So by now we had complete setup of docker environment installed either on one of the platform (Linux/Windows/MAC).

This section we are going to learn on Docker Images and Containers.

Containers: containers are the packages of software unit which contains the code and the necessary run time environments for your applications.

Images: Images are the Templates/Blue prints for containers and it contains the code plus the required tools/ runtimes.

The running “unit of software”

Containers

Templates/Blueprints for containers

Images

Docker

We can build and image by packaging all the code and run time environment. By using this image we can build multiple containers are required. Image is a shareable code that can shared between multiple environments, by using this images we can build number of containers easily.

Container

Container

Running NodeJS App

Running NodeJS App

Running NodeJS App

Container

NodeJS App Code

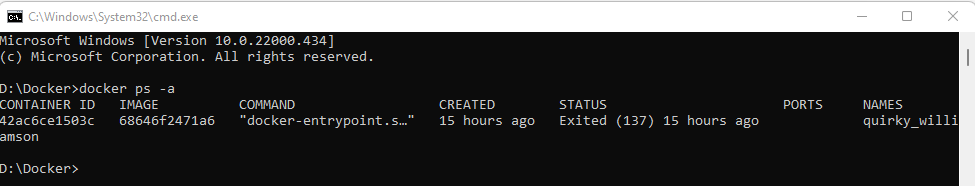
NodeJS Environment

Image

Containers are the running instances of the images, there are actually two ways building images

1. Use an existing, pre-built Image (Eg. Dockler Hub)
2. Custom Images

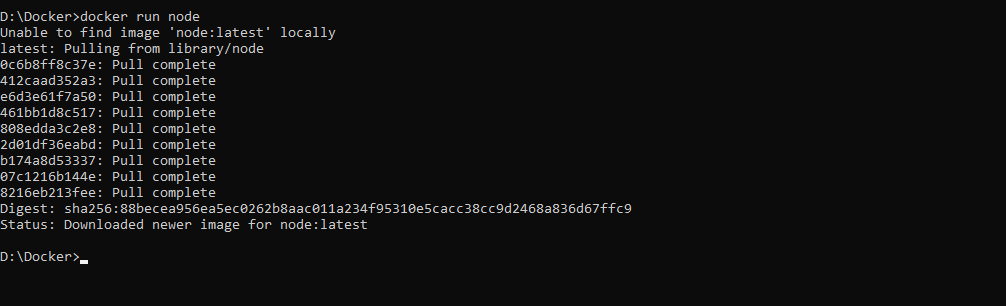
You can open a terminal from any folder and run the following command to run the container.

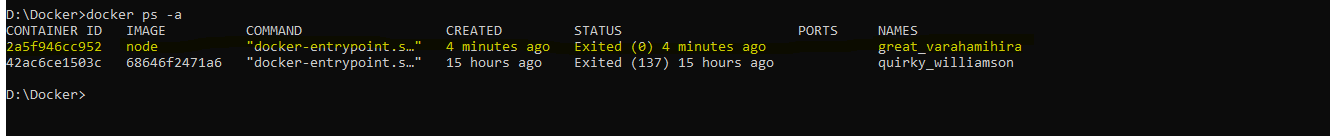


Normally you find all the stable and latest images on Docker Hub

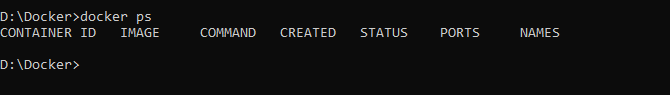
<https://hub.docker.com/search?q=node&type=image>

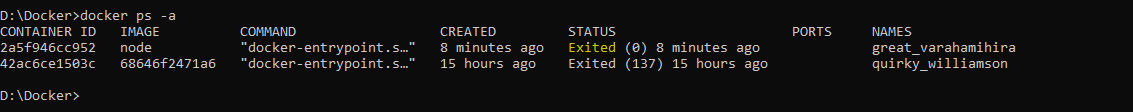






As of now we had downloaded and build a container based on Image, but it is not running or not doing anything much as of now because the interactive shell comes with the container doesn’t exposed to us by default.

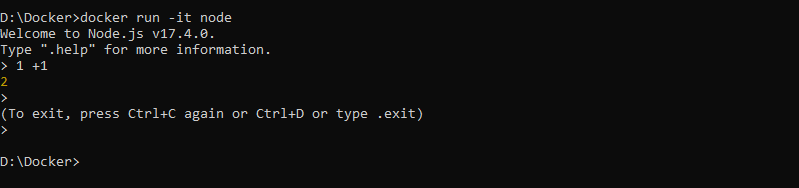


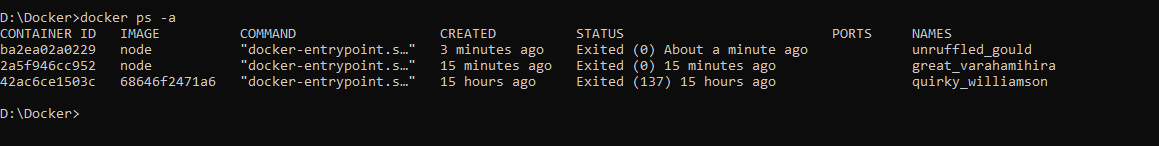
****

If we repeat the command by providing the option “-it” which means we are asking to expose the command interactive shell from the container to the host Operating system as shown below.

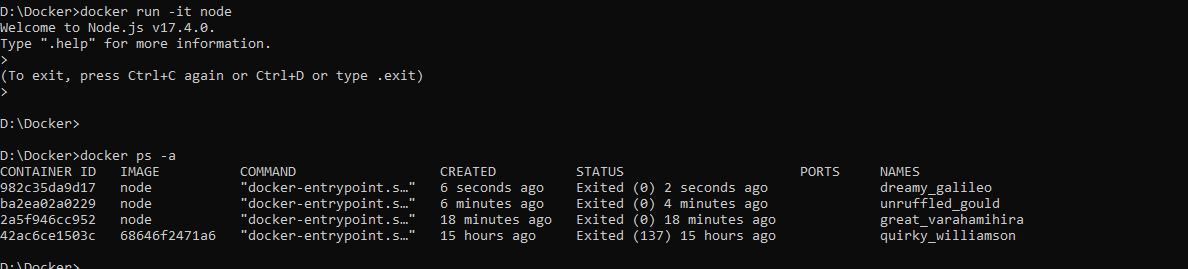


Type Ctrl +c twice in order to come out of the container and shutdown the container.





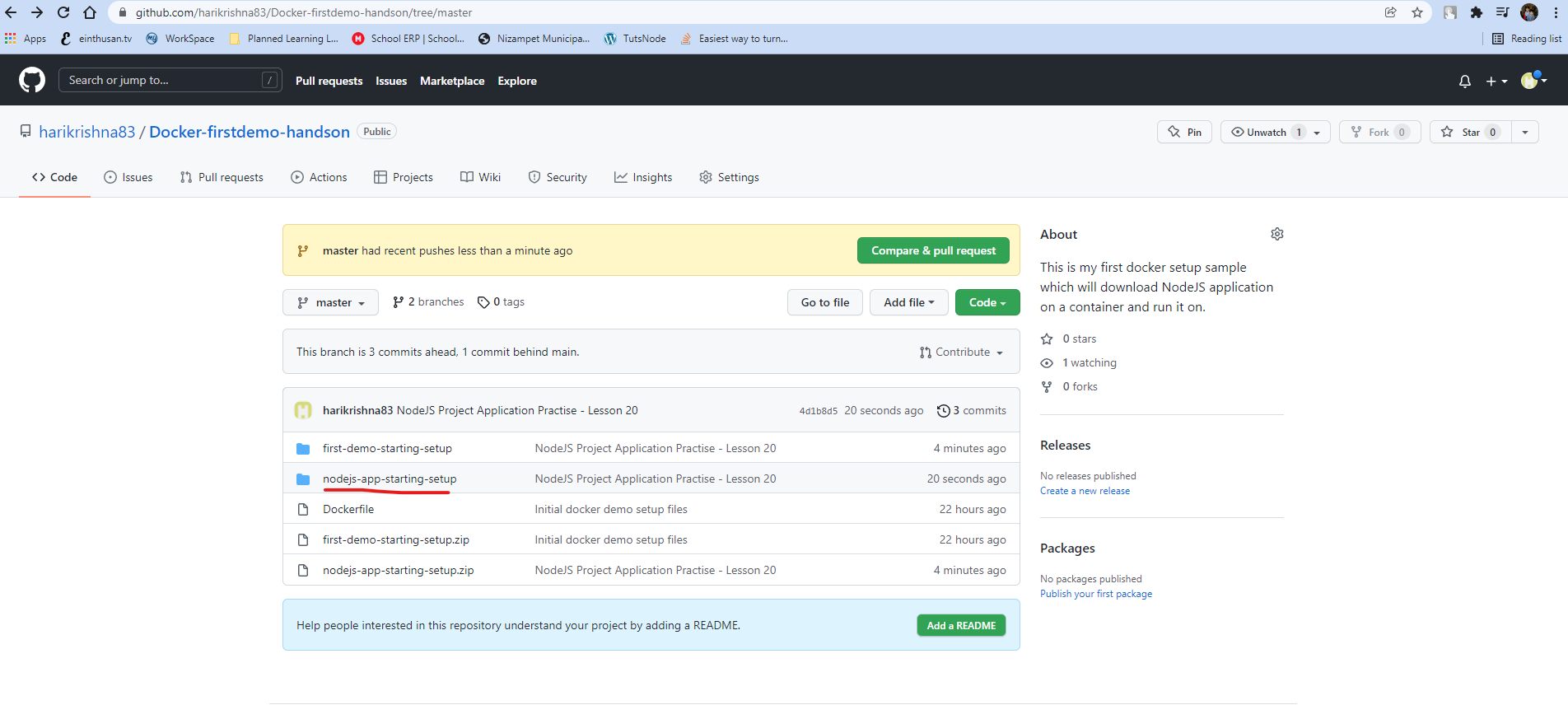
Note: Docker run command will create a number of instances from the image, so if you run the command “docker run” how many times so you will find that many instances.

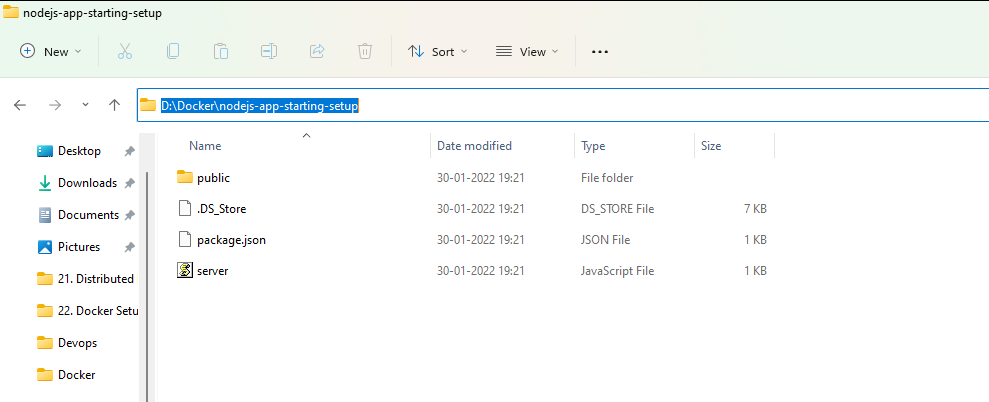


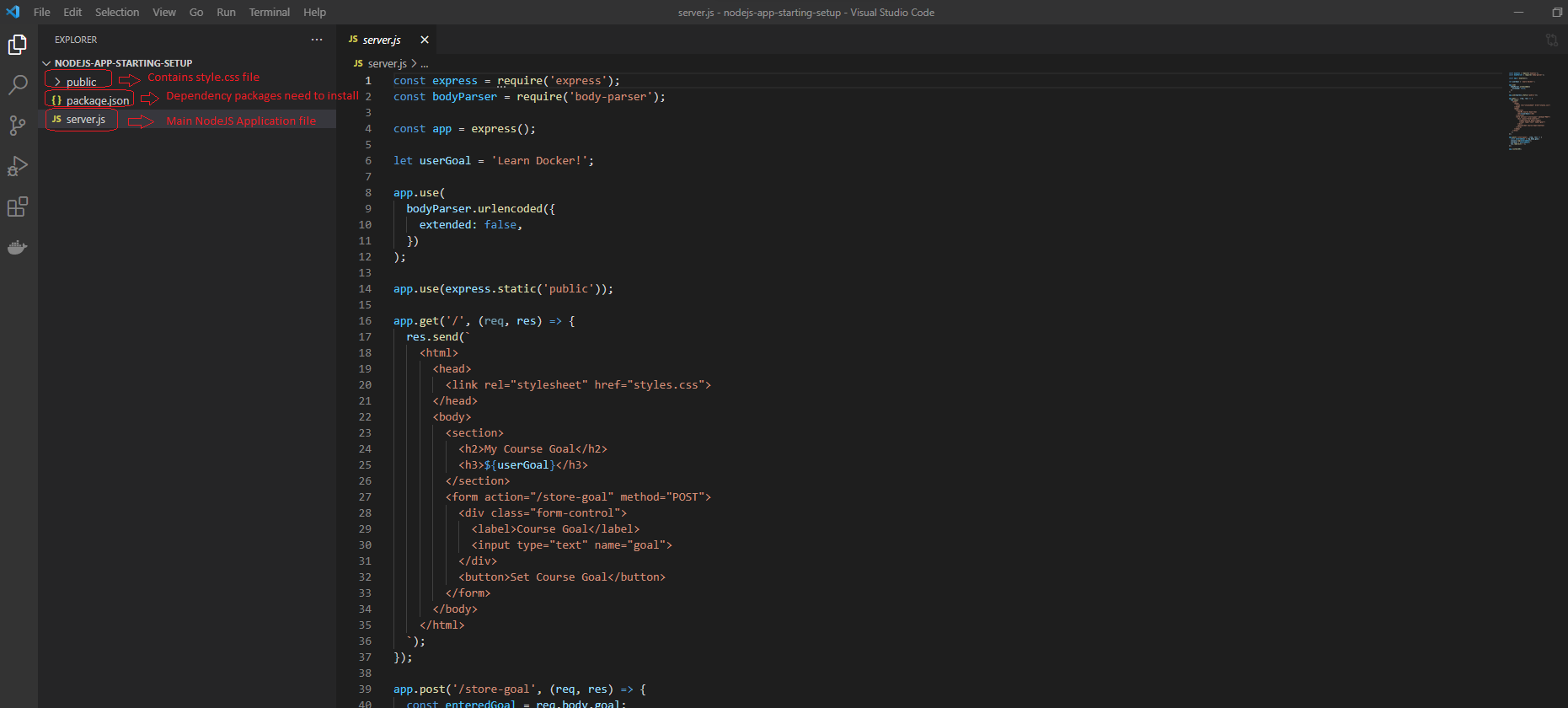
As you observed on above screenshot all the docker instances are created based on using single image “node”.

Now we are going to setup a NodeJs project from my GitHub for my practise.

<https://github.com/harikrishna83/Docker-firstdemo-handson/tree/master>

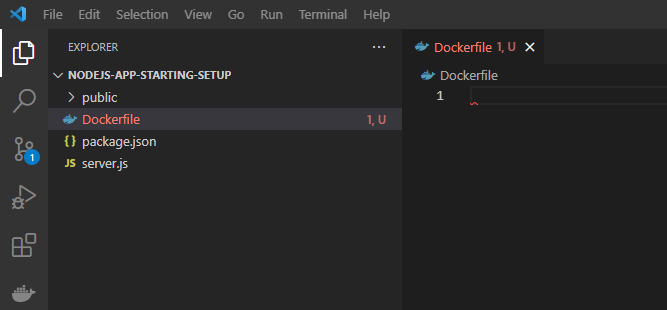






Now we all set with a NodeJS application and will create the Dockerfile to build this as an image with our custom configuration.

To build this custom image we need to create a file name “Dockerfile” at your project directory with all the instructions. We will write all the instructions to build the docker container on that file.

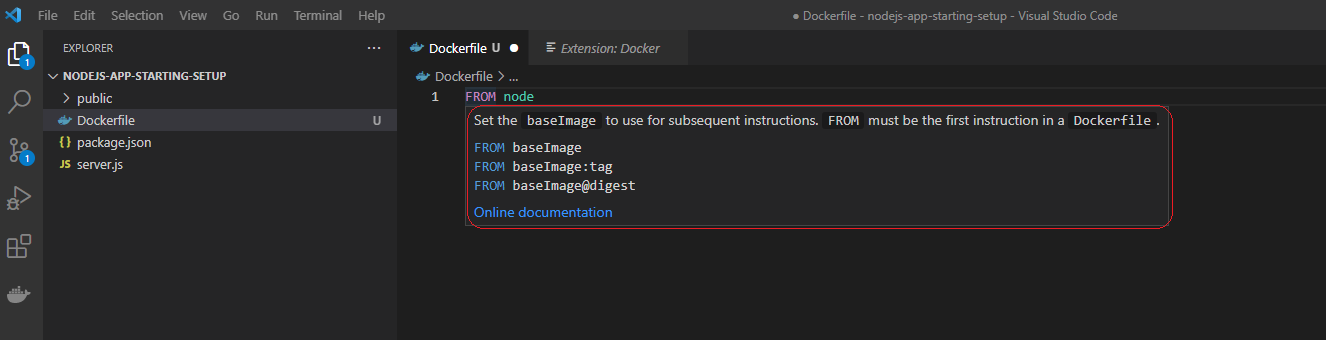


Note: Please make sure you install the Docker extension for the visual studio code.



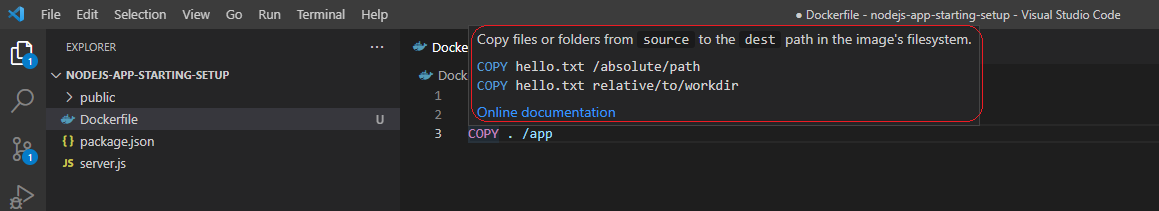
Now this Dockerfile will contain the set of instructions how to build the container by using a base image.

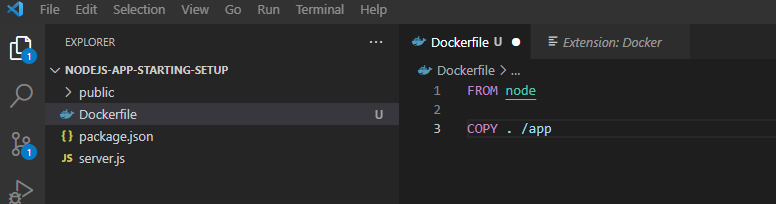
We will start with the first instruction “FROM” on the Dockerfile which will pull the base image from locally if exists, if not it will pull or downloaded it from dockerhub and cached it locally for next time that base image. On top of that base image we will build NodeJS configuration.



Now next instruction we need to tell the docker that which files that need to go for our docker container. For this we will use the instruction COPY, the COPY will take two arugments one is source means files from the host operating system and other is destination, which describes the destination on the container.

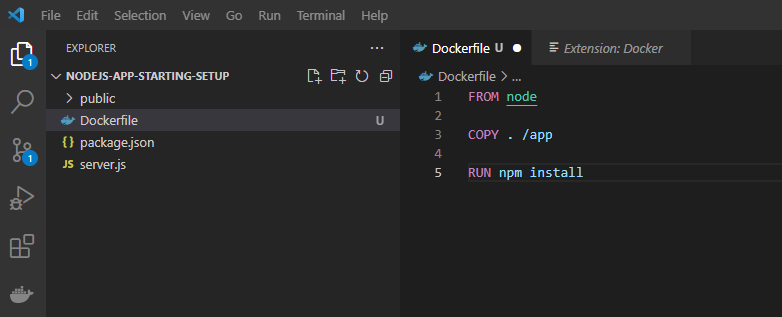
Note: By default if you give ‘.’ on source it will consider all the file from the project directory. If you give ‘.’ On destination it will consider the root directory on container. Generally it is advisable to mention some other directory instead of root directory on container, so I mentioned /app to create the directory if it doesn’t exists and copy all the files to the /app directory.



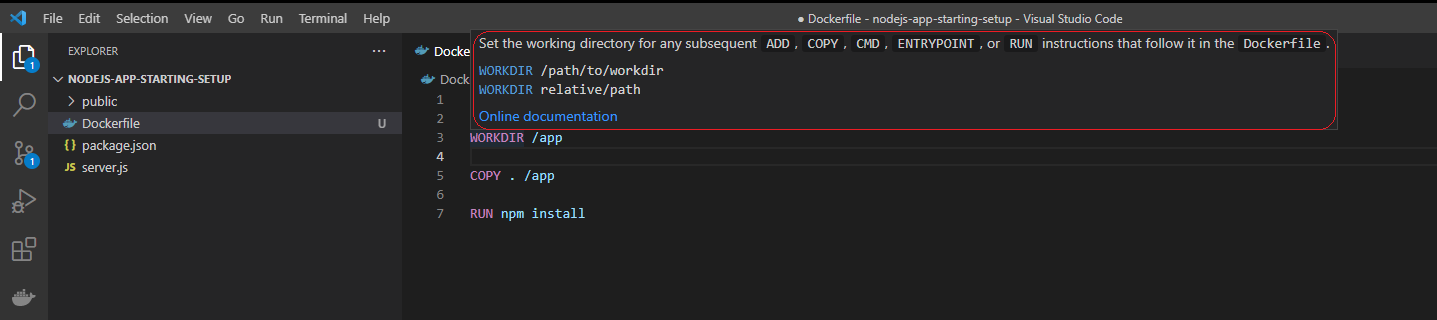


Next step is to install all the dependencies for the NodeJS application, for that you can say to run the command by using ‘RUN’

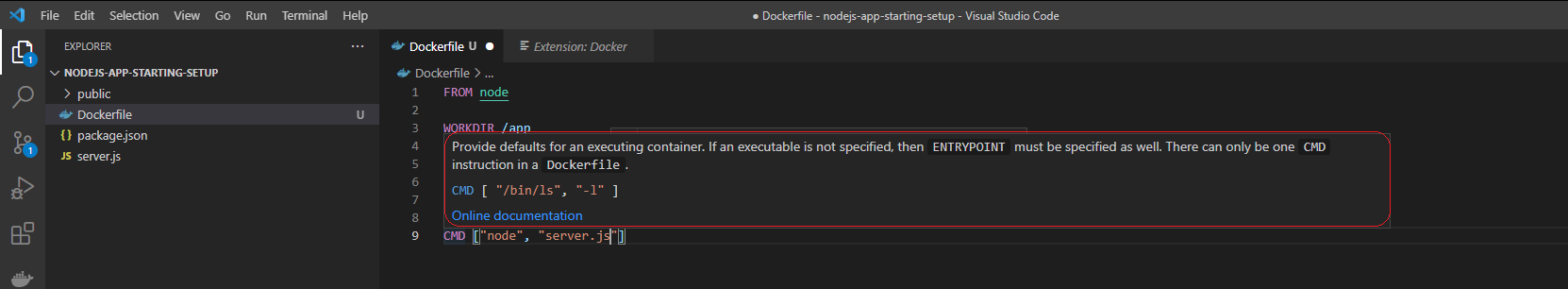




But here there is a problem, by default the working directory of a container is root, but here I had copied the files under /app directory, so command npm install will looks for the root directory of the container, so to avoid this we need to set a WORKDIR to /app

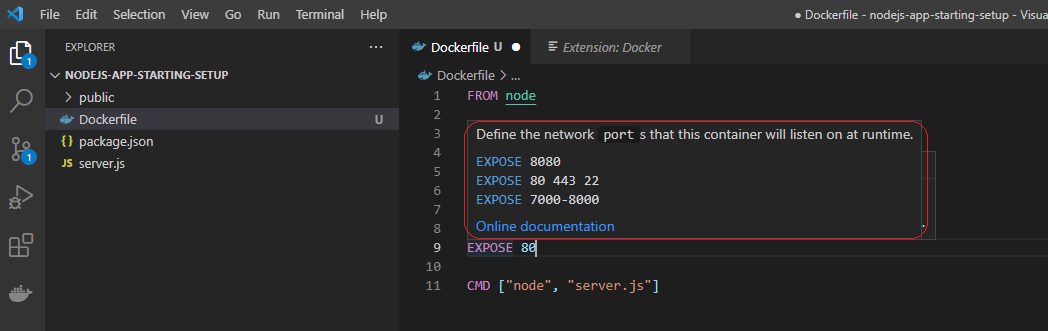


After installing the dependencies, we need to run the command “node server.js” which will run the server file. For that you need issue the command RUN on docker file, but if you use the RUN command there is a problem that it will tell the container while building the image run this command that’s not what we want as of now. We want once the container is built properly and on container at run time this command need to run. For this we need to use CMD instead of RUN on docker file. The syntax for the CMD is little different, we need open an array and provide the commands inside quotes.

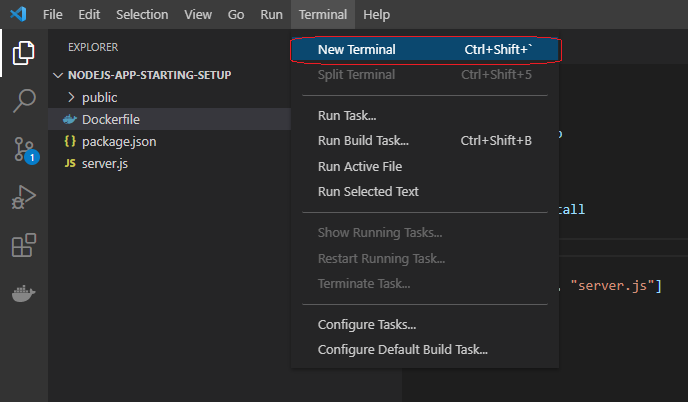


The different between RUN and CMD is RUN will execute while building the image and the CMD will execute the command once the container build is completed and at execute at run time.

By default the docker container is isolated and by default it is not exposed to outside, so as the application listens on port 80 we need to expose the port from the container to outside by using the keyword “EXPOSE”.

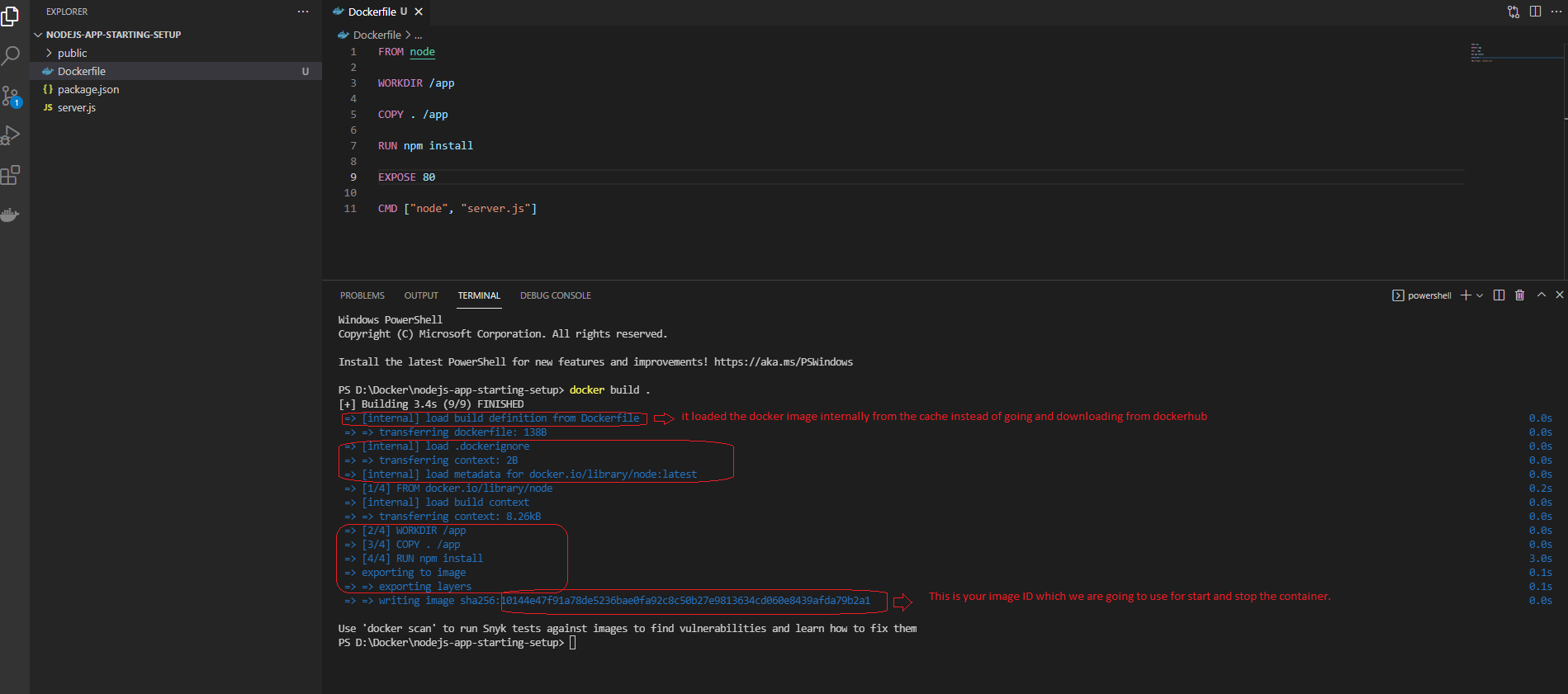


Now we got all the instruction to build and run a container, so in order to proceed further open a terminal on visual studio code and execute the respected docker command as shown below.

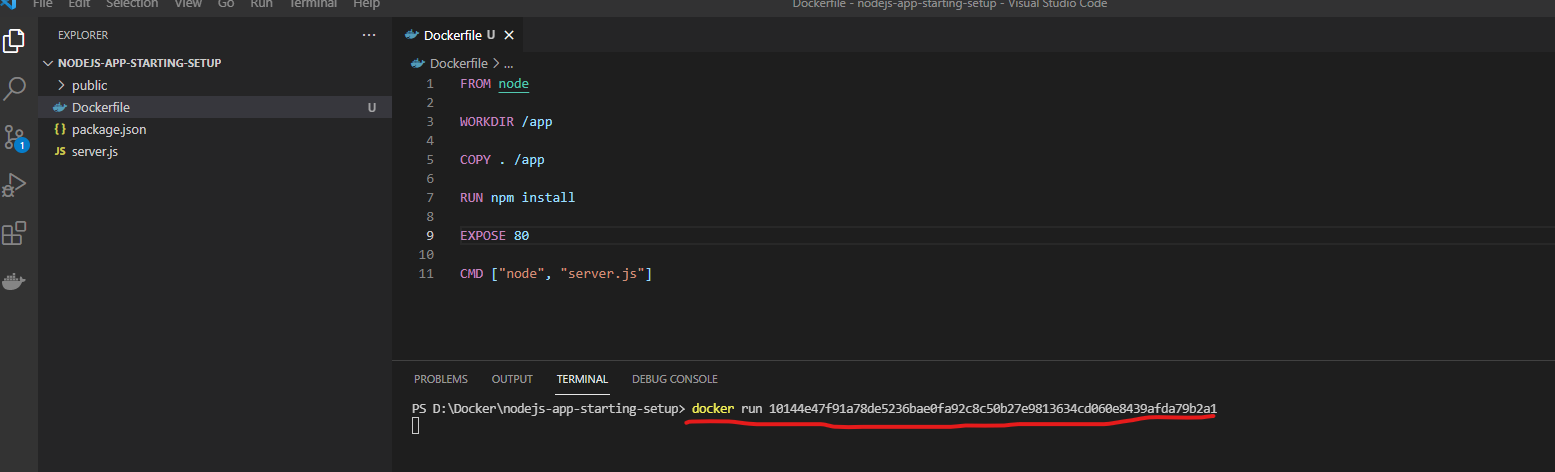


Now we will issue a first command to build the image by using the Dockerfile from the terminal.

Note: in this command “docker build .”, the dot represents that Dockerfile is there at the present directory.



To run the container that was built on above, you need to issue the command “docker run <Image ID>

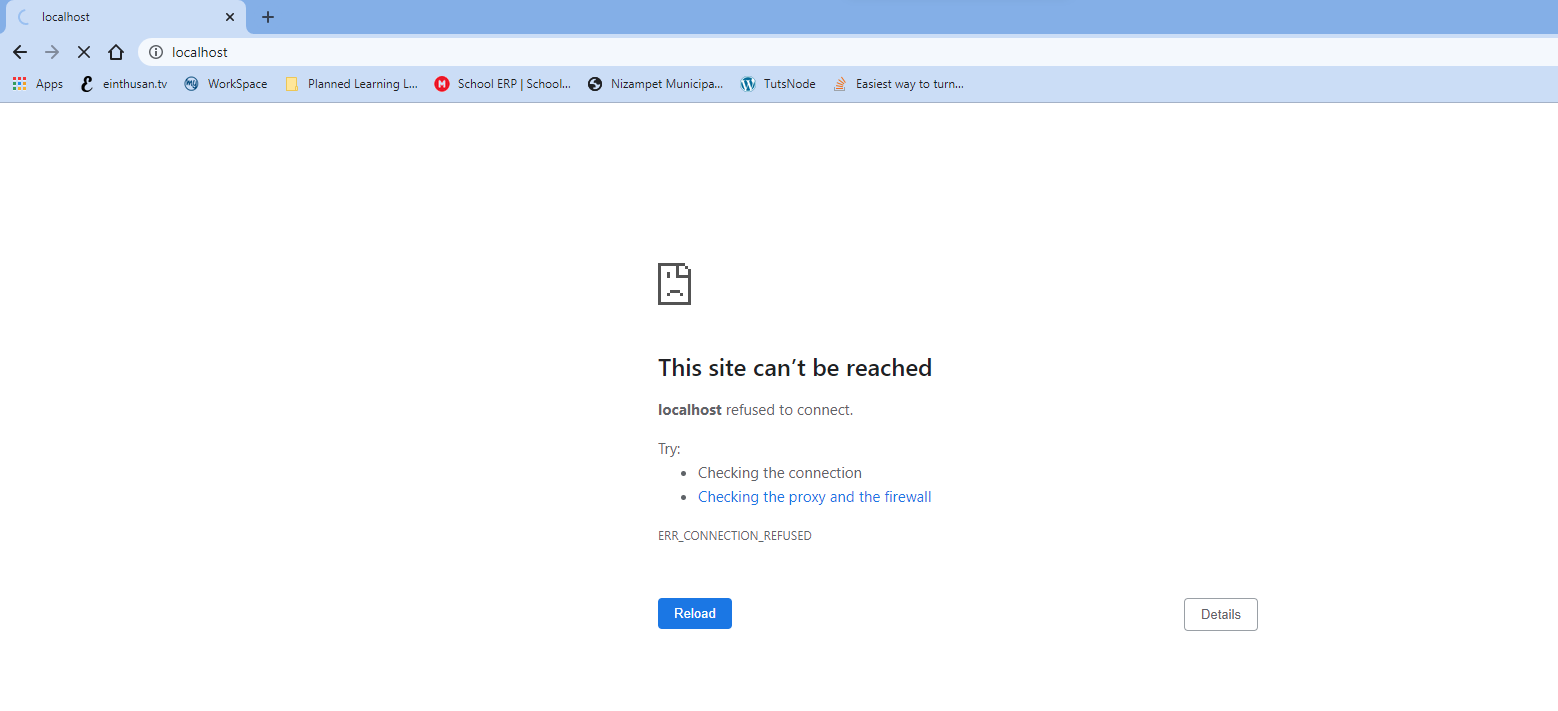


As you can see the container is still running because of the instruction CMD on the docker file, the command issues to start the NodeJS process to run.

Open another terminal by clicking on “+” and issue the command “docker ps”.



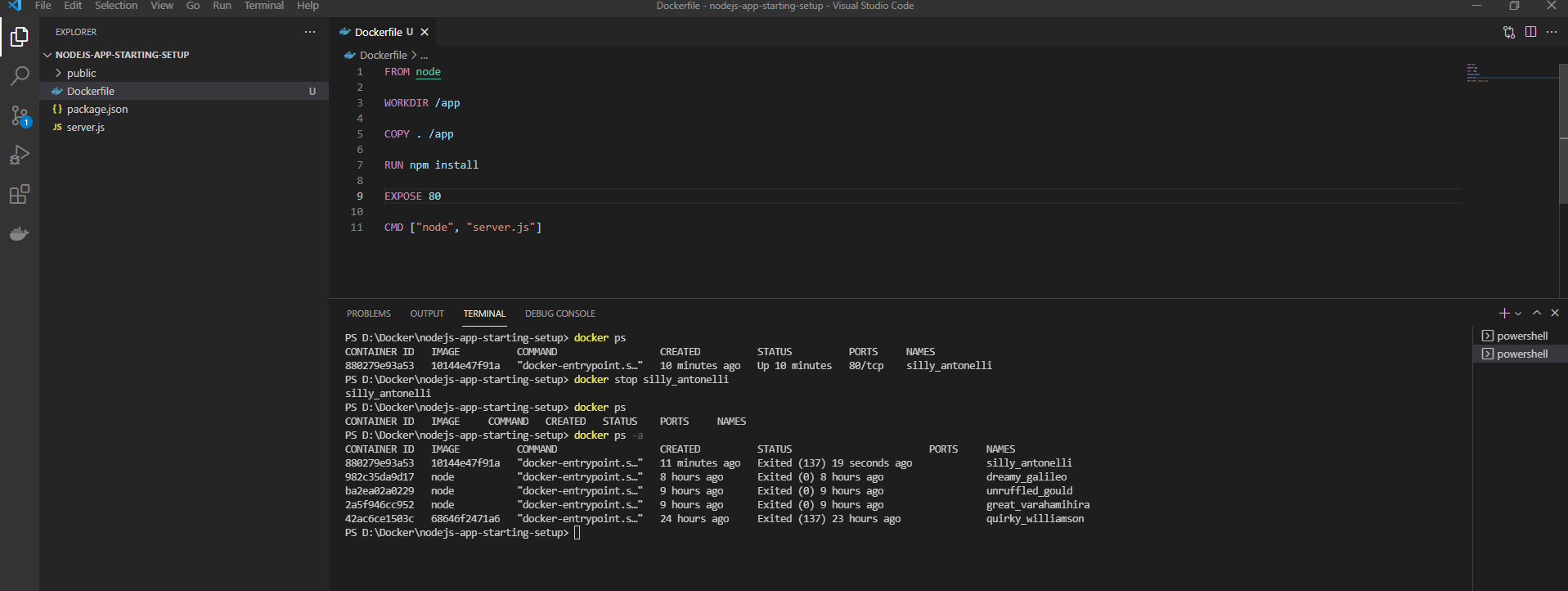
But if you browse it on locahost on 80 port it won’t work.

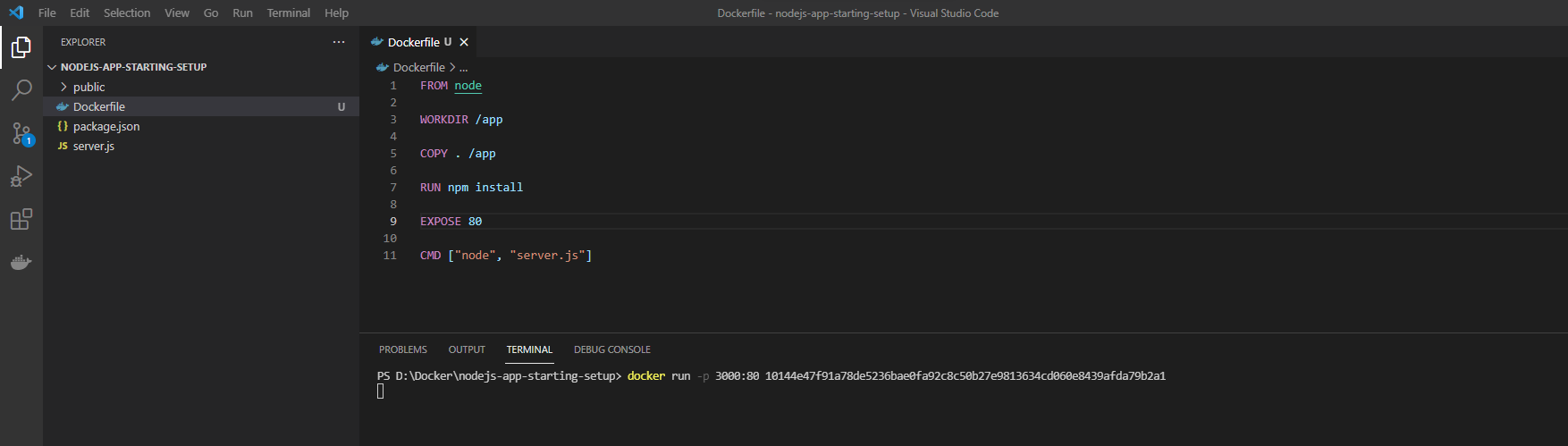


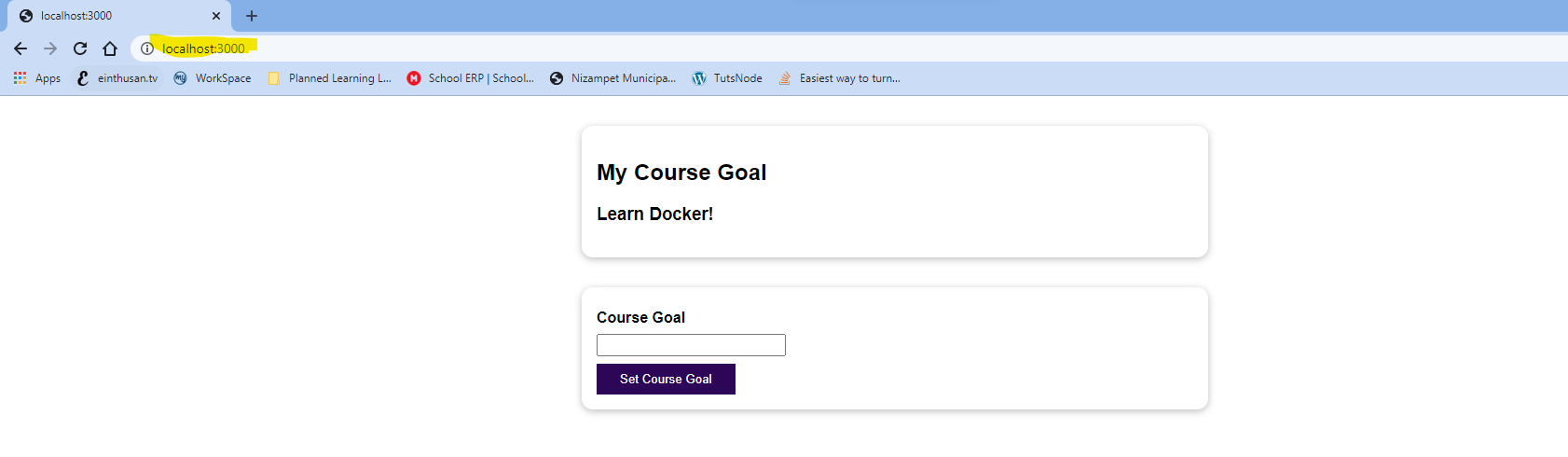
If you analyse the reason why it is not working even though we expose the port 80 on Dockerfile, as we know docker is an isolated container and by default it doesn’t expose the port to outside we need to mention “-p” to the docker run command in order to expose the respect port

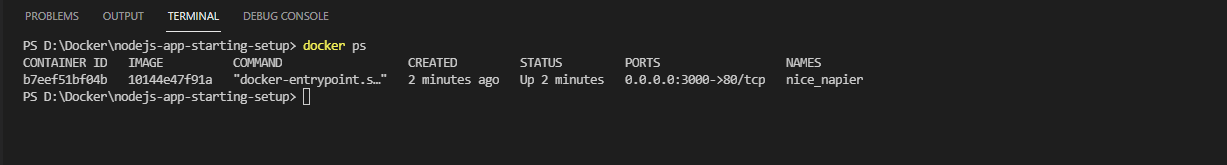
Note: option “P” stands for publish which has a syntax of “-p <localport>:<container port>”

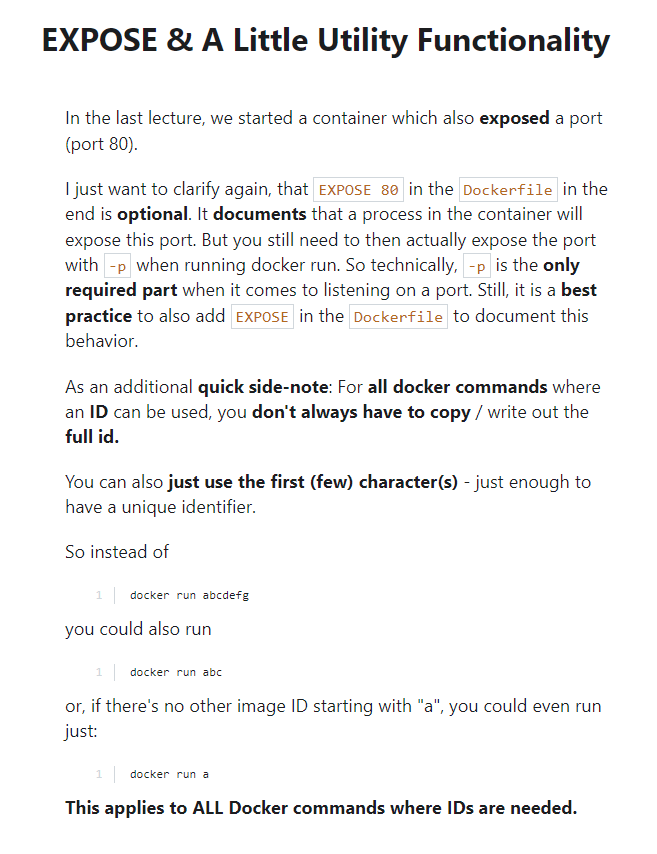
Now shutdown the running container and restart it with -p option.





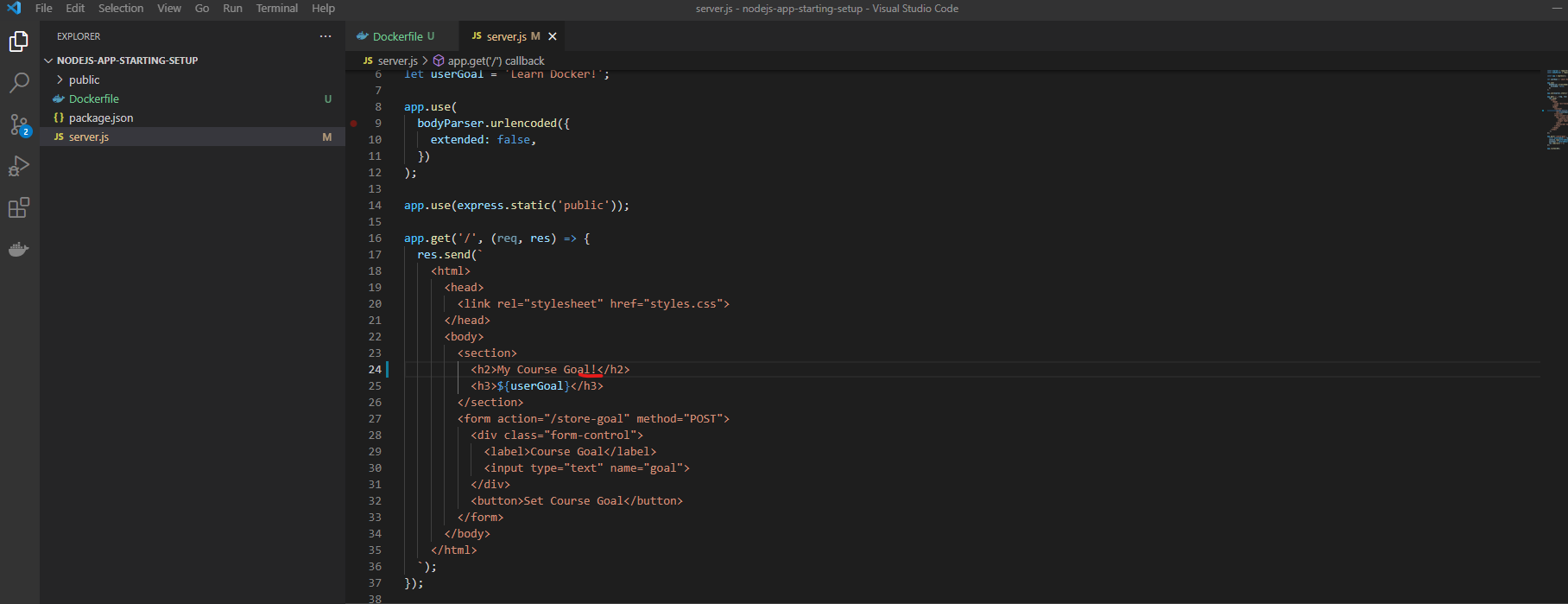


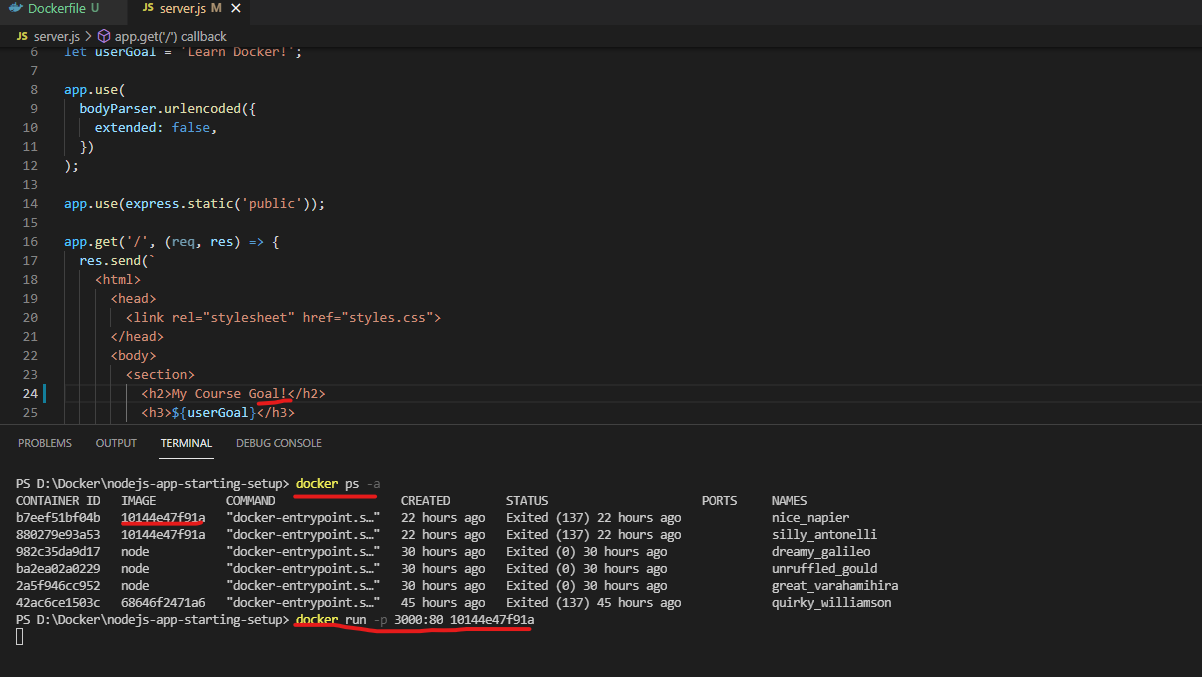


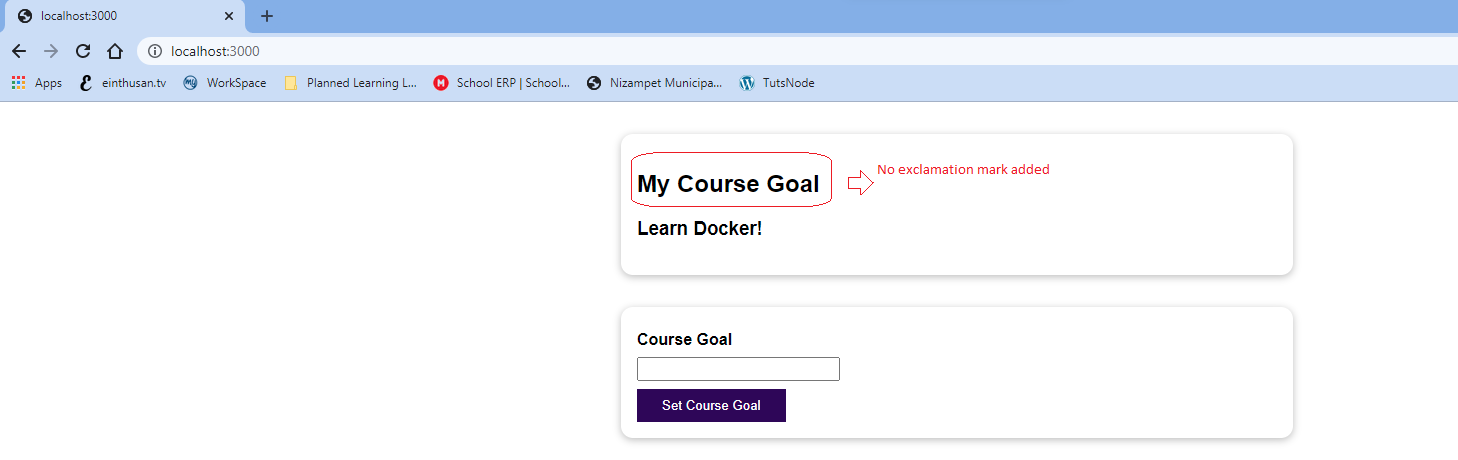


Now we need to understand the images of the docker are read-only, which means once after you built the image by using “docker build” command it freezes the code on the image. Even though you make the changes on your code and start the docker container by using the command “docker run <image ID>” the changes won’t reflect. In order to reflect the changes you need to re-built the code which it creates a new image with the updated code.

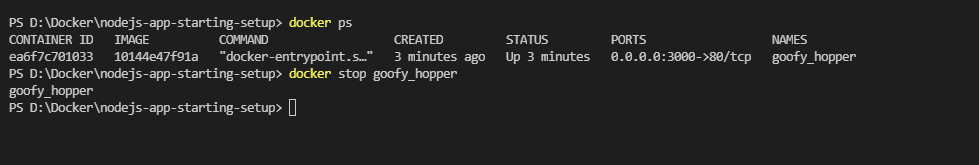
Ex: I made a change by adding exclamation mark under the heading 2 on the server.js file, but while you run the node the changes won’t reflect.

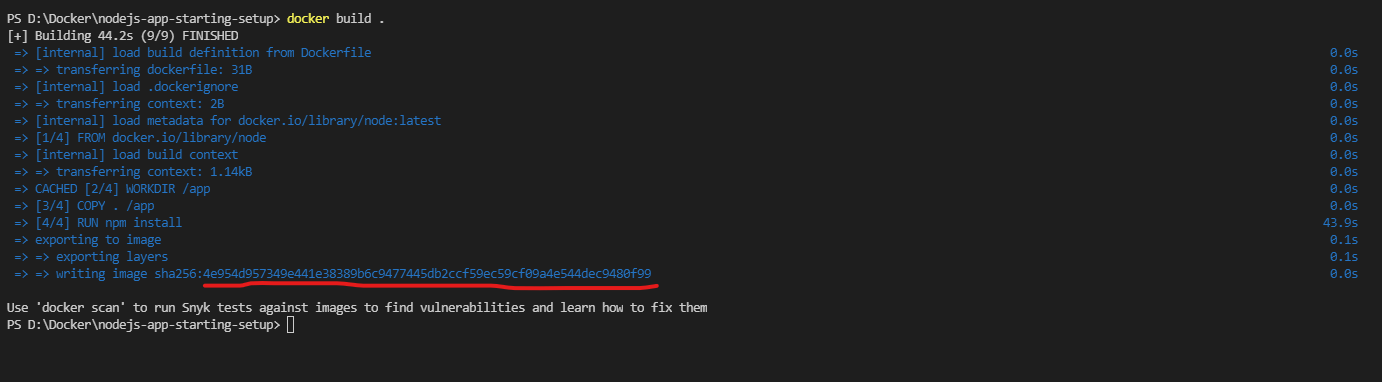


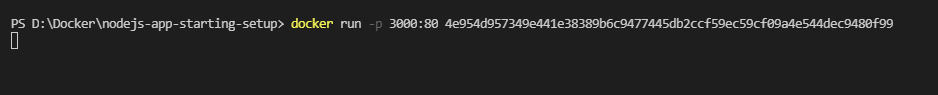


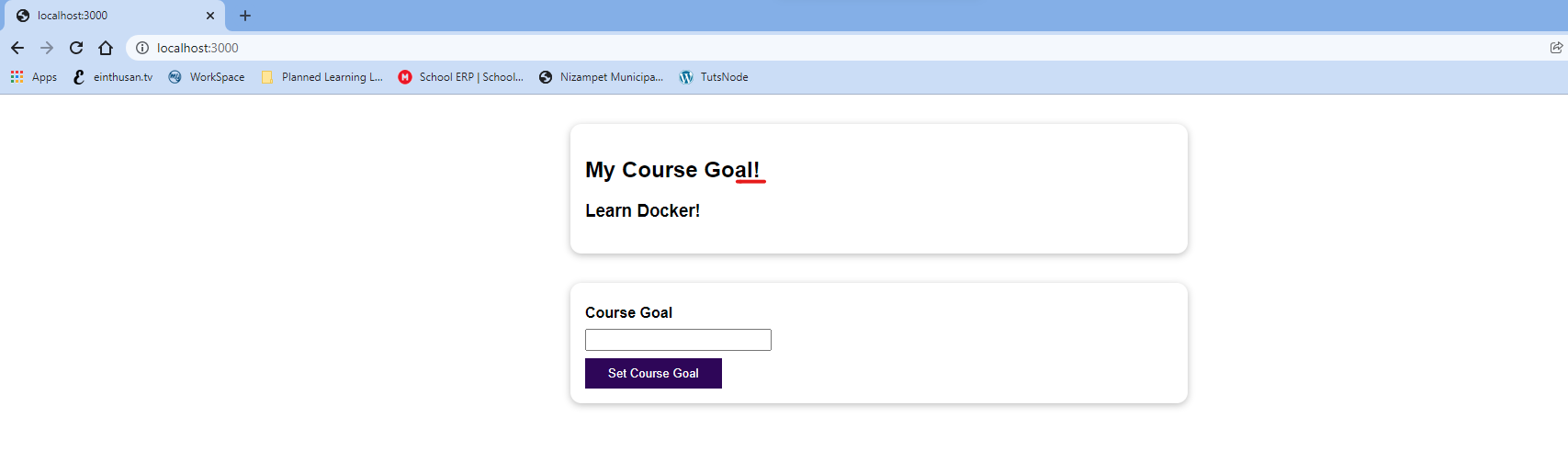


Now In order to reflect the change, we need to re-build it again.









So the image is closed once it is builted.

Docker follows layer based approach, each line that you mentioned on the Dockerfile is a layer, and additionally once you built the images it will create a read only image. Building a container, by using that read-only image will also be an additional layer.

Container Layer (read-write)

Container

Instruction #3: Image Layer 3

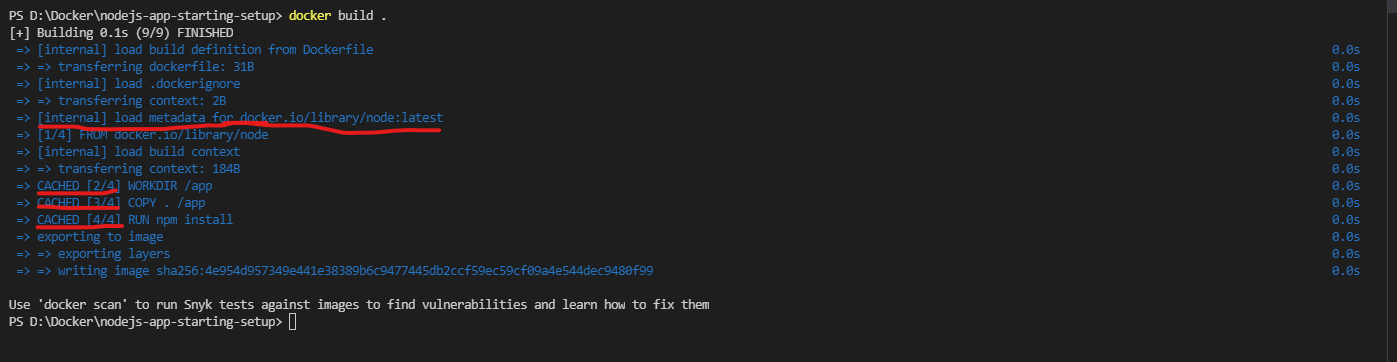
Image

Read-only

Instruction #2: Image Layer 2

Instruction #1: Image Layer 1

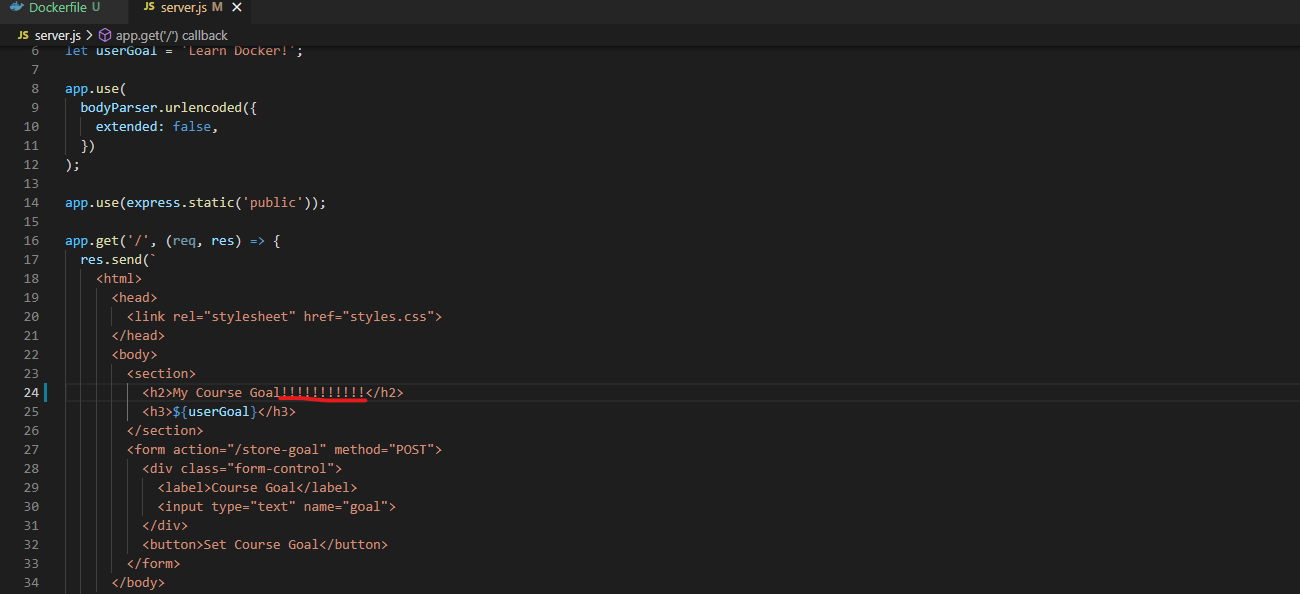
Suppose if you doesn’t make changes of your code and on your Dockerfile, and try to build the image again, it will build the image very quickly because docker knows on each layer there was no change.

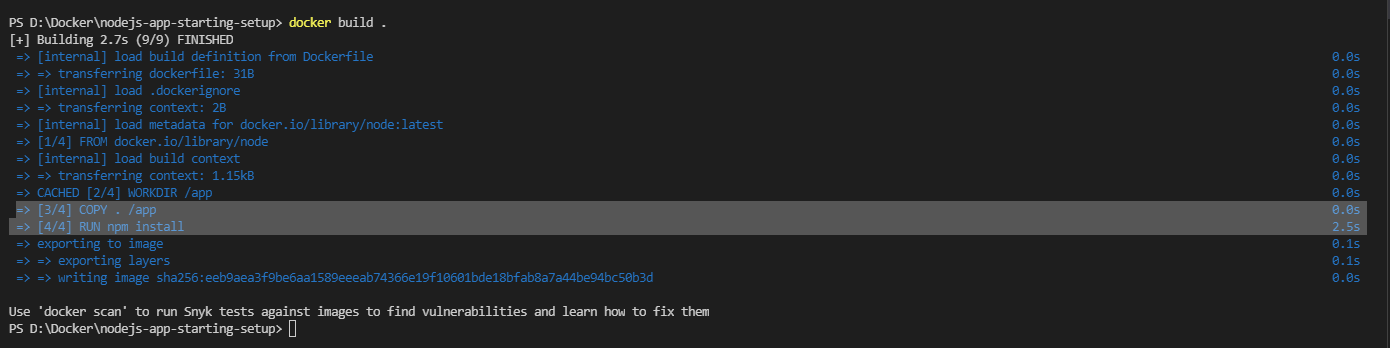


If you observe on above screenshot, the base image is loaded from internal cache not from dockerhub. Also the executing the layers mentioned below are from cache as they don’t have any changes.



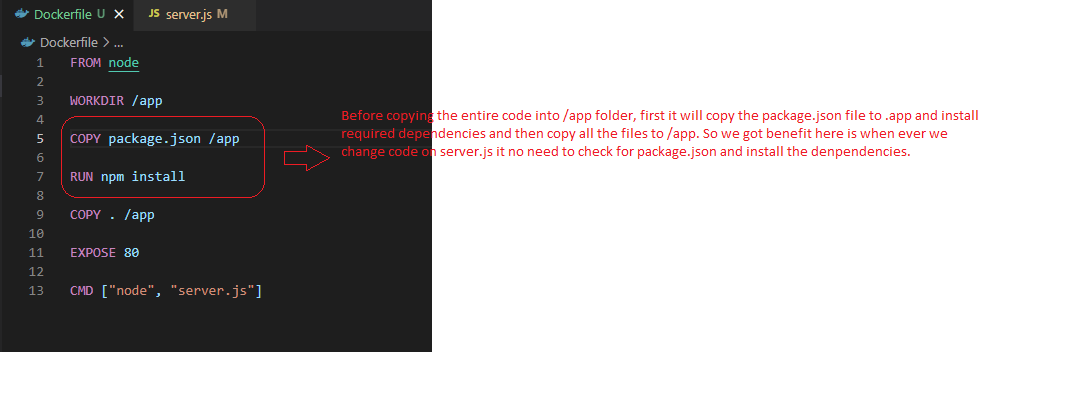
Suppose if you make any changes of the on server.js file and try to rebuilt it.



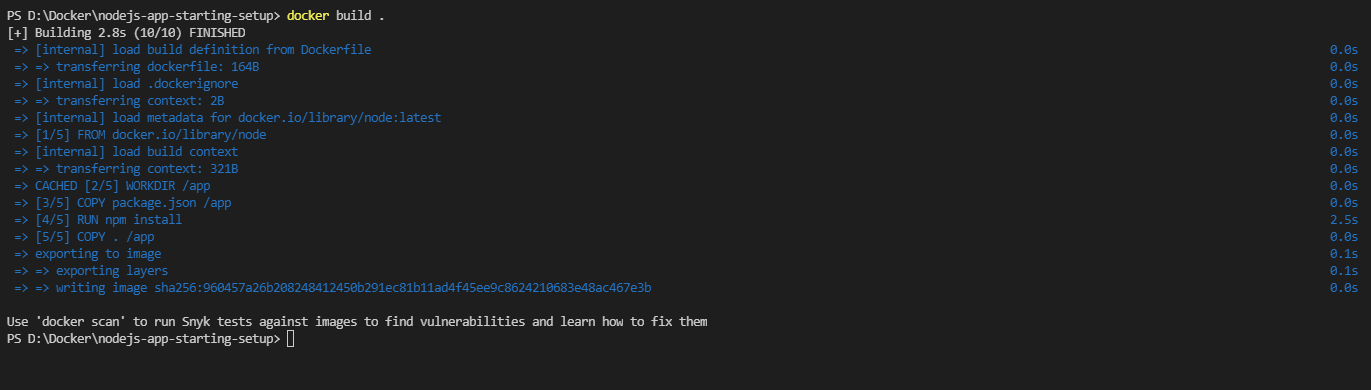


As you can while building the code other layers got completed very fast through cache but the layer RUN npm install will took some time as it recognizes the change in the code. Here it need to compile the server.js and need to check and install the dependencies again.

So to optimize the code lets re-write the code on the Docker file as shown below.



So in feature it won’t run npm install command, if we make any changes on server.js file. So we need to understand that docker follows layer based approach.



Please remember that container is an extra top layer of your image which contains code and run time environment. A container can’t copy the code from the image, instead it will create a run time (memory, cpu etc..) environment based on the code residing on image.

## Managing Images and Containers

$ docker --help 🡪 To see all the options available with the command docker.

Add –help to see all options

Containers

Images

Can be **named** --name

Can be tagged (named) -t, docker tag …

Can be **configured in detail** see --help

Can be **listed** docker images

Can be **listed** docker ps

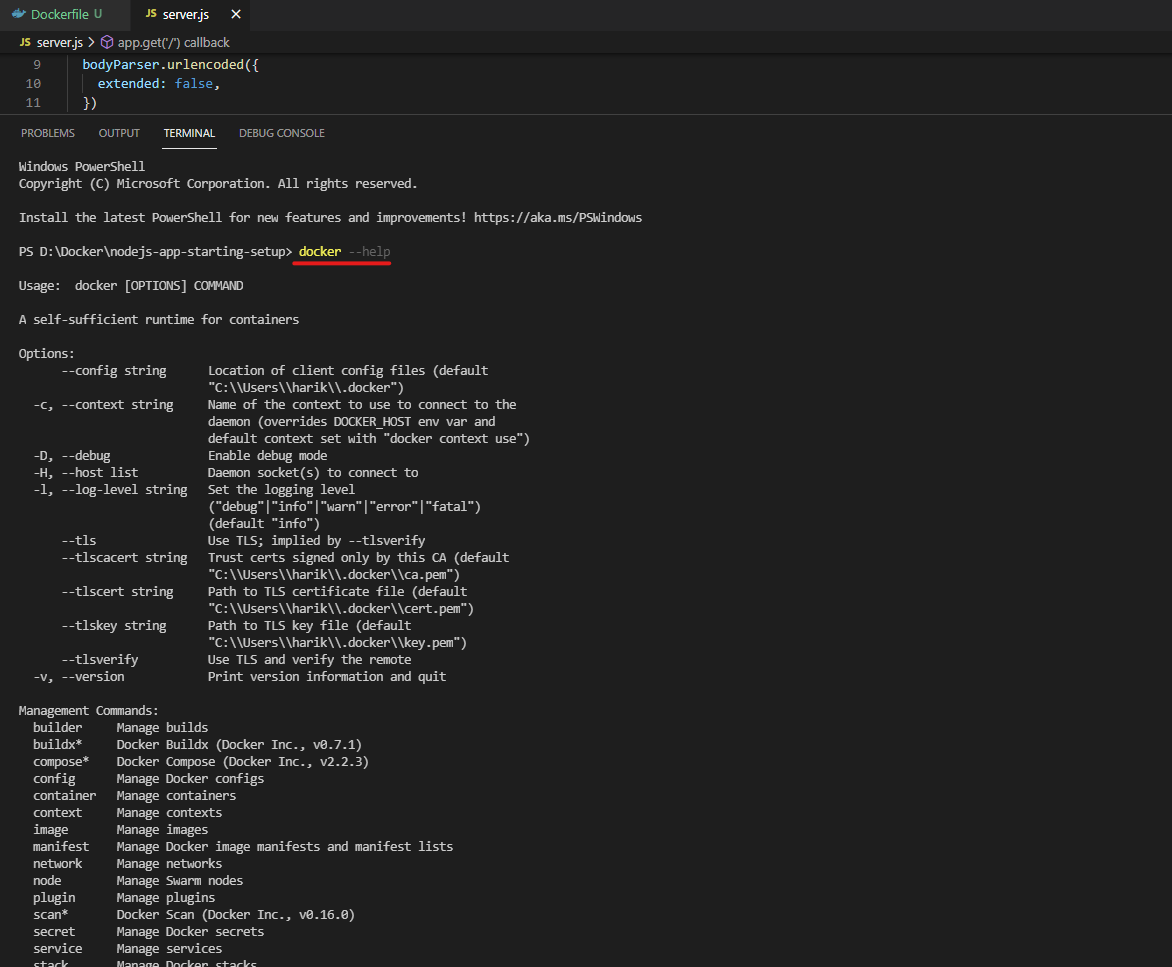
Can be **analysed** docker image inspect

Can be **removed** docker rm

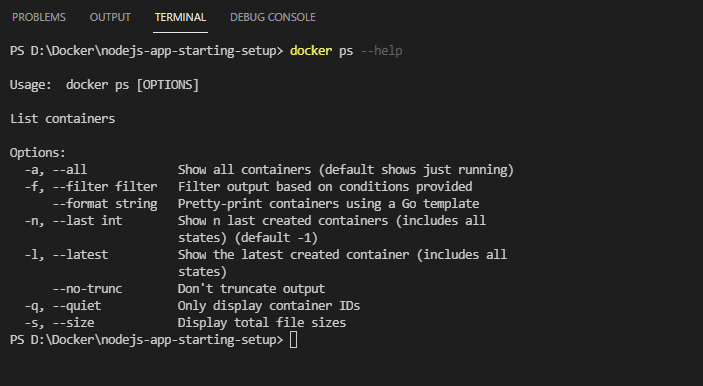
Can be **removed** docker rmi, docker prune

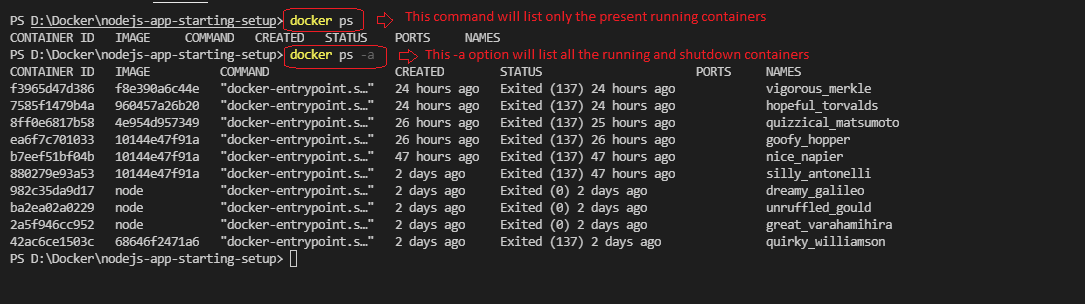
## Stopping and Restarting Containers

$ docker --help 🡪 will list all the help with respect to the container.



If you want any help related to any specific command



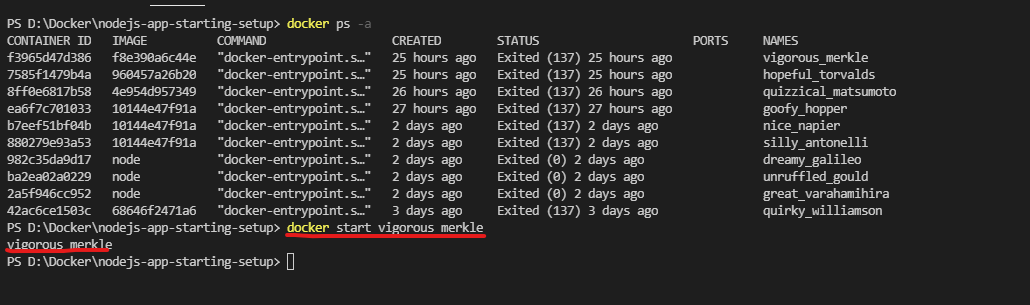


As we know up to now we had started and run the containers by using the command “docker run <image ID | Name>”. But while docker run command it will create a new container instead of starting existing container, so for each run of docker run command it will create a new container.

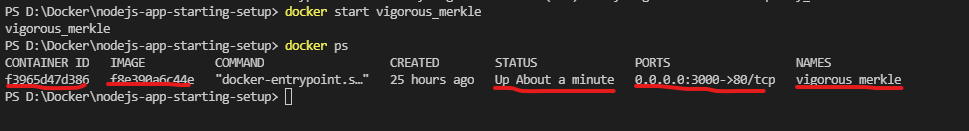
So if you want to start and stop the existing created containers, there are options for the docker command.

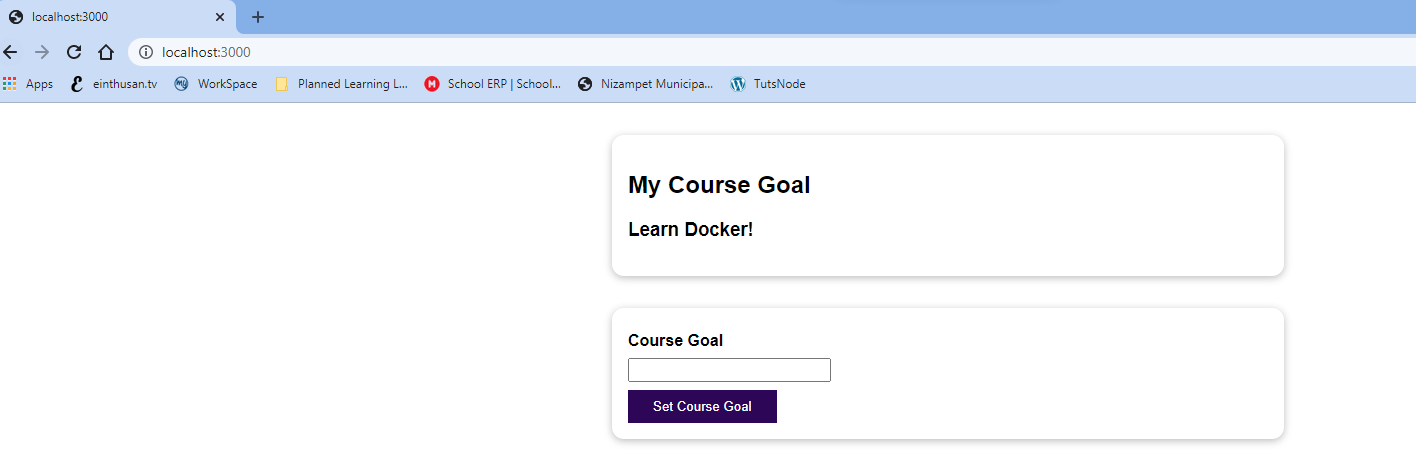
$ docker start <Image ID | Name>

$ docker stop <Image ID | Name>

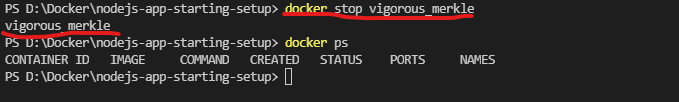


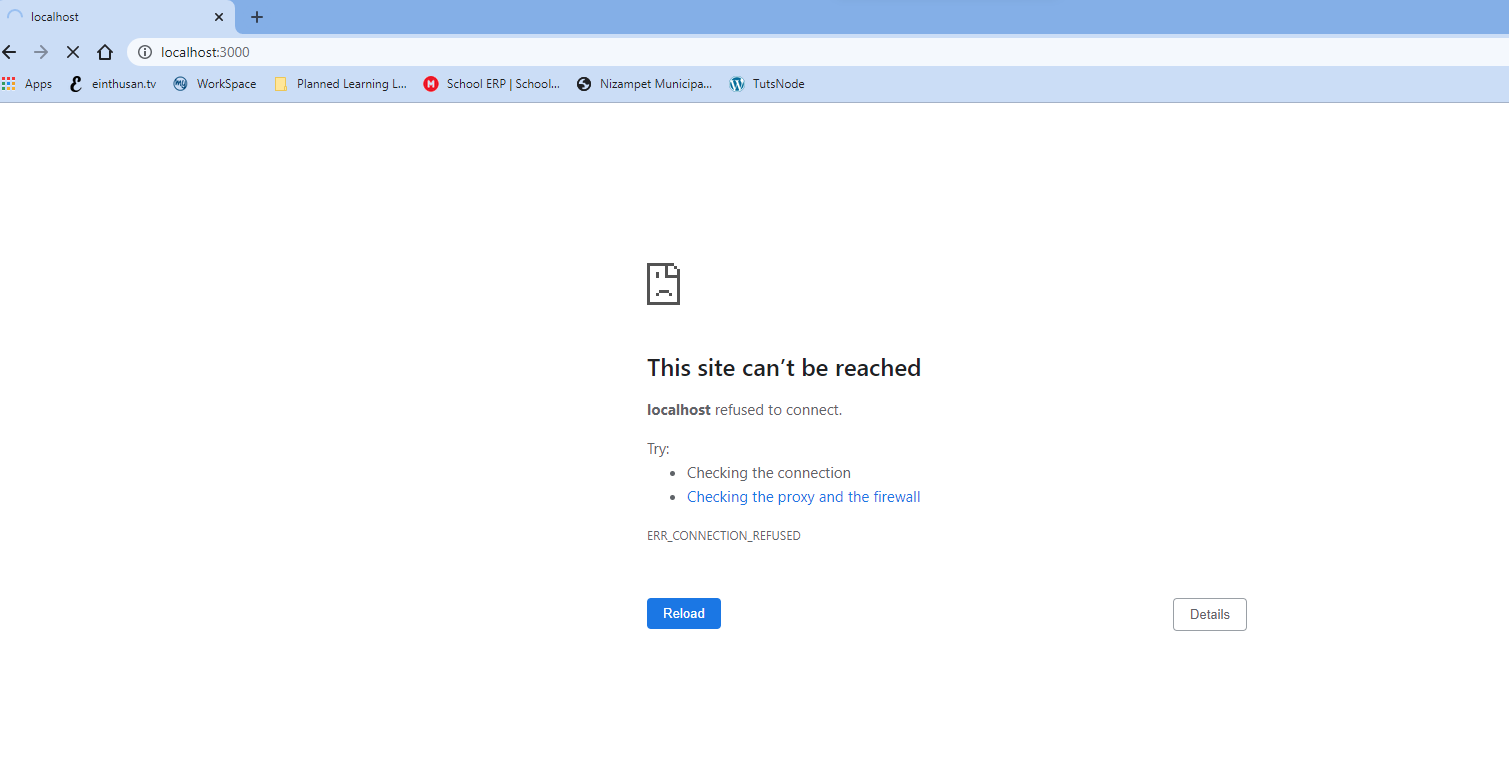
As you can observe that while running the docker start command, it had started the container and exited. But on background the image is running, also we did mention -p option as the container was configured with the option.





If you want to shutdown the container, use the command docker stop.

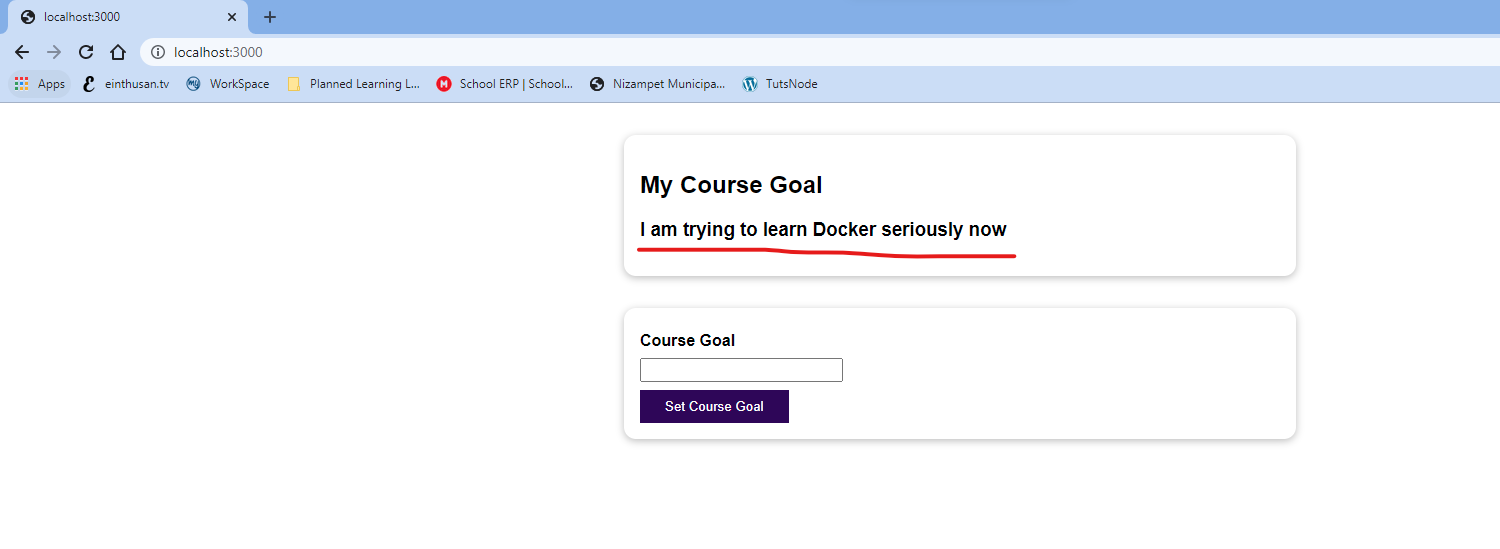


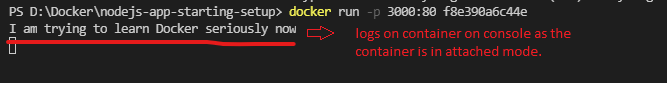


## Understanding Attached & detached containers

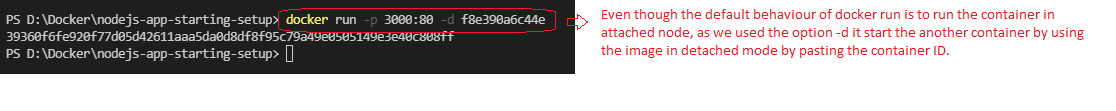
By default when run the command “docker run <image ID>”, it will start the container on attached mode means it won’t exit from the terminal and it will show the output on the console whatever we are logging.



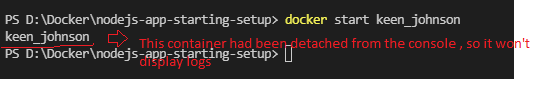




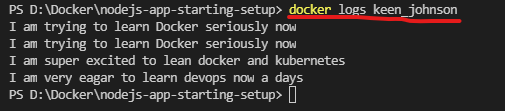
You can also have an option “-d “ to docker run command to change the default behaviour to run in detached mode.



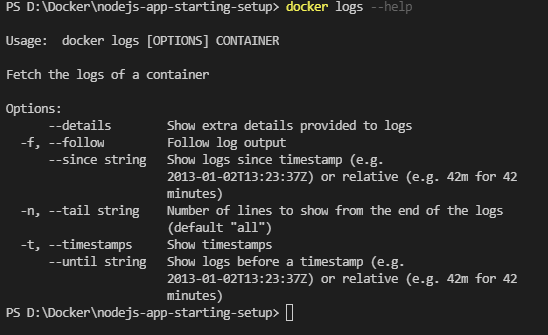
Same way when we run the command “docker start <Image ID”, by default it will start the container in detached mode, means it give you the terminal back to you.



So if you want to show the logs from the detached container we have an another way to issue the command “docker logs <image ID| name>”.

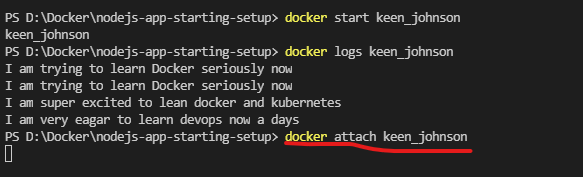


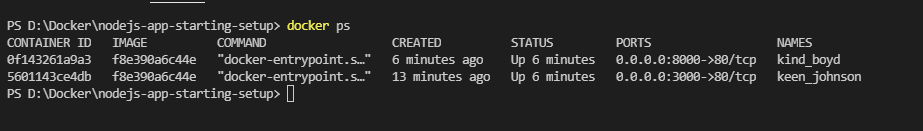
We have another option to -f to docker logs command, which will display the logs continuously. By using this option it will attach the container again as it keeps listening the logs for the container.



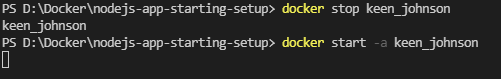


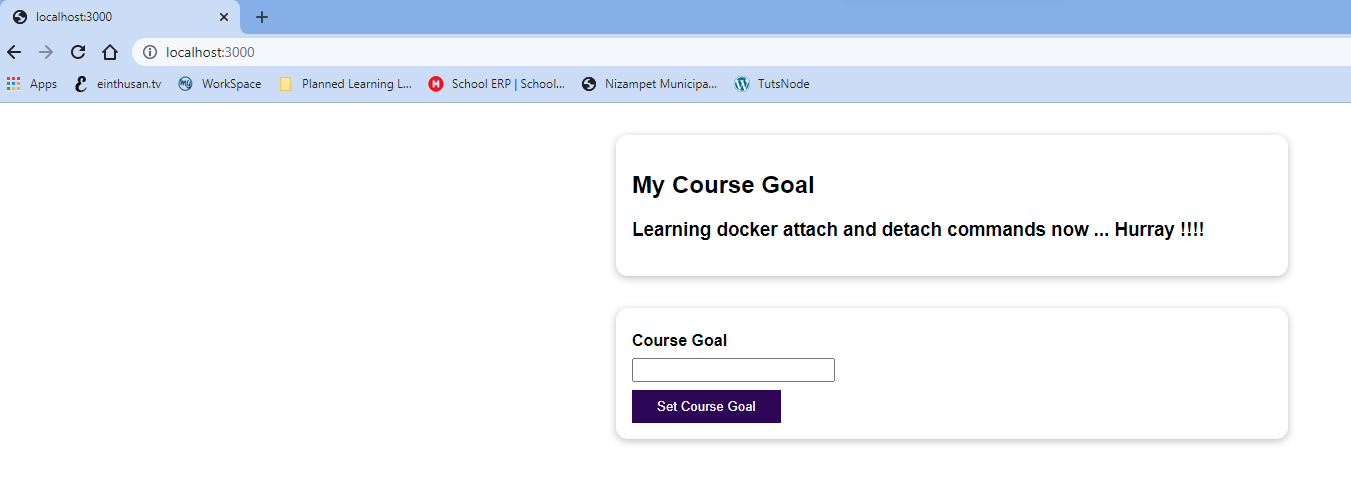
Suppose if you want to manually attach the detached container, we have an option   
“docker attach <Name | Image ID>”





By default docker start command will start the container in detached mode, if you want to use the same option in attached mode you can use the command “docker start -a <image ID| name>”.







## Docker Interactive mode

Up to now we had always used the NodeJS application as an example for running it as a docker container, now we will write a python code which will accept some input and execute the code.

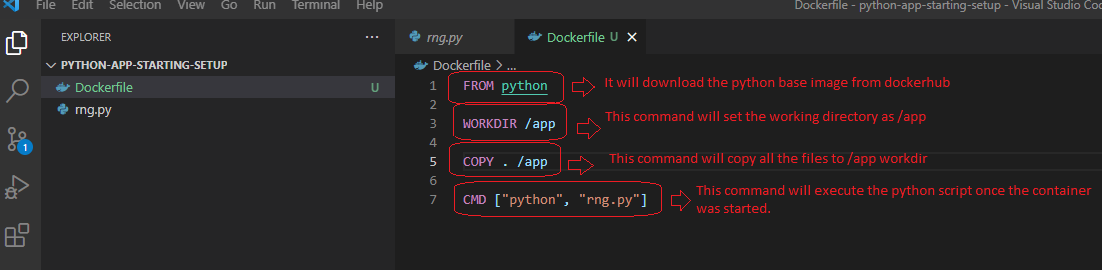
So while running the docker we need an interactive shell which will you enter an input.

Below is the code that we are trying to run it, you can download the code from the GitHub as shown below.

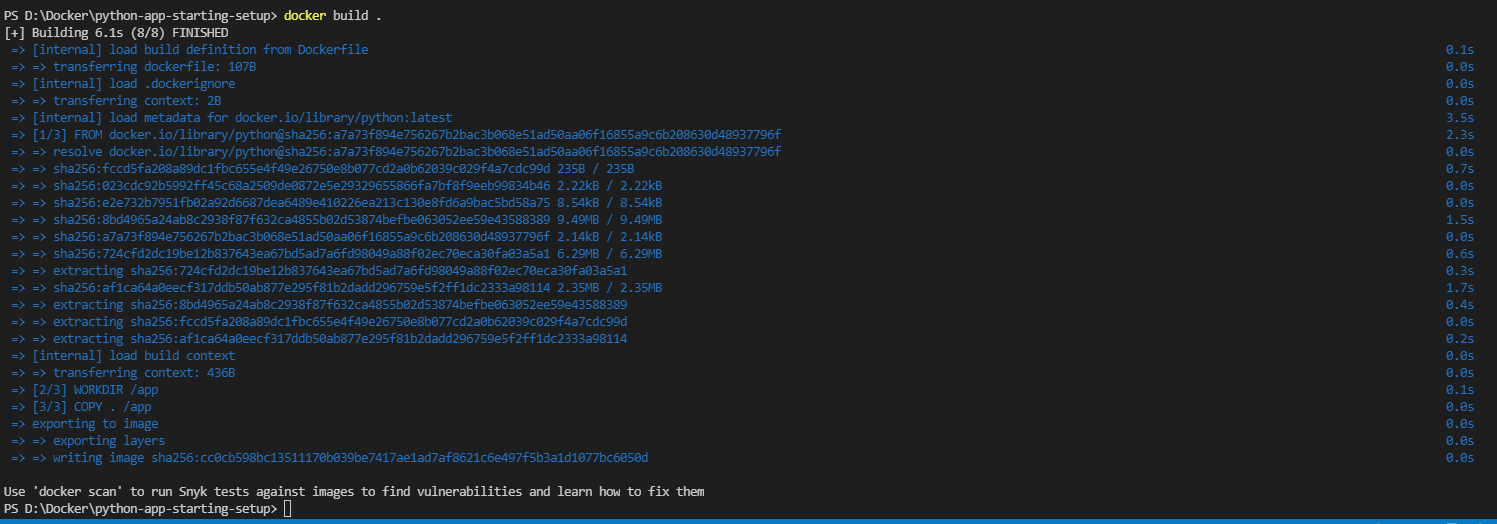
<https://github.com/harikrishna83/Docker-firstdemo-handson/blob/master/python-app-starting-setup/rng.py>

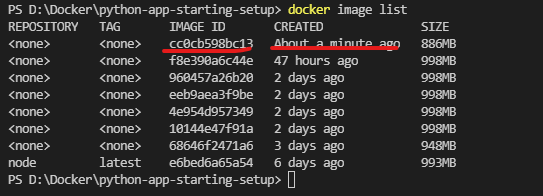
This python code will accept Min and Max number as input and calculate a random integer number and provide you as output. So this app will need interactive input to run.

Now Let’s write a Dockerfile in order to start this python code in a container.



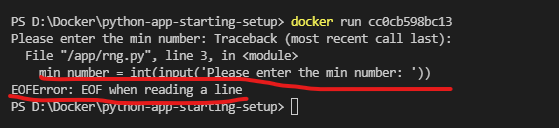
Now in order to create this code as docker image, let’s build it with the command “docker build .”





Now let’s run the docker command “docker run <image ID”.

Note: Here we are not exposing any port as it only runs simple python code.



As you can see it needs an input for the script, as we are not providing any interactive input on the shell it abruptly exited.

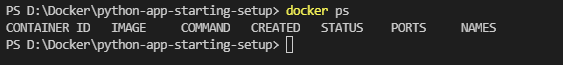


As we need an interactive shell to enter the input, we will run the container with the options -i -t

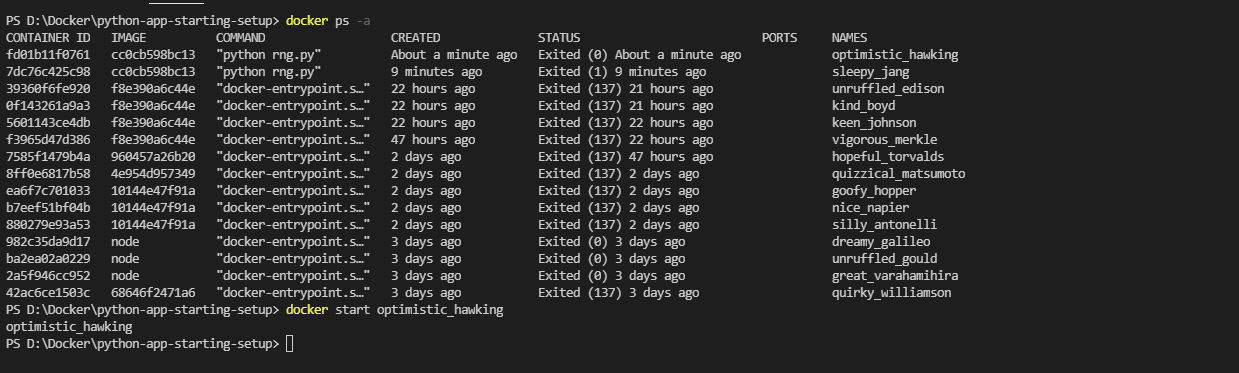
-i, --interactive Keep STDIN open even if not attached

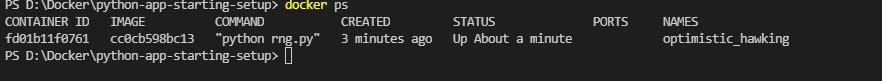
-t, --tty Allocate a pseudo-TTY





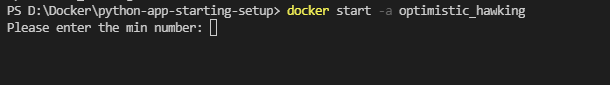
Now if you suppose, you start the container with the command “docker start <image ID | name?”





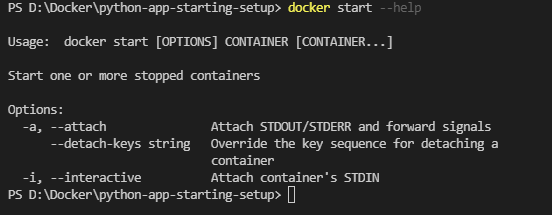
As you seen above, we can’t able to enter any input as it started in detached mode.

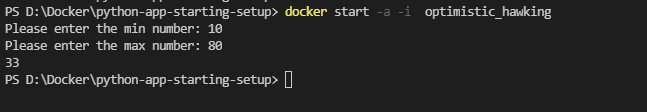
To deal this, let’s start this container in attached mode

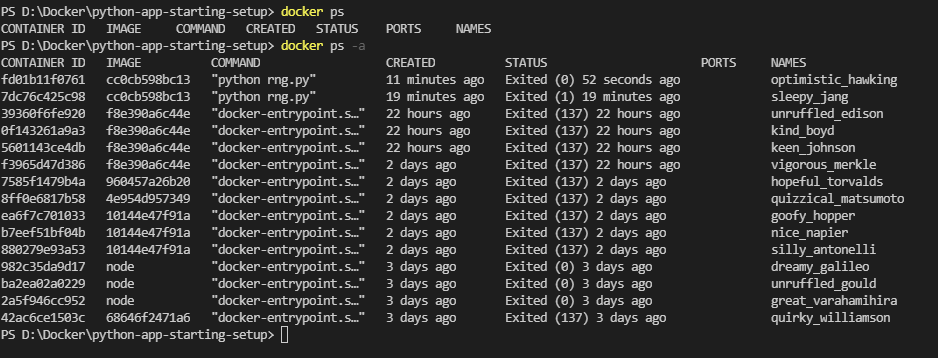


As you can see from, it started in attached and asked for input, after entering input it behaves strangely as it doesn’t have interactive shell and it doesn’t go further and asked for the next input.

Now let’s start this container with the option -a -i as show below





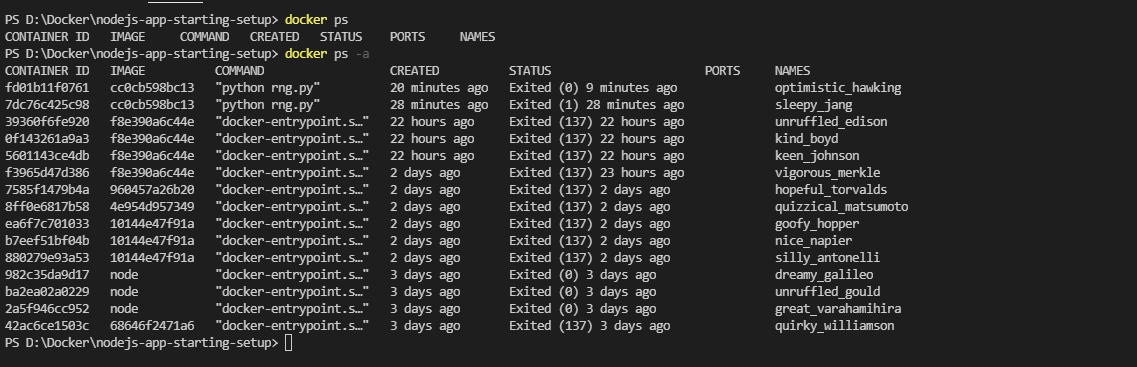


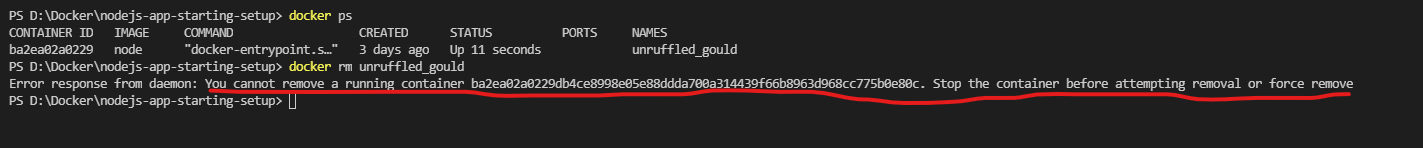
## Docker Deleting Images & Containers

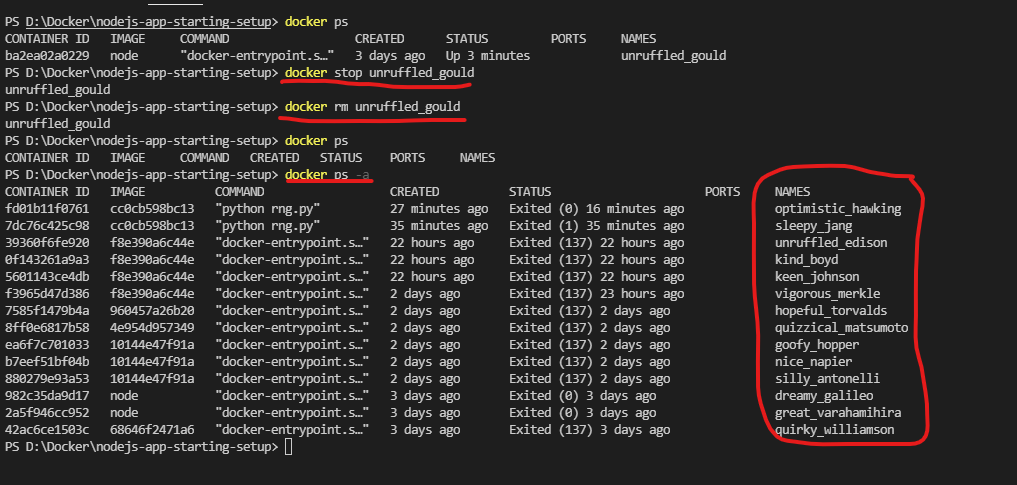
As of now, we had created multiple containers which was created previously and was not using them. Those containers had occupied some resources on your system if they start.

Note: You can’t able to remove any running containers, it will throw an error to first stop the container and then remove it.

You can delete a docker container using the command “docker rm <container name | ID>”.



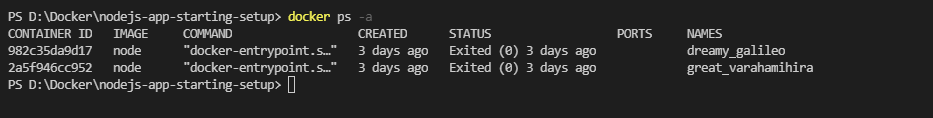




Now if you want to remove multiple containers, provide all the containers with the space to docker rm command.



Now I have only the containers which had the base image downloaded from dockerhub.



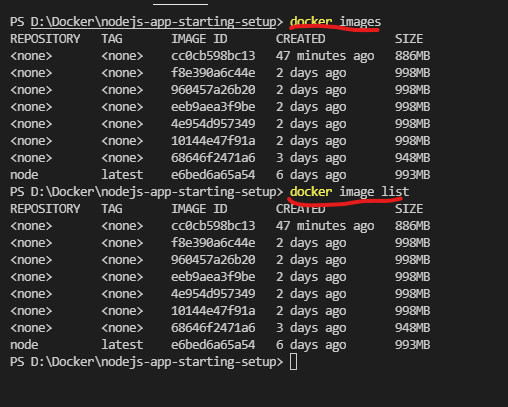
Note: You can run a command “docker container prune” to remove all stopped containers at once.

Now let’s see how we can delete the images that was created with the docker build command and which are not using as of now.

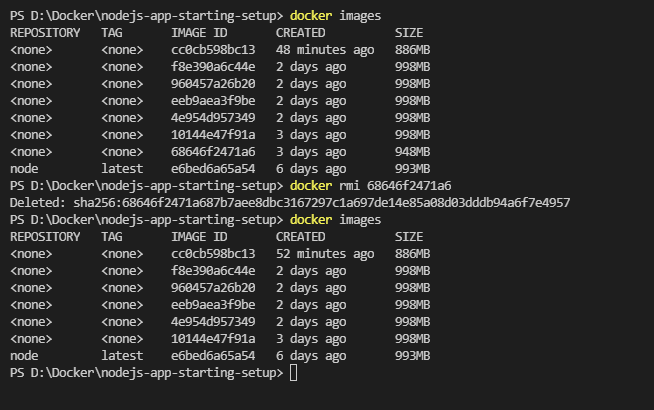
You can list all the images that with any of the following command.

$ docker images

$ docker image list

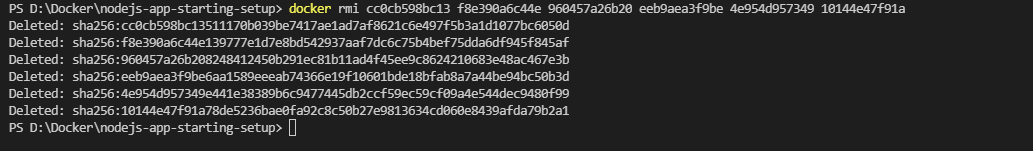


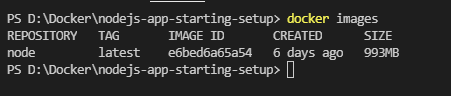
You can use the command “docker rmi <image Name|ID>” to remove the image.



Note: If the image was using by any running or stopped container, it won’t delete. You need to stop the container and remove that container first in order to remove this image.

If you want to remove multiple images, provide those with spaces to the command “docker rmi “





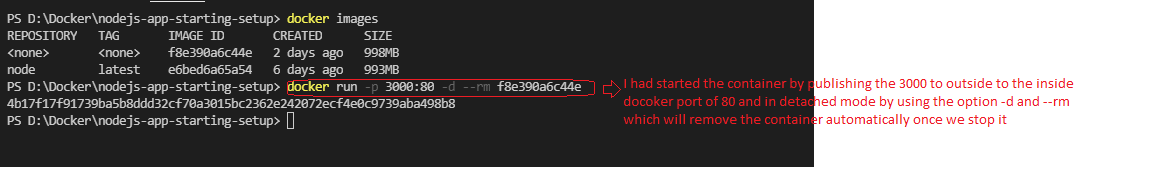
Note: You remove all the un-used images at a time, by using the command “docker image prune”

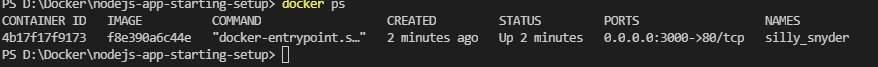
## Docker Removing stopped containers automatically

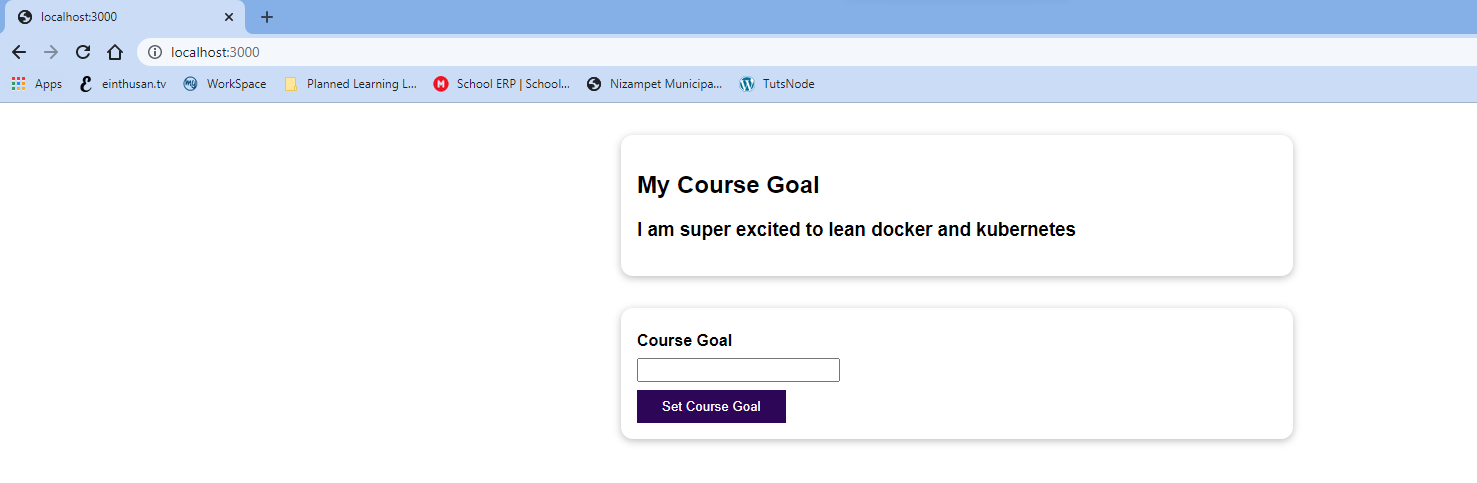
If you check the help docker run command, you will find an option --rm, which describes it will remove the container once it stopped automatically.

$ docker run --help

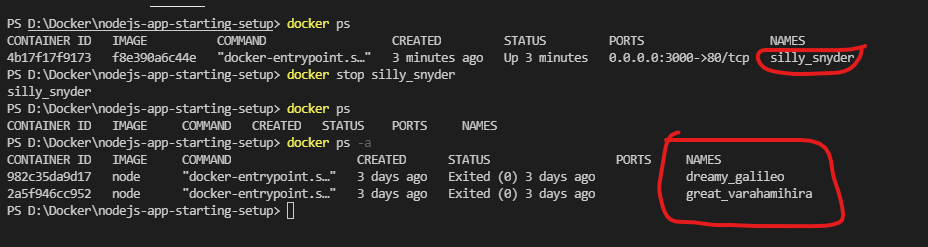








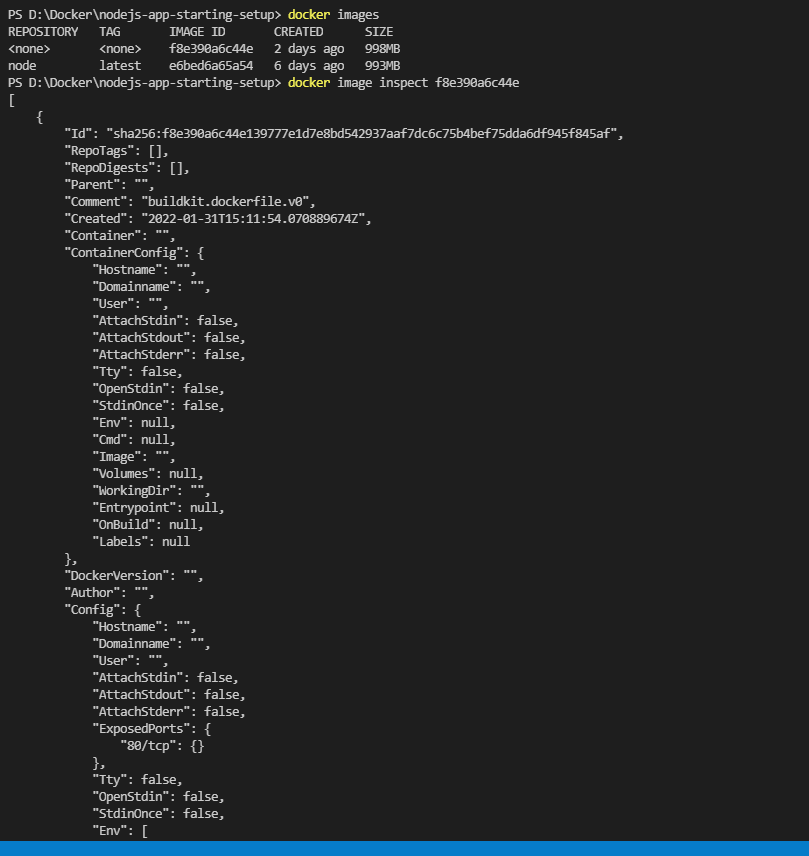
Now let’s stop the container and see whether this container got deleted automatically or not.

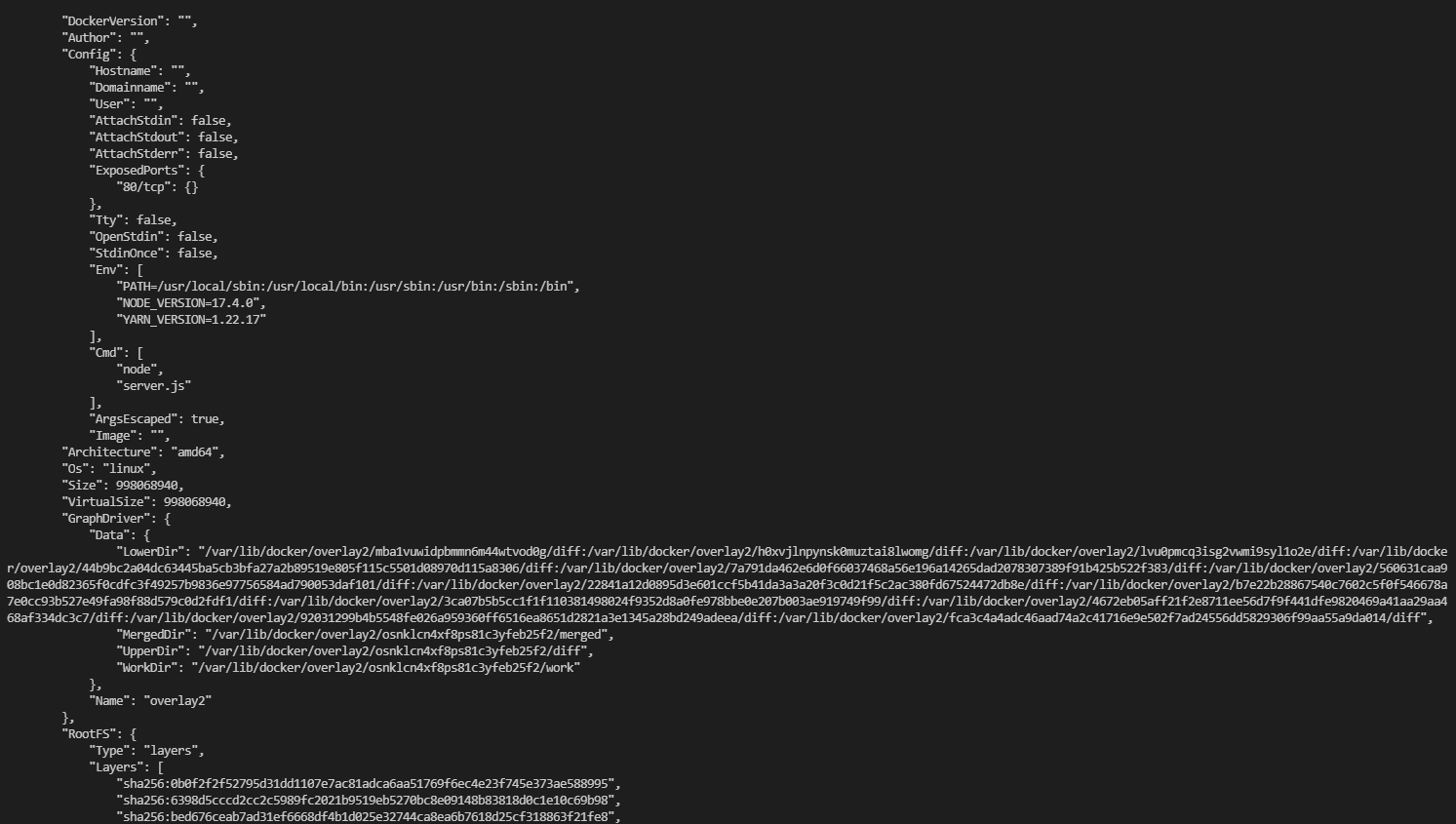


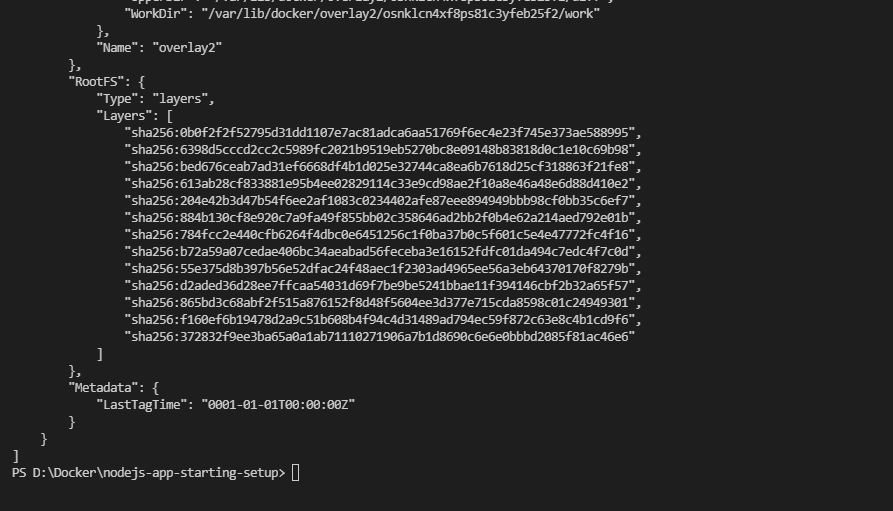
## Docker Image Inspect

You can inspect a particular Image by using the command “docker image inspect <Image iD | Name>

This command will you a detailed output like which OS versions that image contains, what are layers presented on that image.



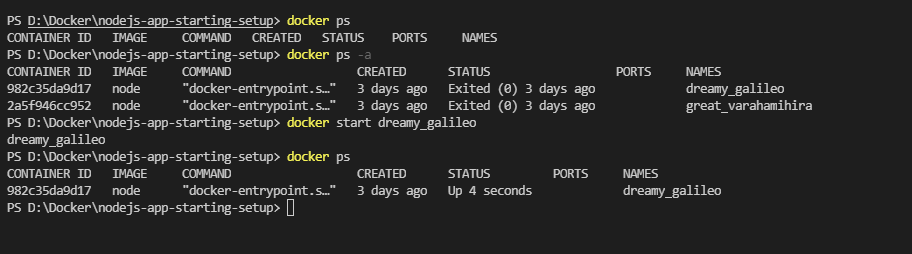




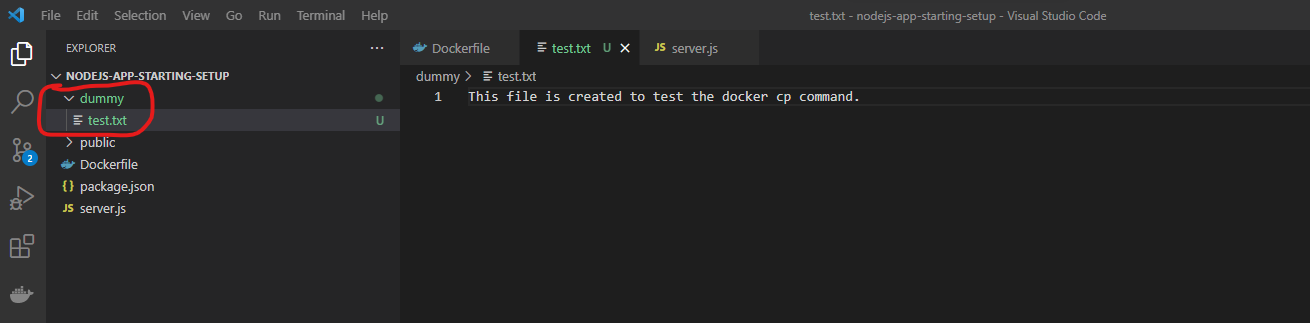
## Docker Copying files into & From a Container

We have a command to copy the files to the running and viceversa using the command “docker cp”.

Now let’s start a container and create a dummy folder and text in your local machine and will copy those to a running container.

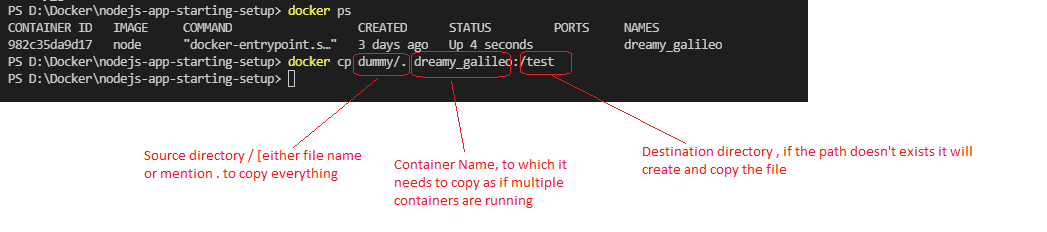


Now I will create a dummy folder and text file inside my local project directory and will copy the files to a container.

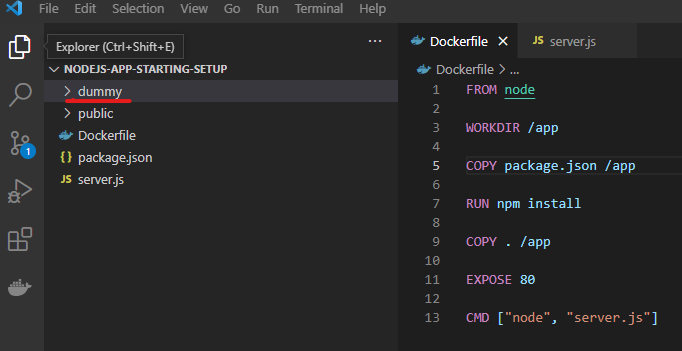


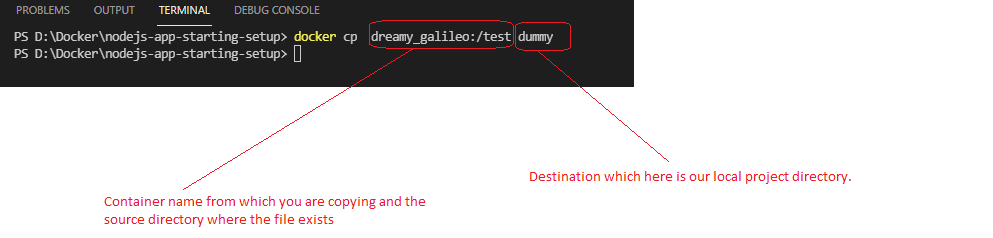
Now we can copy this file to container by using the below command

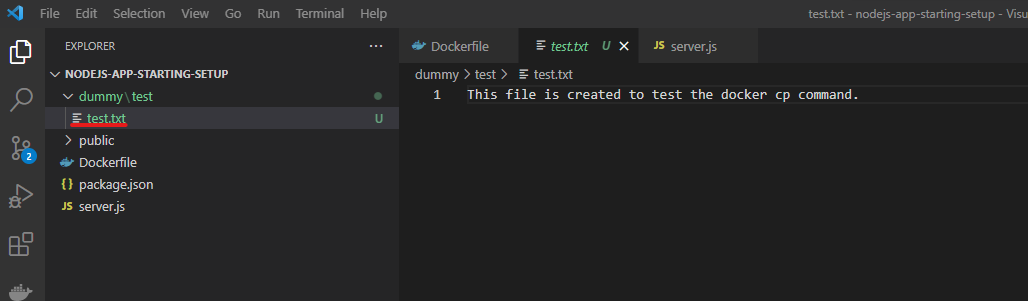
$ docker cp dummy/[test.txt|.] dreamy\_galileo:/test



Now remove the file from your local directory and copy the file from the container to your local directory.

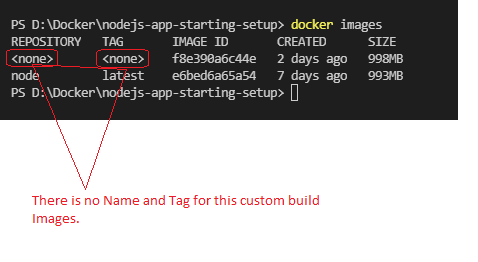


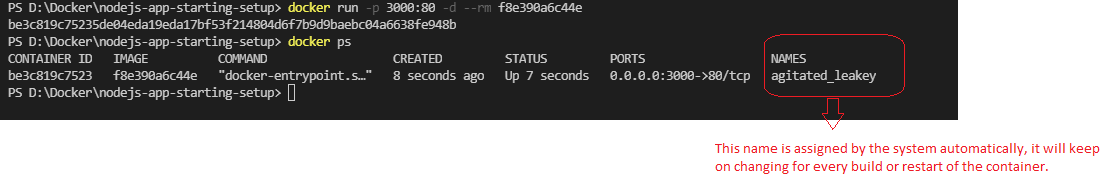




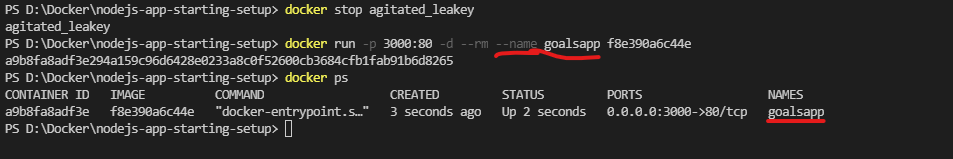
## Docker Naming & Tagging Containers & Images

Up to now we didn’t named our containers & names specifically, the system assigned its own name for containers. Now we will assign a specific name to the containers and images for identification purpose and start the containers.

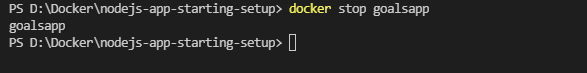




Now stop this container and assign my own name to the container.



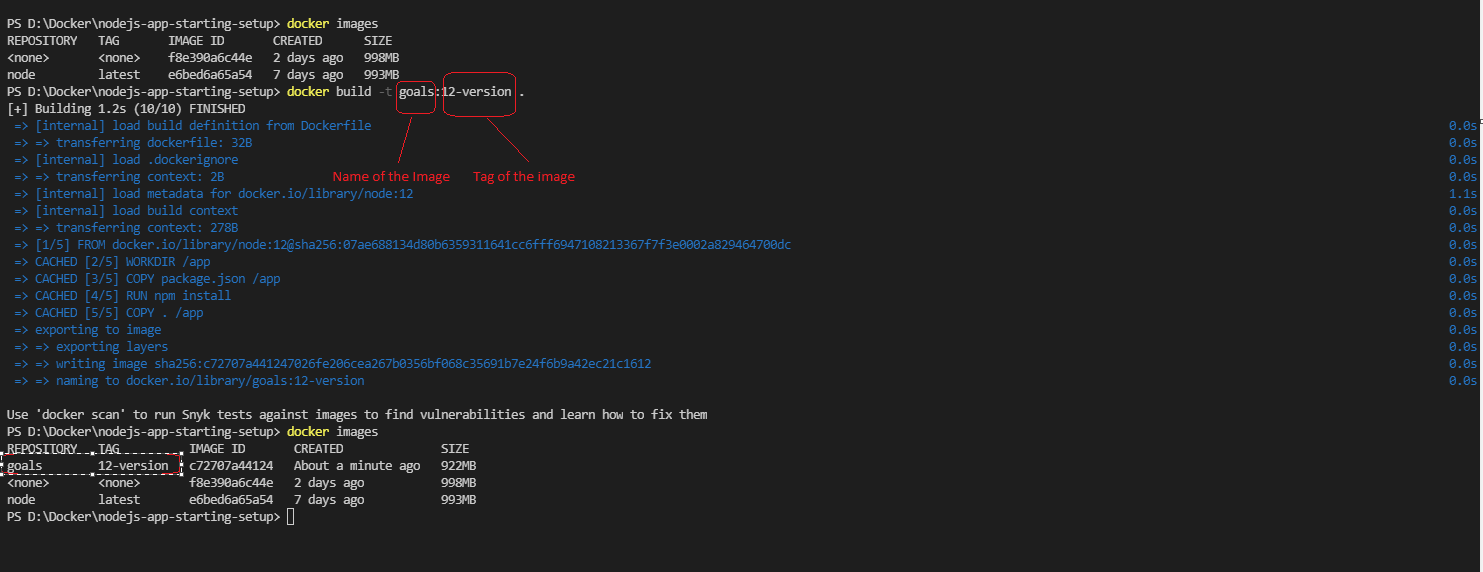
If you observe above I had set a name manually by using the option --name <name> while running a container. I can stop and start this container using that name



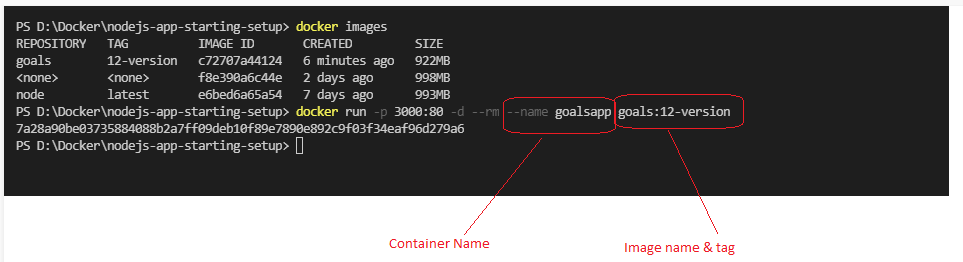
Now a similar concept exists for images, but with a slightly different pattern. Naming Images consists two parts [ <name> : <tag>]. A name is defined as group of possible more specialized images Ex: “node” and tag defines a specialized image within a group of Images Ex:14

Note: A tag can be a word or a number anything you can use.

Note: You can remove all images including tagged images, you need to run “docker image prune –a”.



Now you start the container using that image using the below command



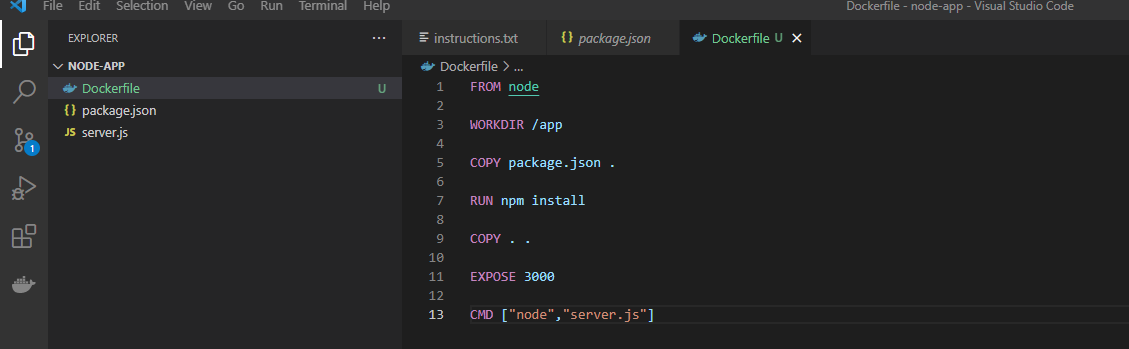


## Assignment

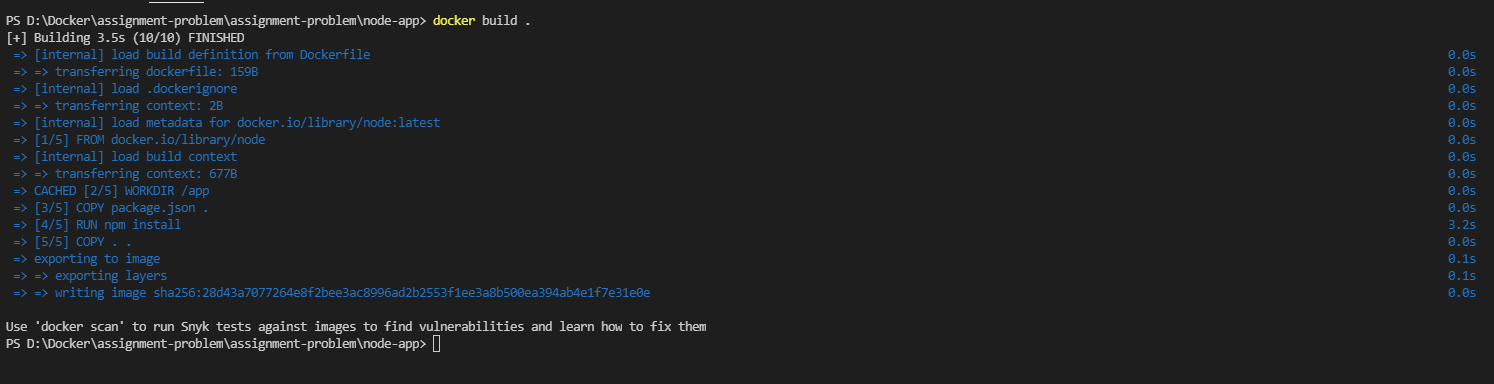
Assignment code and Instructions can be downloaded from the following GitHub.

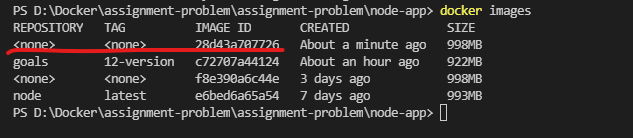
<https://github.com/harikrishna83/Docker-firstdemo-handson/tree/master/assignment-problem>

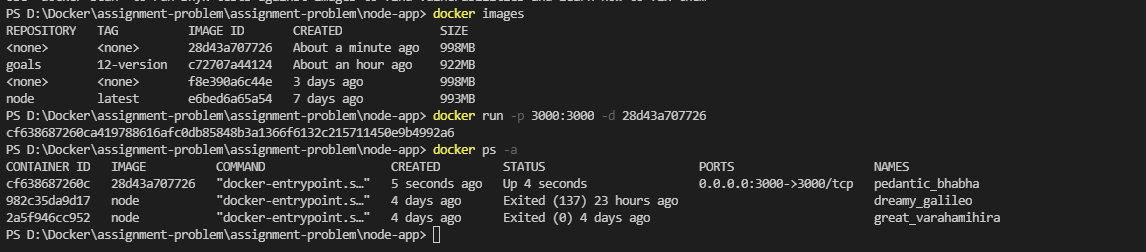
1. Dockerize BOTH apps - the Python and the Node app.
2. Creating Dockerfile node app

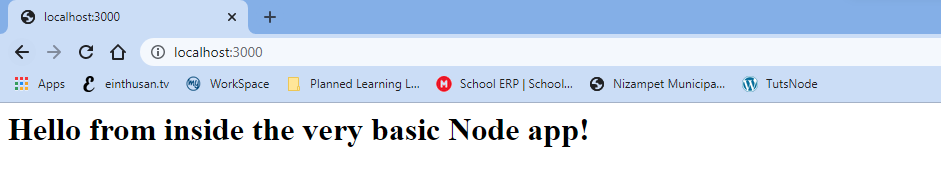


Building the image by using the Dockerfile

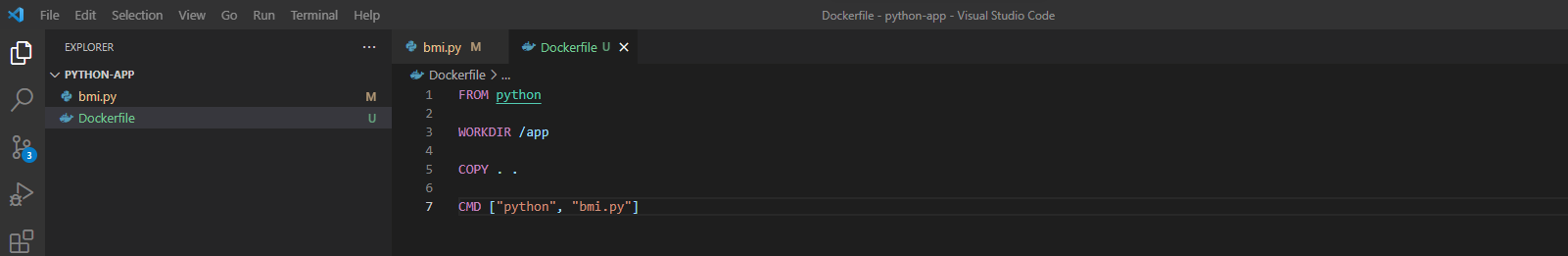




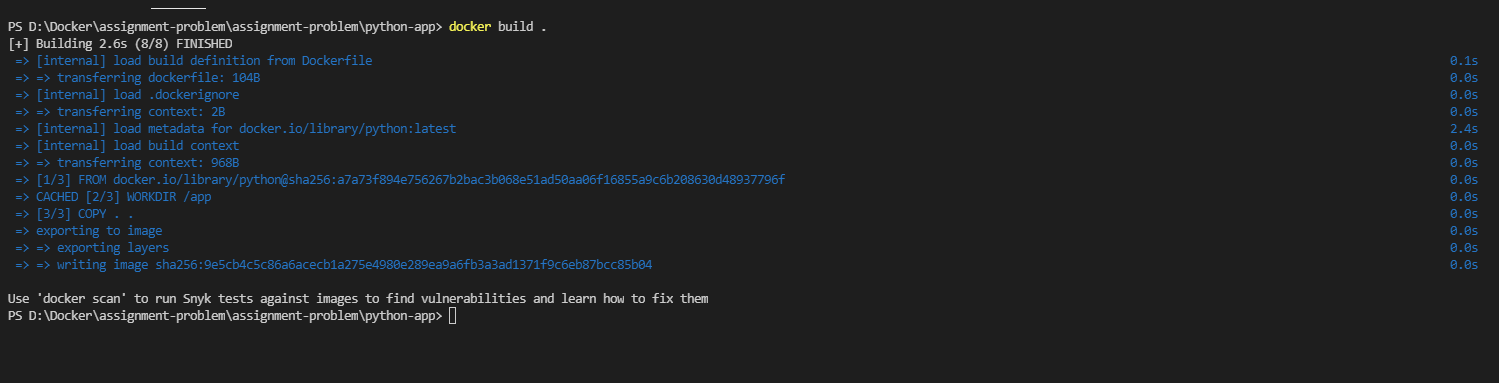


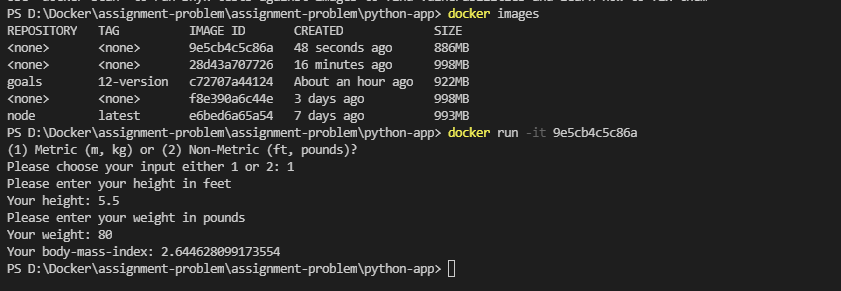


1. Let’s start the dockerize the python app

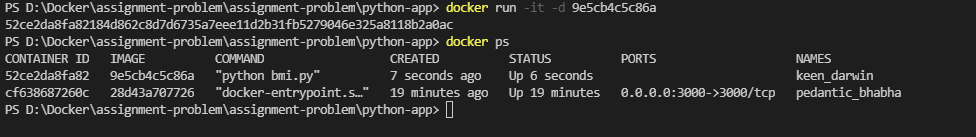


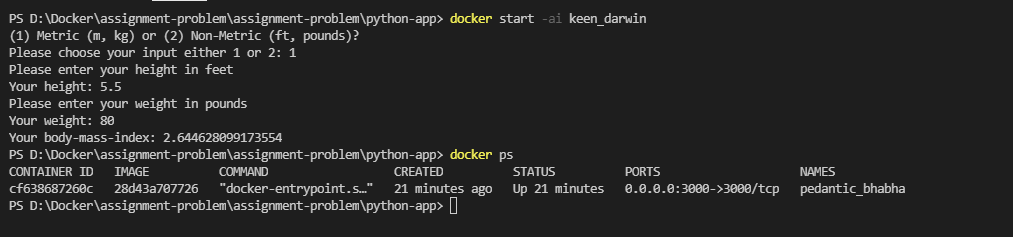
Let’s build a docker image by using this instructions above.





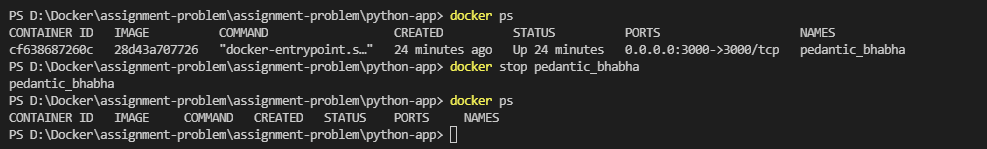
Let’s verify now whether the both applications (NodeJS & python) are containerized.

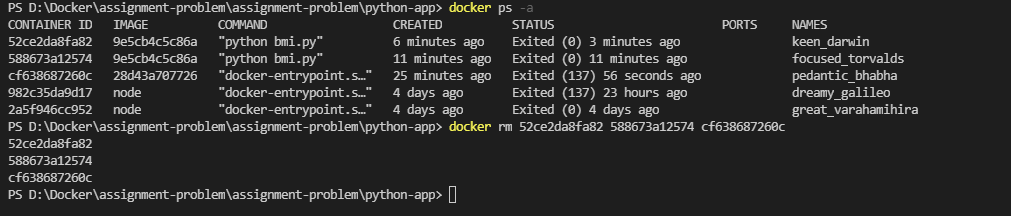


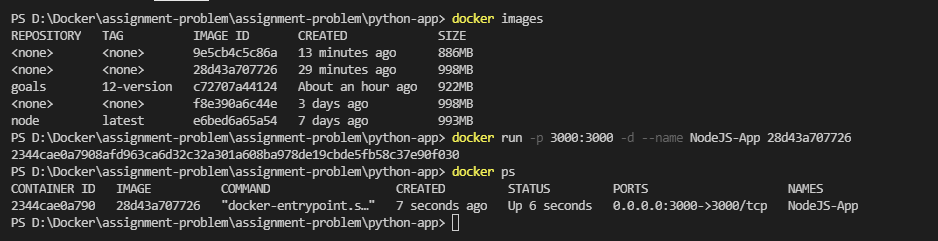


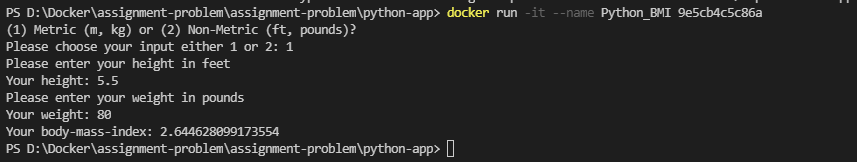
Now first 2 steps creating the Images and running the container got completed.

3 – step Now recreate both containers and assign names to both containers.



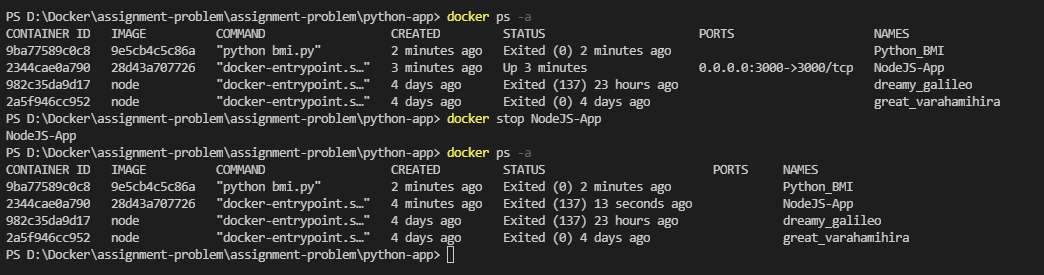




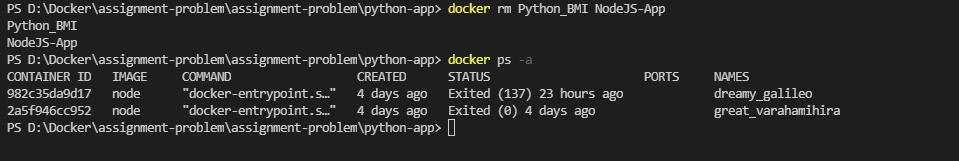




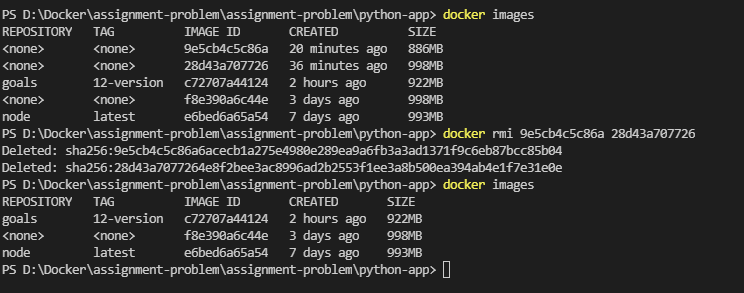
4 – step : Clean up (remove) all stopped (and running) containers,



Removing containers now

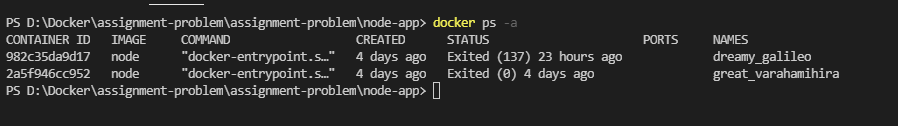


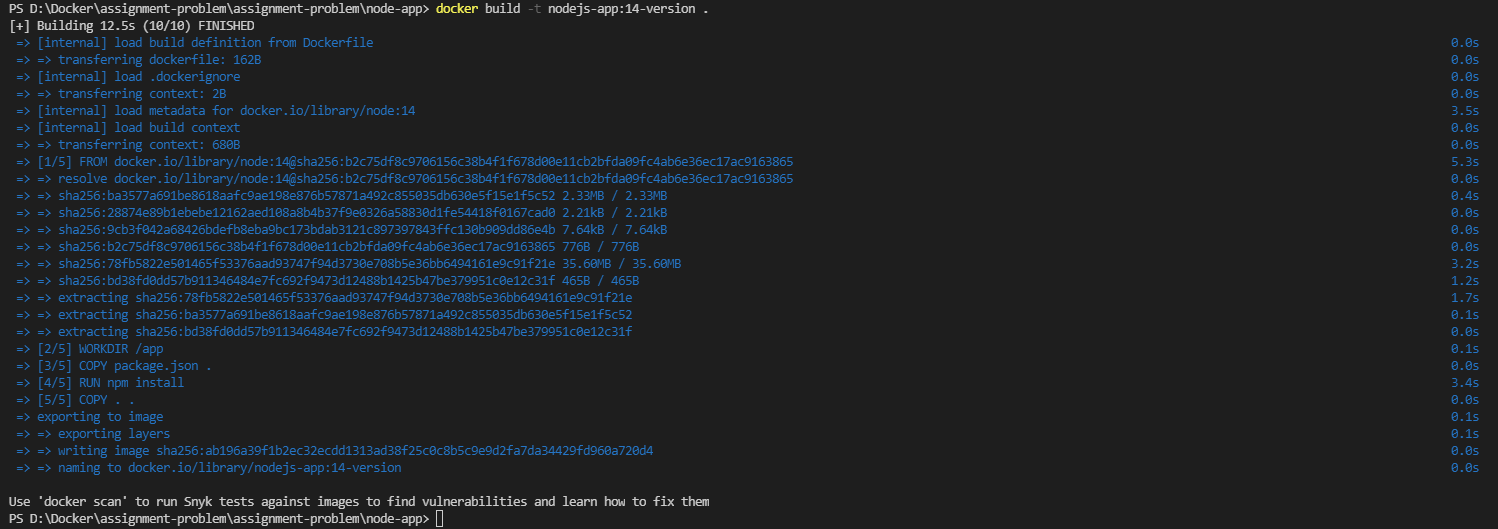
Removing the images now also

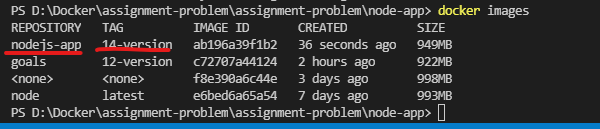


5 – step: Re-build the images - this time with names and tags assigned to them.

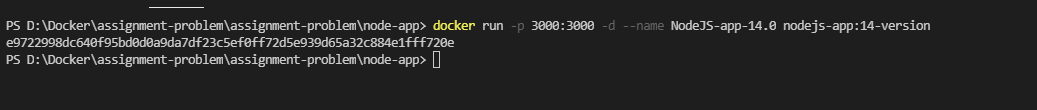
Creating the image for NodeJS with name

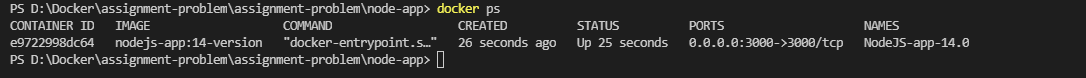


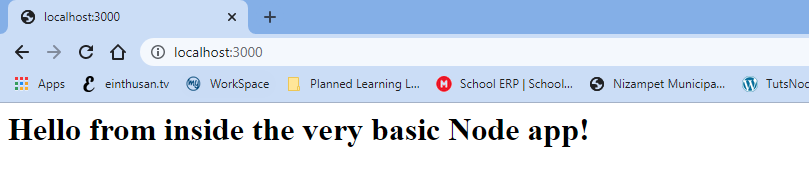




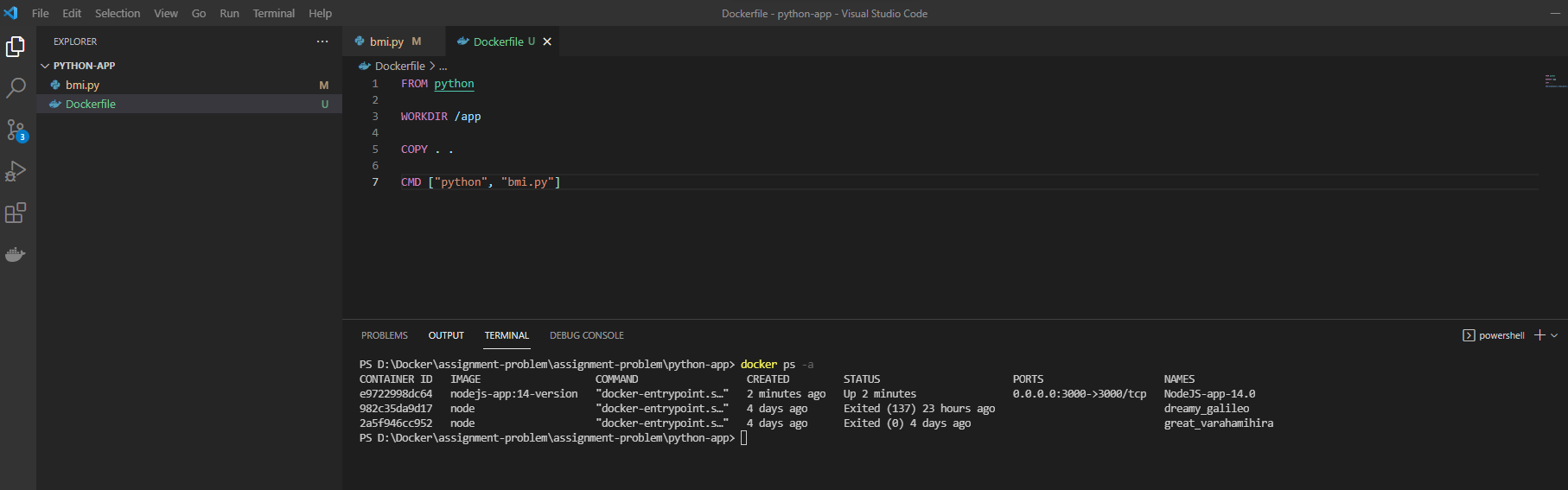
Let’s start container with this NodeJS App image now.

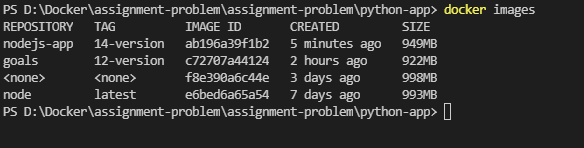


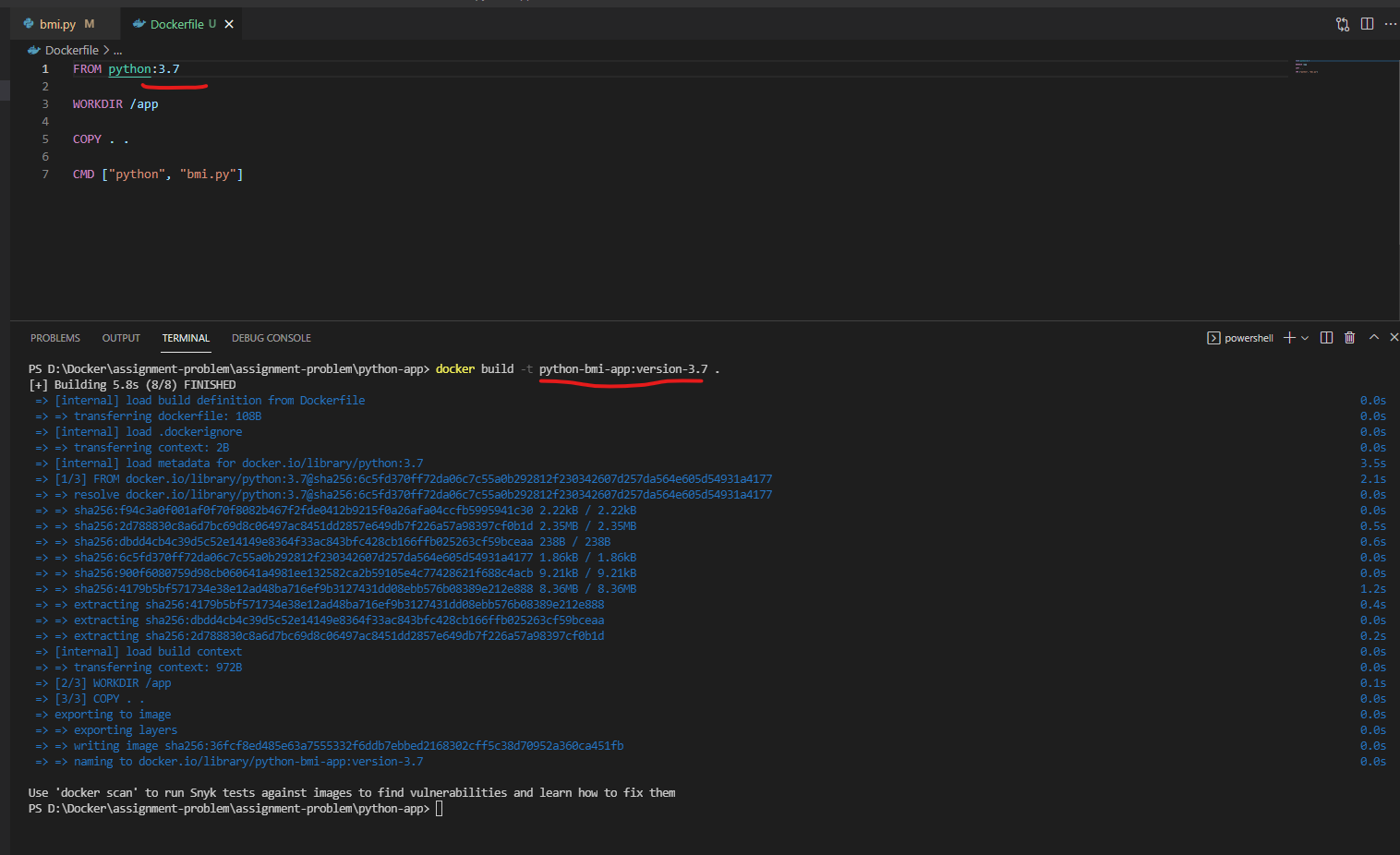


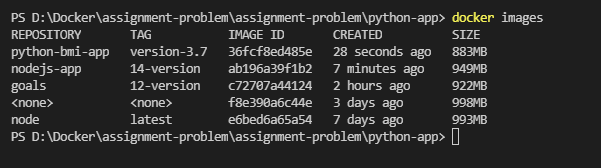


Creating the Image for Python App with proper name now

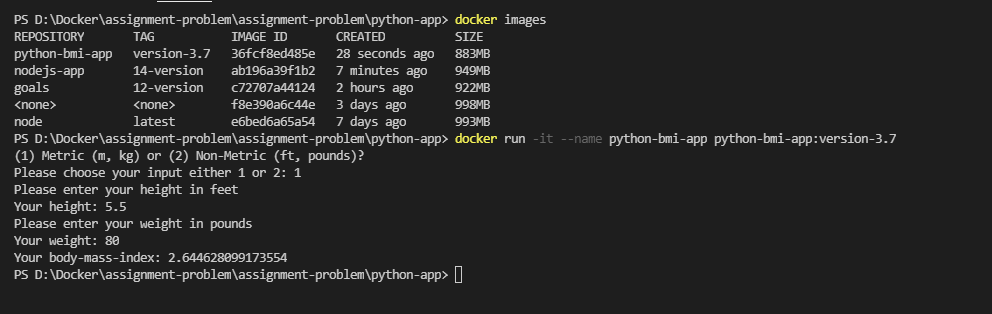




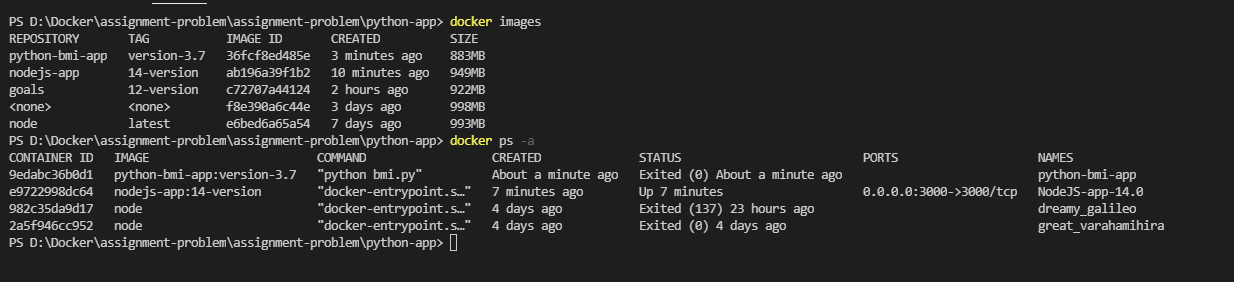


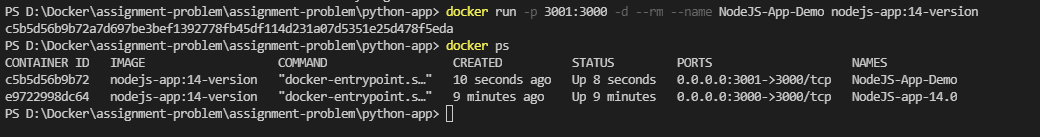


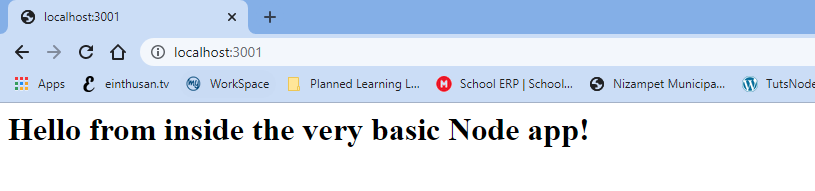
Let’s run the container by using this python image

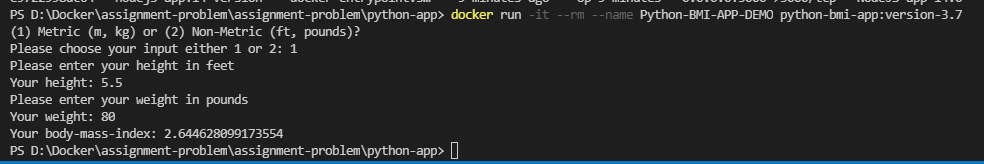


6 – step: Run new containers based on the re-built images, ensuring that the containers are removed automatically when stopped.

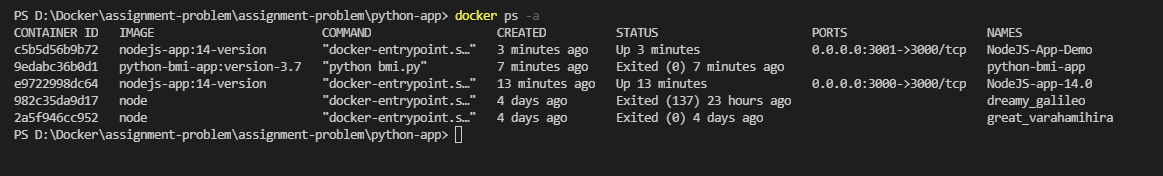


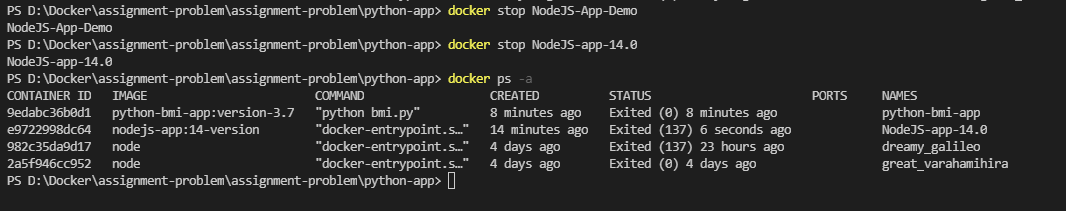


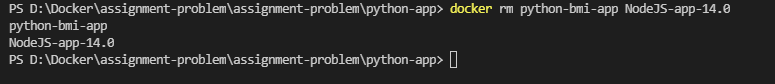


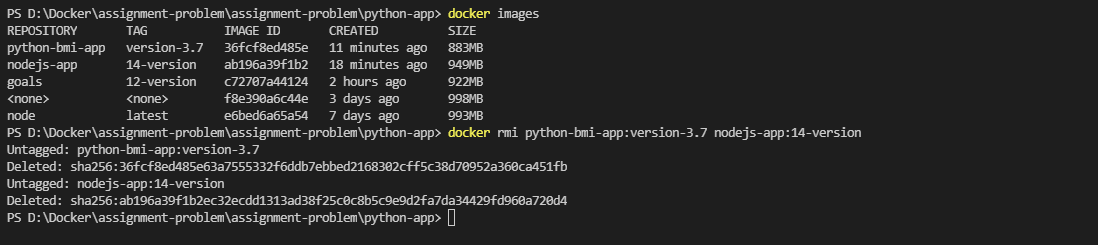


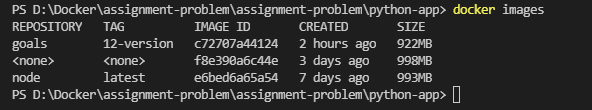
Finally clearing all the containers & Images created up to now.











## Docker Pushing Images to repository

You can have two ways to push your images

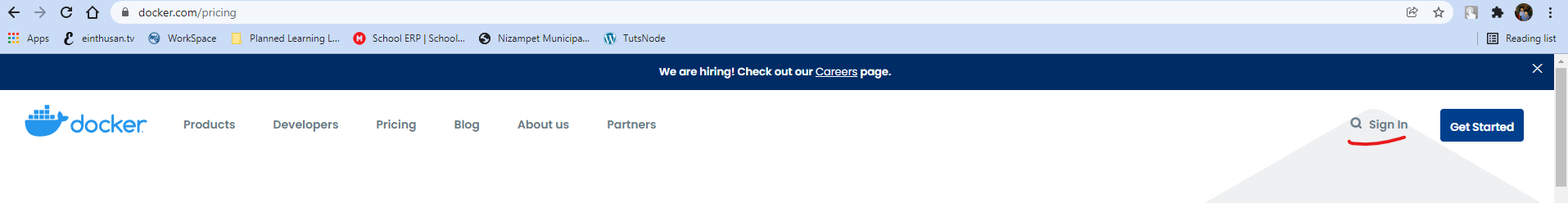
1. Docker hub
2. Private Repositories 🡪 There are so many repositories exists, you need to choose one of it. We will discuss this private repositories later

Login to docker hub, by using the following link

<https://hub.docker.com/>

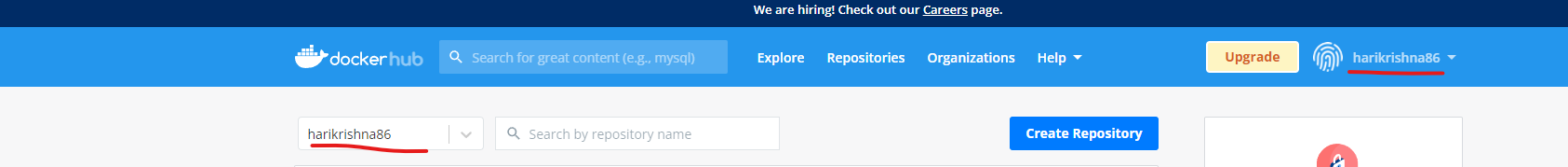
You can check the pricing at the following link, please go through the link and choose the appropriate method.

<https://www.docker.com/pricing>

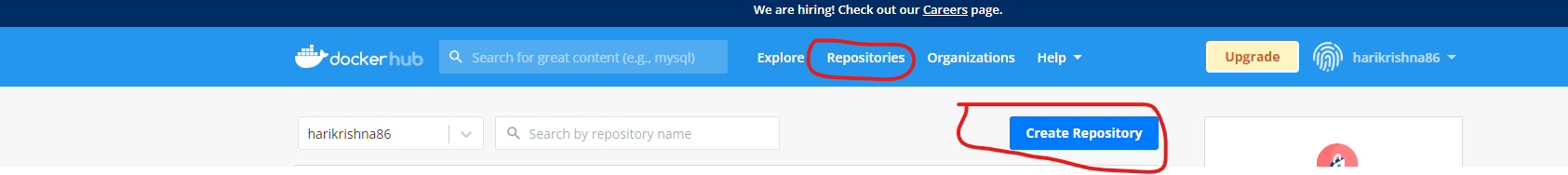


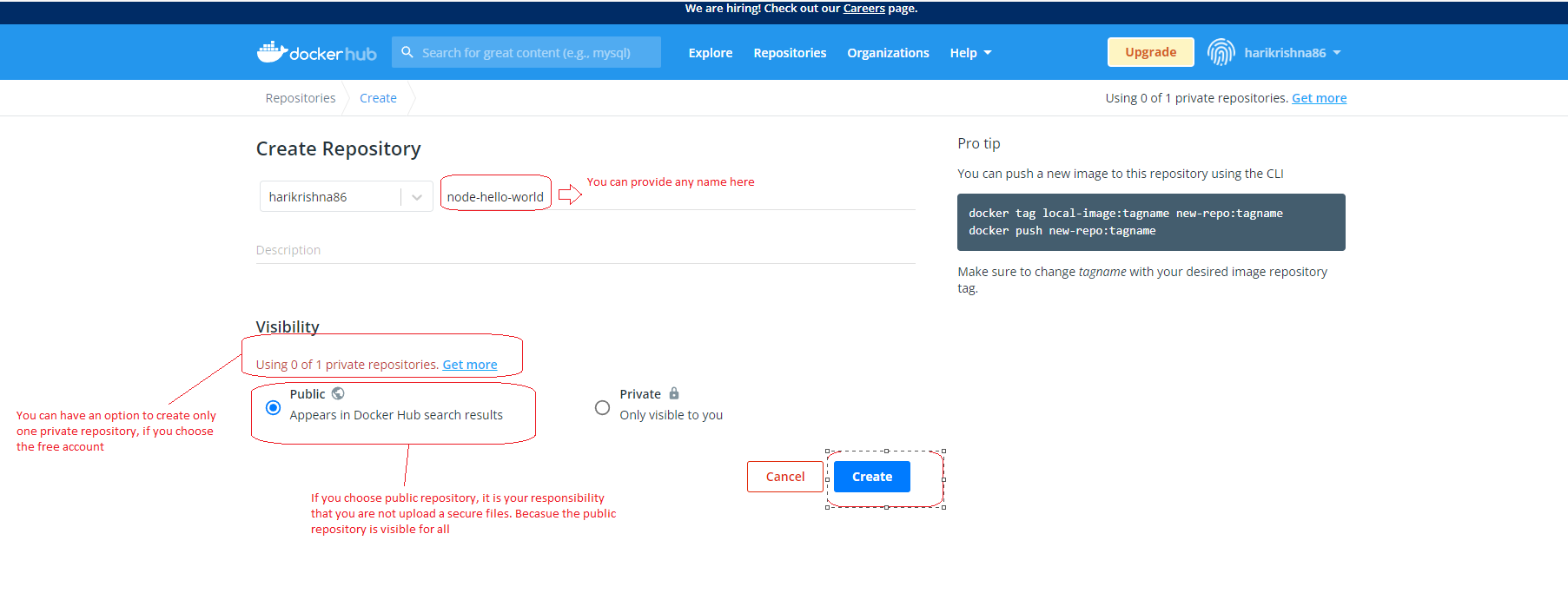
You need to create an account on docker hub and create a new repository in order to push your images.

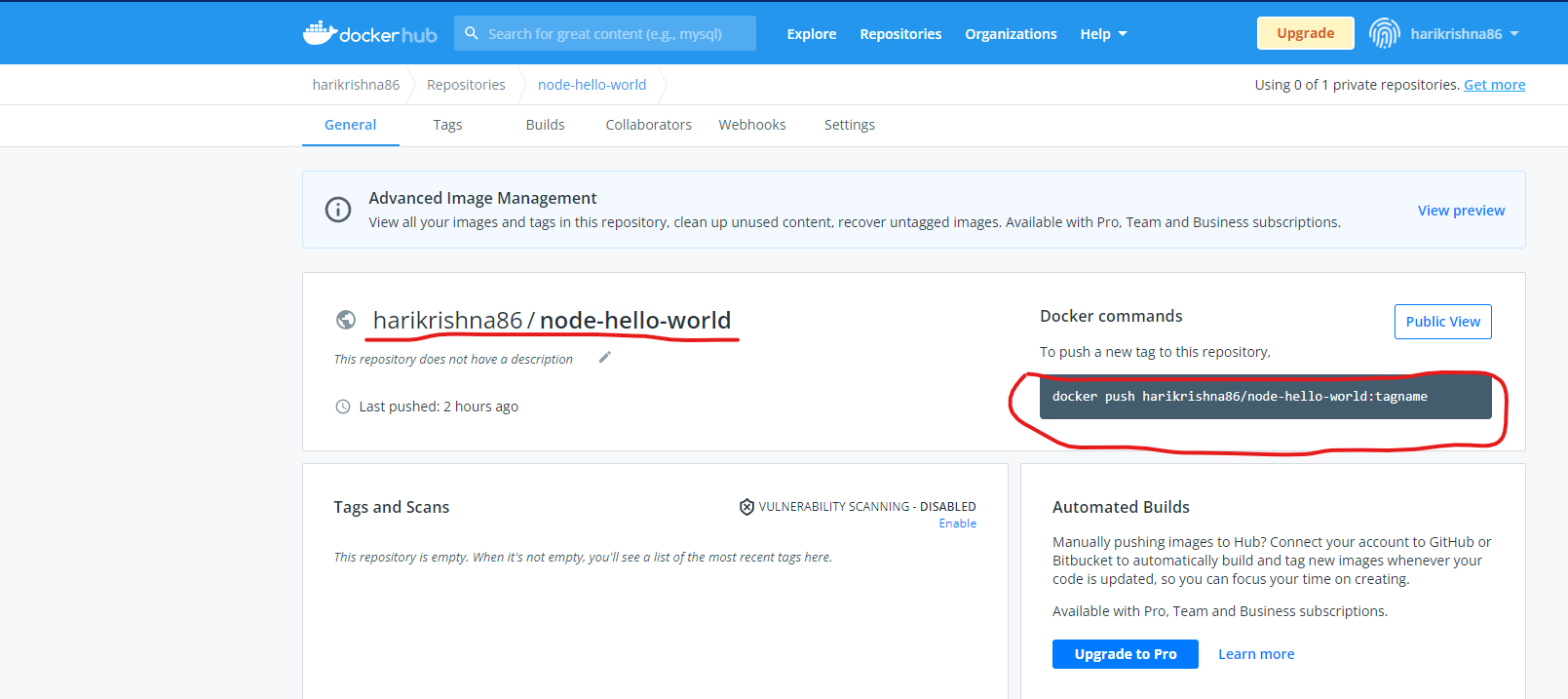
I had created an account and logged into my account after verification as shown below.



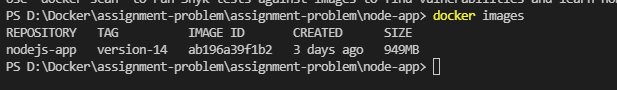
Now you need to create a repository, in order to store images, you can create a repository by clicking on “Repositories “ 🡪 “Create repository”.



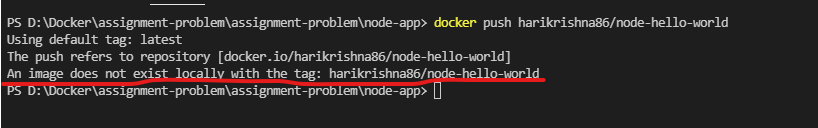




As of now I don’t have any images as because I didn’t pushed it any still. You can push the images by using the command “docker push harikrishna86/node-hello-world”



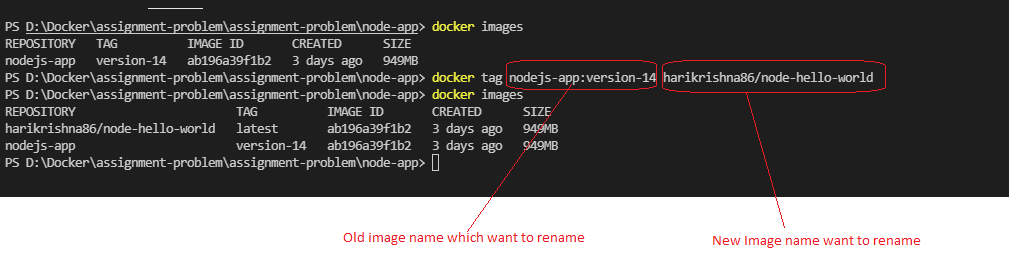
If you try to push the image now with the above command, it will throw an error because the image won’t exists locally with that name.



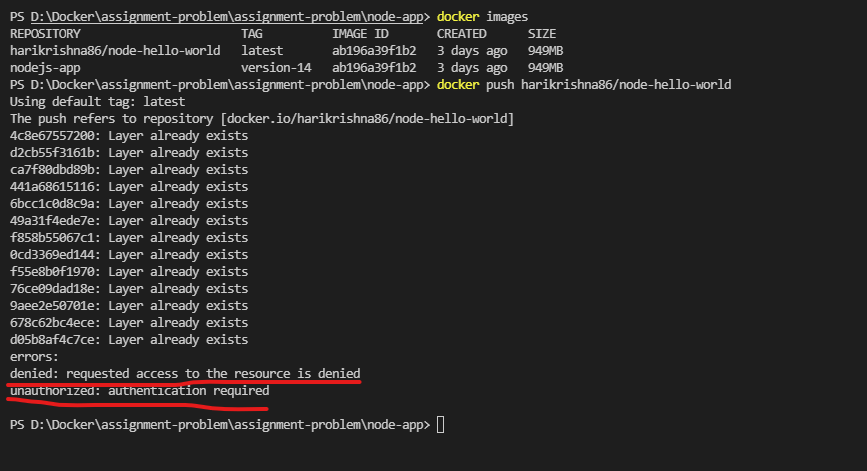
So we need to rename that image in order to push to docker hub. So renaming can be done in two ways

1. You need to build the image using docker build command run that image using docker run command. ( Note: this will create a new instance)
2. You can rename the existing image using the command “docker tag” (Note: this command will clone the existing image and create a new one.

As the image already exists, I will use the command “docker tag” now

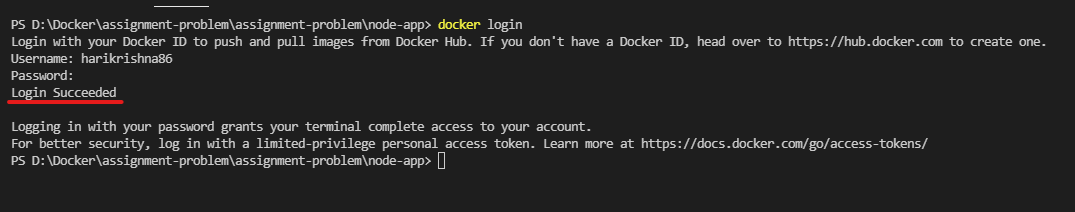


Now let’s try to push the image to my repository on docker hub

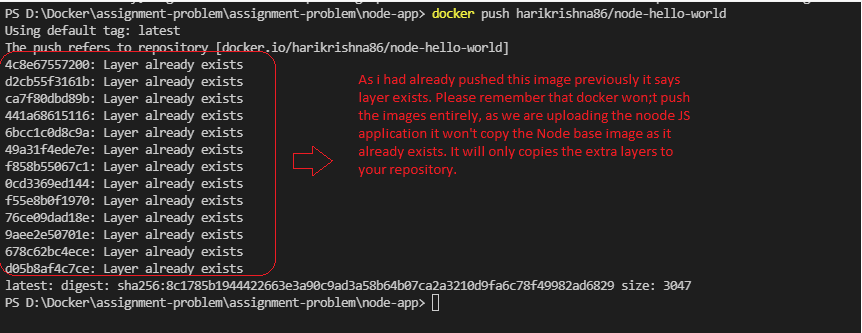


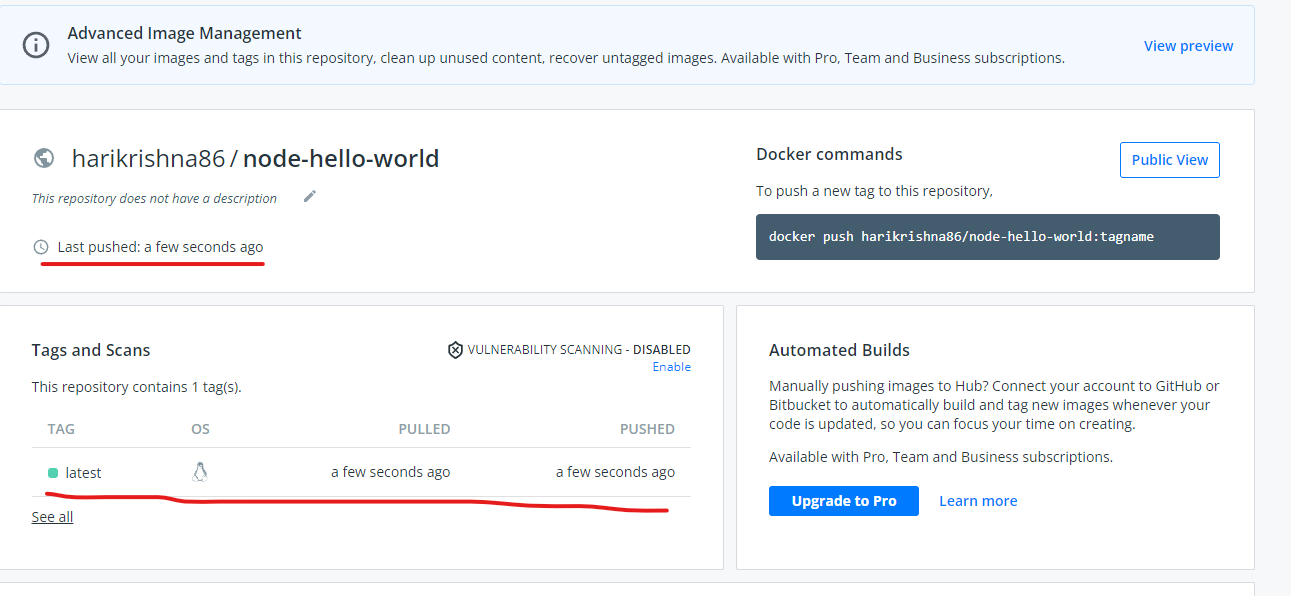
It says authentication required and access is denied, because in order to push an image to your repository you need to authenticate, if this authentication won’t exists anyone from public will upload their image on your repository.

So in order to authenticate, you need to login to your repository by using the command “docker login” and logout by using “docker logout”.



Now let’s try to push the image to your repository again.

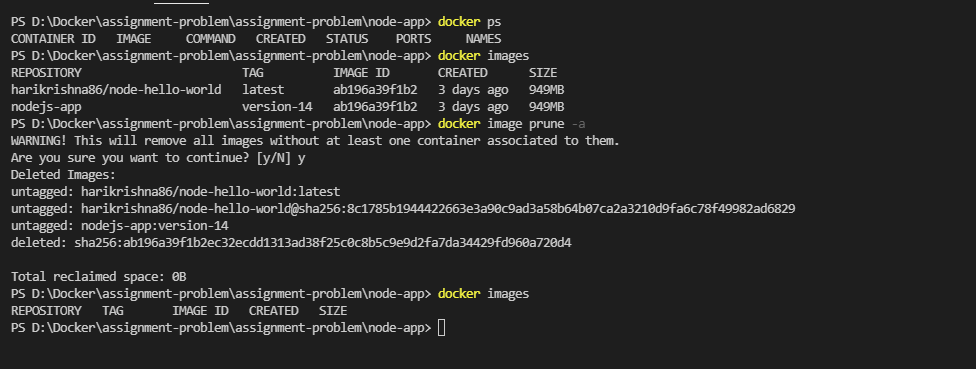




Note: Docker will pulls the latest image from docker hub if the image doesn’t exists locally. But if the image exists locally it will give preference to local even though it is not latest.

## Docker Pulling and using shared images

Up to now we had pushed the image to our repository, now we pull the image from local repository by deleting the existing image.



Note: To download or pull the image from public repository, you won’t need to login you can also pull the image by logging out.

