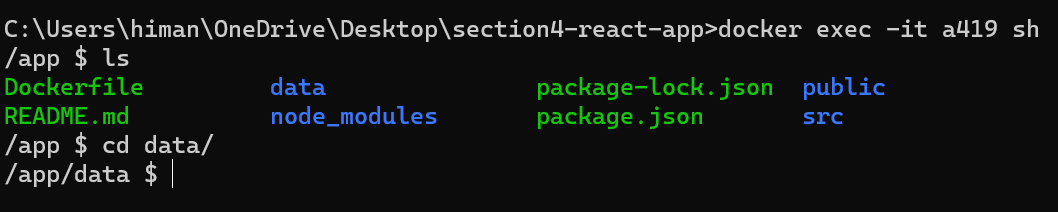
🡪 Name of the image.



Below we see our container ID.

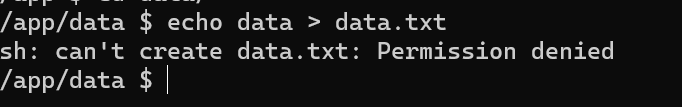
Let’s start a shell session,

docker exec -it a419 sh



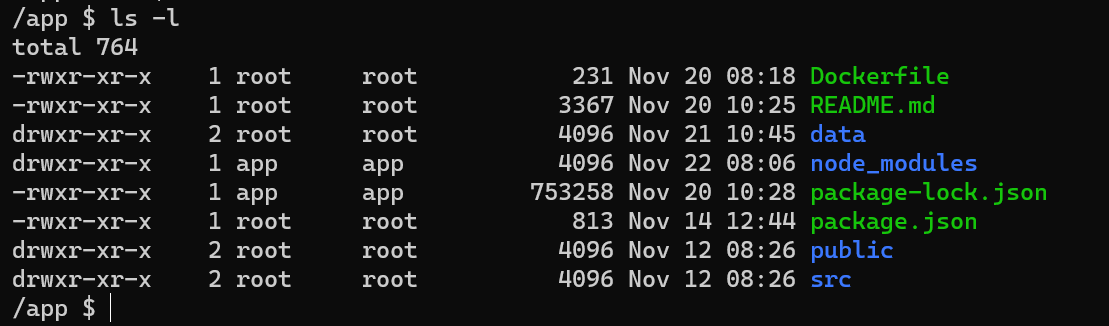
Now let’s something to a file in this directory,

echo data > data.txt



We get a permission error.

Here is the reason, go out of this directory and get a long listing ls -l :



Look at the *data* directory and the owner of the directory which is *root* user and as we can see only owner of this directory has the write permission.

So the *app* user, the user which runs our application belongs to the *others* group and this group does not have write permission for this directory.

The *reason we face this issue is because we let Docker automatically create this data directory for us*. So to prevent this from happening we have to go to our Dockerfile and after we create /app directory, we are going to run mkdir data.

FROM node:14.16.0-alpine3.13

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

USER app 🡪 *We set our app user*

WORKDIR /app

RUN mkdir data 🡪 *Same user creates our data directory*

COPY package\*.json .

RUN npm install

COPY . .

RUN npm install

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

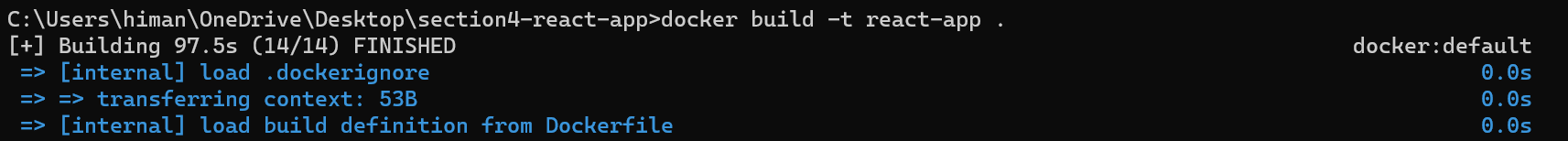
EXPOSE 3000

CMD ["npm", "start"]

This way we create this directory using the app user that we set earlier in this file and now this directory will be owned by the app user and it will automatically have the write permission.

But now we have to rebuild the image because we have modified our Dockerfile.

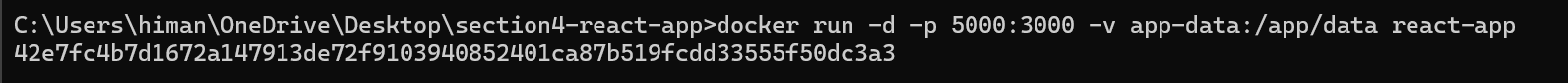
docker build -t react-app .



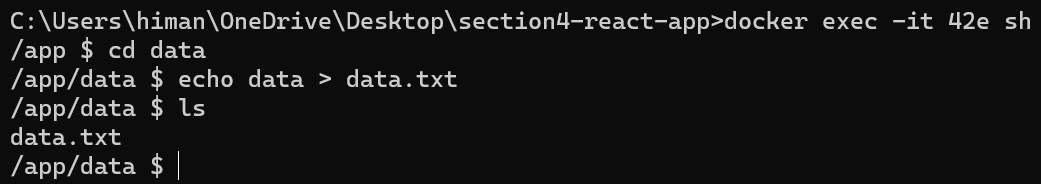
Now let’s start a new container,

docker run -d -p 5000:3000 -v app-data:/app/data react-app

But this time the *data* directory already exists inside the file system of this container and app user owns this directory.



Let’s open this container interactively with sh,

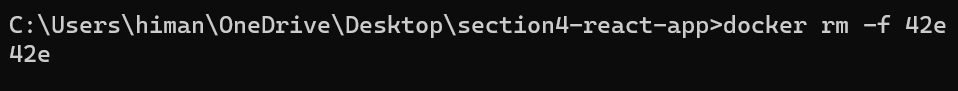


Now we can write something in this directory.

Here is the beauty of volumes, if we delete this container, this data.txt file will still exist because this directory is stored outside of this container, its actually a directory on the host.

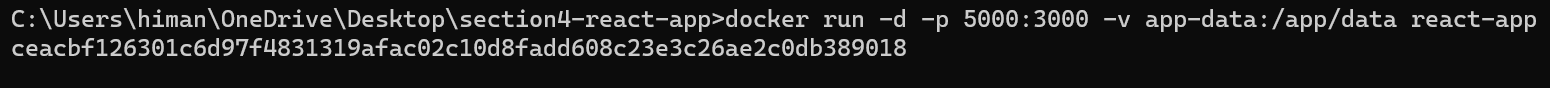
Let’s see this in action, by removing this container forcefully (*since its still running*)

docker rm -f 42e



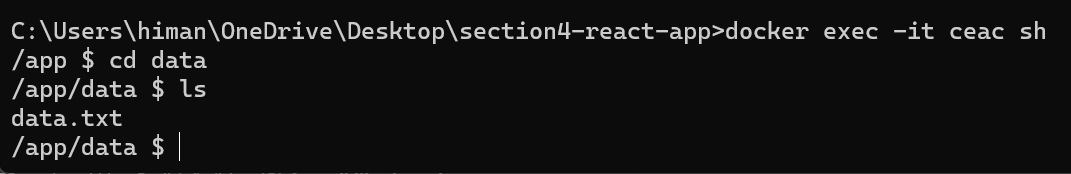
And Let’s start a new container with same volume mapping,

docker run -d -p 5000:3000 -v app-data:/app/data react-app



This is a brand new container and we run a shell session with it,

docker exec -it ceac sh



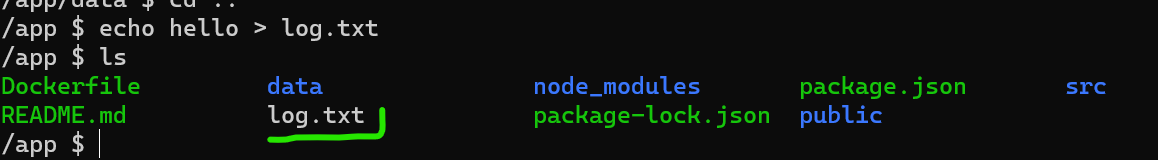
Our data.txt file is still here. So volumes are the right way to persist data in Dockerize applications because they have different lifecycles from containers. If we delete a container that associated volume is not deleted, it still exists.

And also we can share a volume amongst multiple containers.

**Copying files between Host and Containers**:

Sometimes we need to copy files between host and a container. For example, let’s say we have a log file in our container and we want to bring it to host and analyse it.

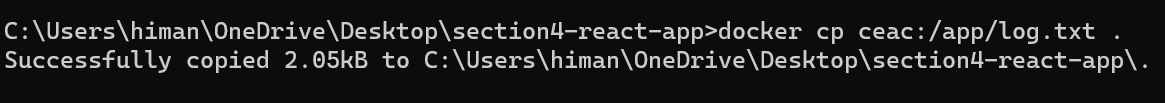
Here in this container we created a log file in the app directory.



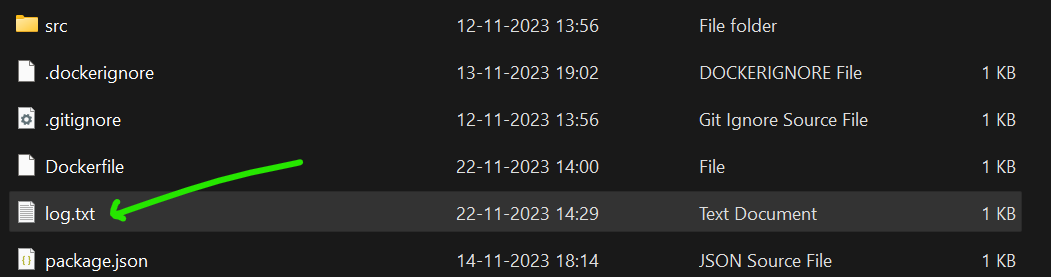
Now let’s copy this file from the container to the host using *cp* command.

docker cp ceac:/app/log.txt .

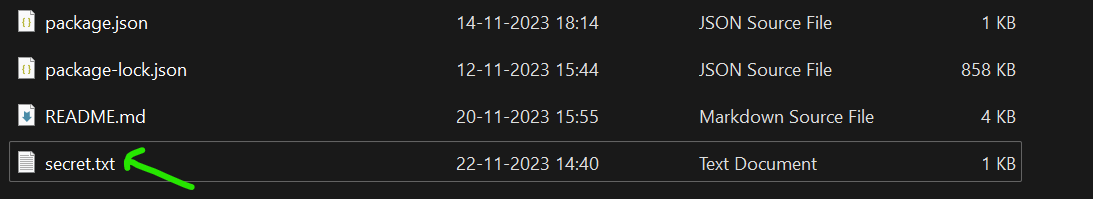
First we need to specify a source (*container-id* ***:*** *full path to the directory*) and then destination ( *use* ***.*** *for using current directory*)



Our log.txt file is copied inside our current project directory,

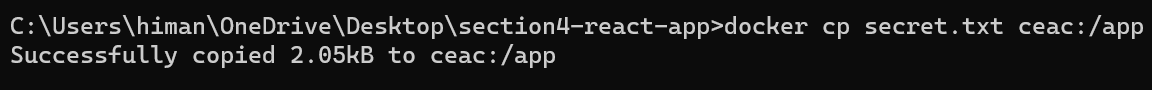


We can also copy a file from host to container. So let’s create a secret file in our project folder.



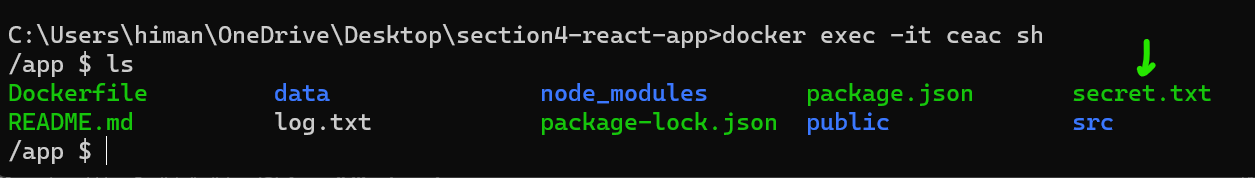
docker cp secret.txt ceac:/app

This time source is secret.txt and our destination is app directory of our container.



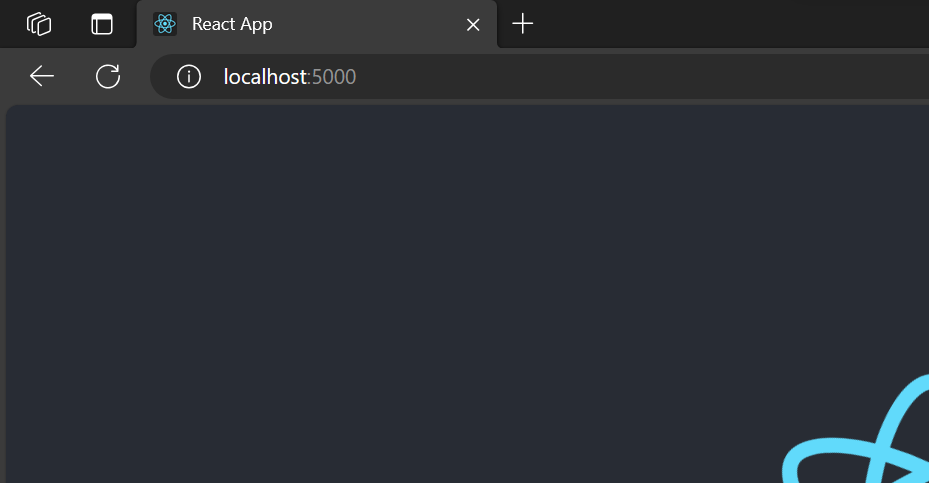
Let’s run a shell session,

docker exec -it ceac sh



**Sharing the source code with a container**:

So we have this react application available at localhost port 5000,



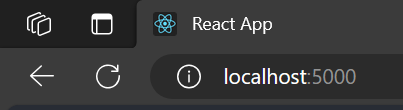
Now let’s see how can we publish our application changes.

So back to our react application code, go to public directory and open index.html

Here we will update the title of this app,

<title>Dockerized React App</title> 🡪 *Change here*

Now if we refresh our browser, we don’t see any changes.



We still see the old title.

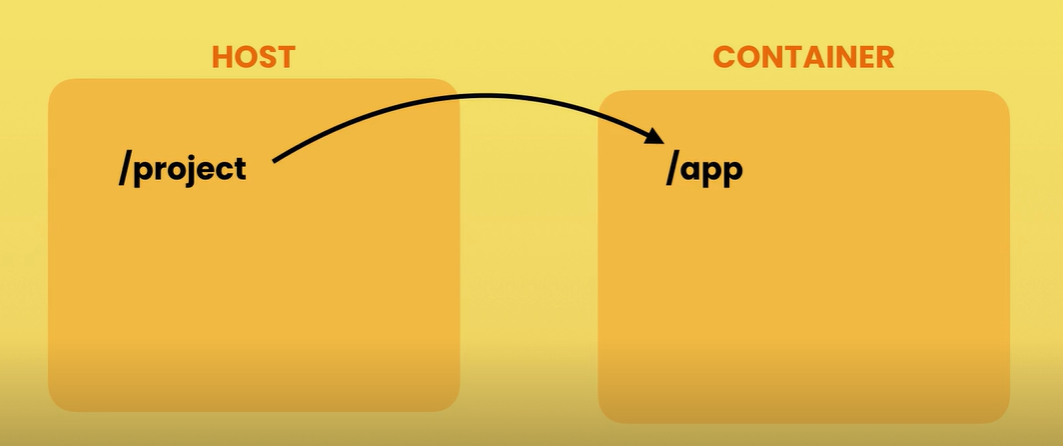
For production machines, we should always build a new image, tag it properly and then deploy it.

But what about development machines, we don’t want to rebuild the image every time we make a tiny change in our code. That’s just too time consuming.

Same goes for copying as well. We don’t want to manually copy files from our development machine into a container every time we change our code.

Here is the solution…

**“***We can create a mapping or binding between a directory on the host and a directory inside the container. So this way, any changes we make to any files in the directory are immediately visible inside the container***”**.

****

Back to the terminal, we are going to start a new container. But first remember the volume mapping command,

docker run -d -p 5001:3000 *-v* app-data:/app/data react-app

Note: Here after *-v* we map app-data volume to /app/data directory inside the container. Now *using the same syntax we can map a directory on the host to a directory inside the container*.

So instead of using a named volume which is a directory that docker manages, we will use the current directory. The directory that holds our application.

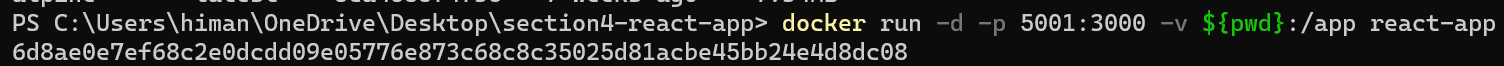
Here we don’t type the full path and just use the pwd (*print working directory*) command.

**BIG NOTE**: *Use PowerShell to run this below command*

docker run -d -p 5001:3000 -v **${**pwd**}**:/app react-app

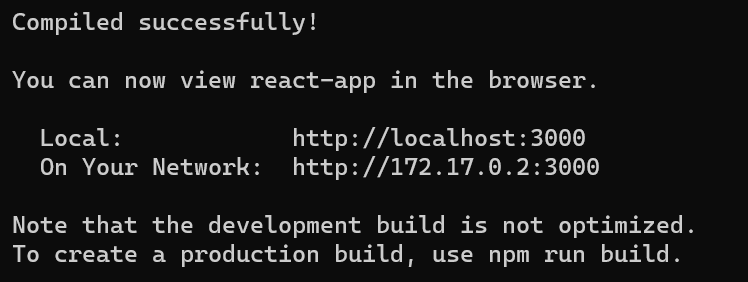
Note: Here we wrap *pwd* command inside ***$()***, so that docker does not consider it a volume name but instead a complete path.

So when we execute the above command, the **${**pwd**}** part of the command is evaluated first and the result will be a full path to the current working directory and then it is mapped to app directory inside the container.

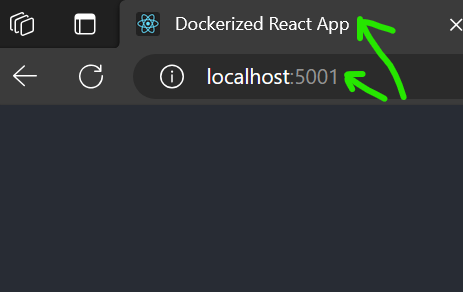


Let’s see the logs with follow option so that we can see the changes as they come up,

docker logs **-f** 6d8a



We can see the updated title on port 5001.



Now let’s make one more change to the application code. So let’s add an ! at the end of title.

**Issue**: After adding exclamation mark, title is not updating. The summary of problem is that *react hot reloading is not working after mounting current directory as a volume in docker even though file changes are visible inside container*.

POSSIBLY, *Need to fix npm start react-scripts part in package.json with --open ssl flag, since it’s a very old react version. But* ***for now changes are applied after stopping and starting the container***.

So moving on…

*Summary of this lecture is*,

To share our source code with a container, once again we use the *-v* option to map the project directory to a directory in the container file system.