Up to now we know how to create and run the Docker Images & containers, now we are going little in-depth to deal different kinds of data which resides on Images, Containers and Volumes using arguments & Environment variables.

There are different kinds of data that we need to deal and we understand what of kind of data that we are dealing with it.

Up to now we had only deal with Application (Code + Environment), it was add to the image while building the docker image and it was read only. Application (Code + Environment) store on Read-only, hence stored in Images.

Another type of data that we need deal is “Temporary App Data”, it won’t deal with the Application code + environment but it will deal with the data that was generated by application at run time temporarily. We might store this type of data in a variables or in a database and can be lost if the container shutdown. This data can be stored on top of the extra layer ie., running container not on Images.

Last type of data that we need deal is “Permanent App data” ex: User accounts. We can store this data in files or in a database. We should not loose data once the container shuts down, so the data should be read-write on containers permanently on volumes.

So In feature we need to deal with the docker volumes a lot.

Permanent App Data (e.g. user accounts)

Temporary App Data (e.g. entered user input)

Application (Code + Environment)

Read + Write, permanent, stored with Containers & Volumes

Must not be lost if container stops / restarts

Read + Write, temporary, hence stored in containers

Read-only, hence stored in Images

Added to image and container in build phase

Written & provided by you (= the developer)

Dynamic and changing, but cleared regularly

“Fixed”: Can’t be changes once image is built

Stored in files or a database

Stored in memory or temporary files

Fetched / Produced in running container

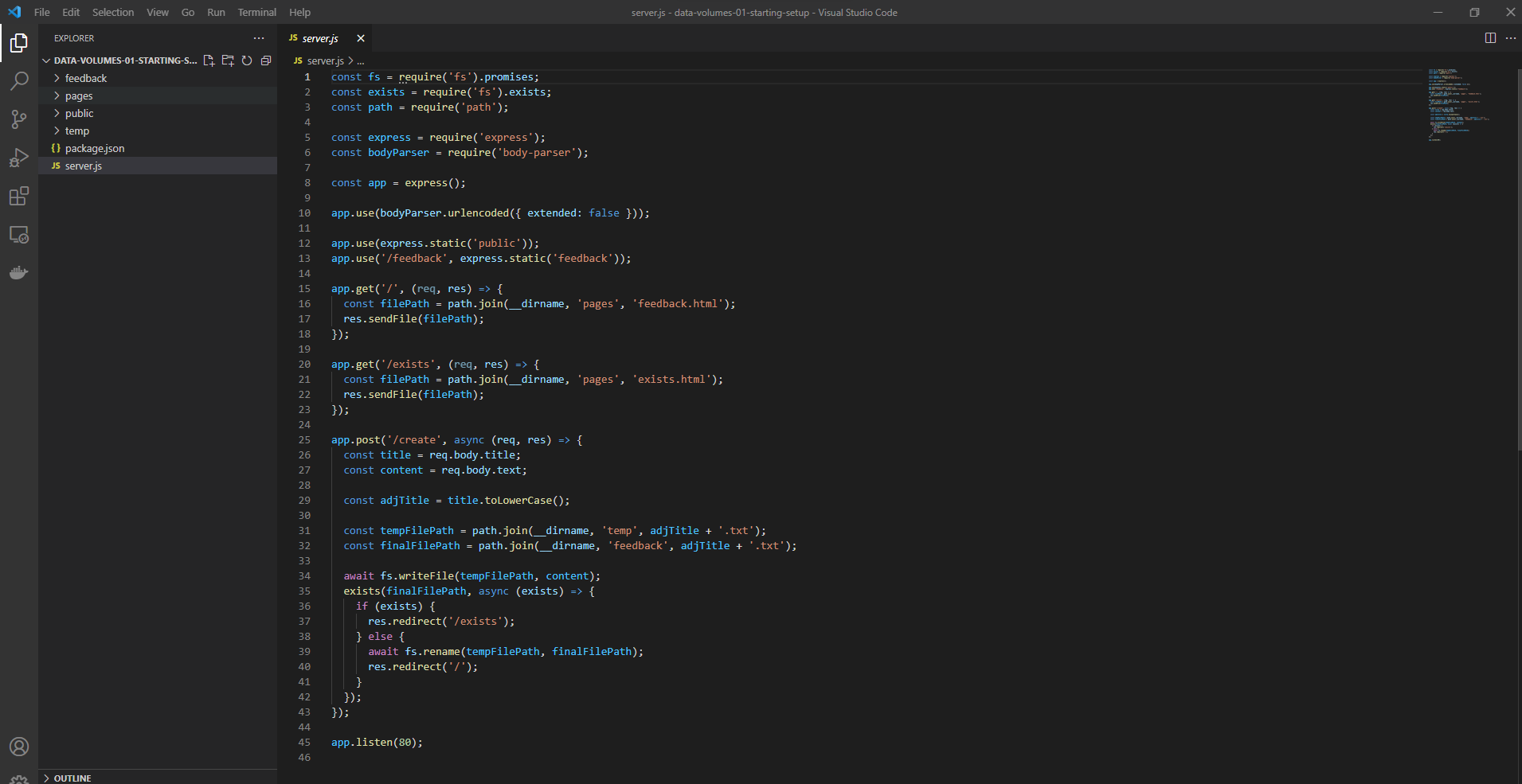
Fetched / Produced in running container

## Dockerize with real time NodeJS Application

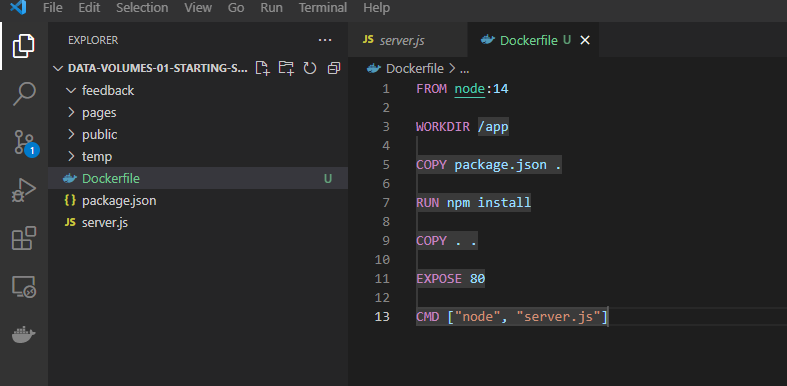
Now we are going a build a NodeJS Application, which will accept a feedback and store in a temporary files initially, once user submits the feedback it will store the files permanently on feedback folder.

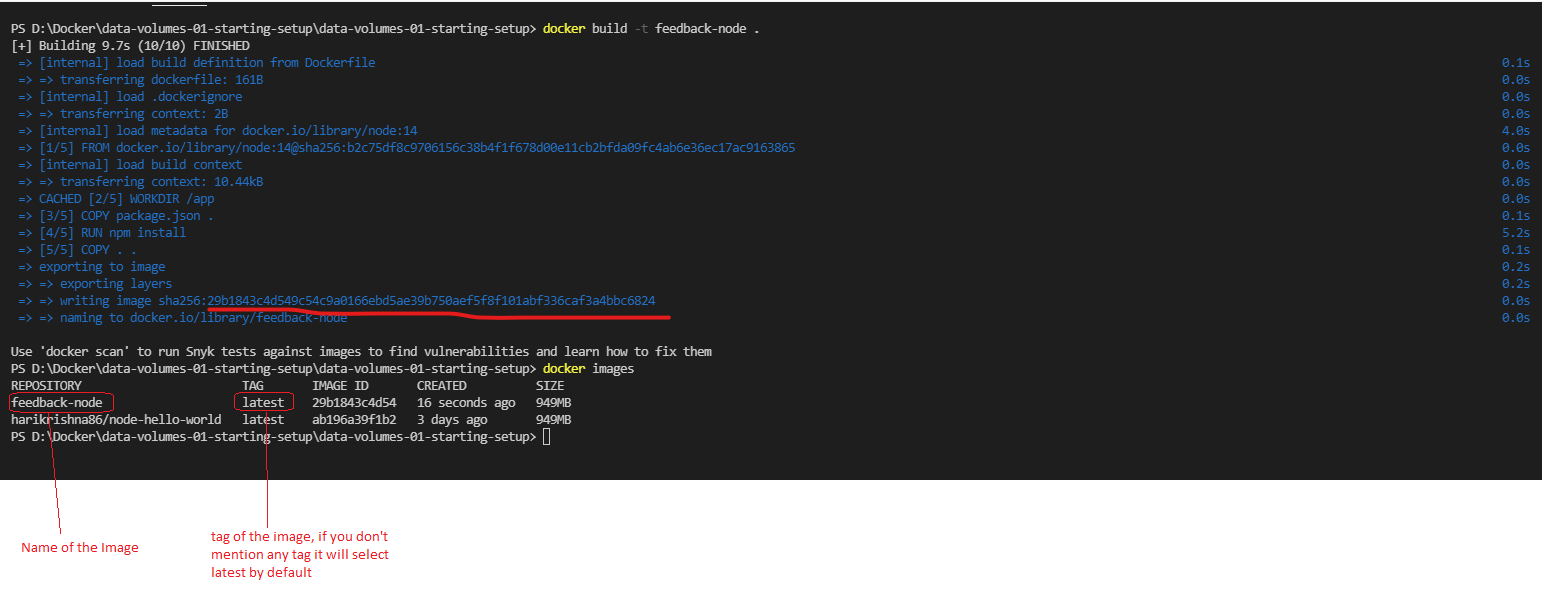
Download the code from the following location.

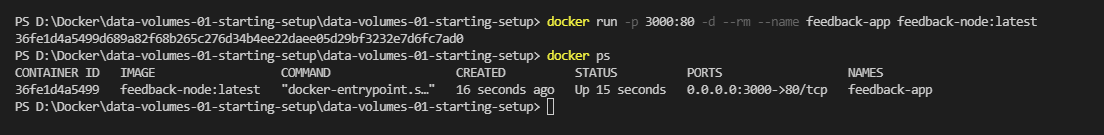
<https://github.com/harikrishna83/Docker-firstdemo-handson/tree/master/data-volumes-01-starting-setup>

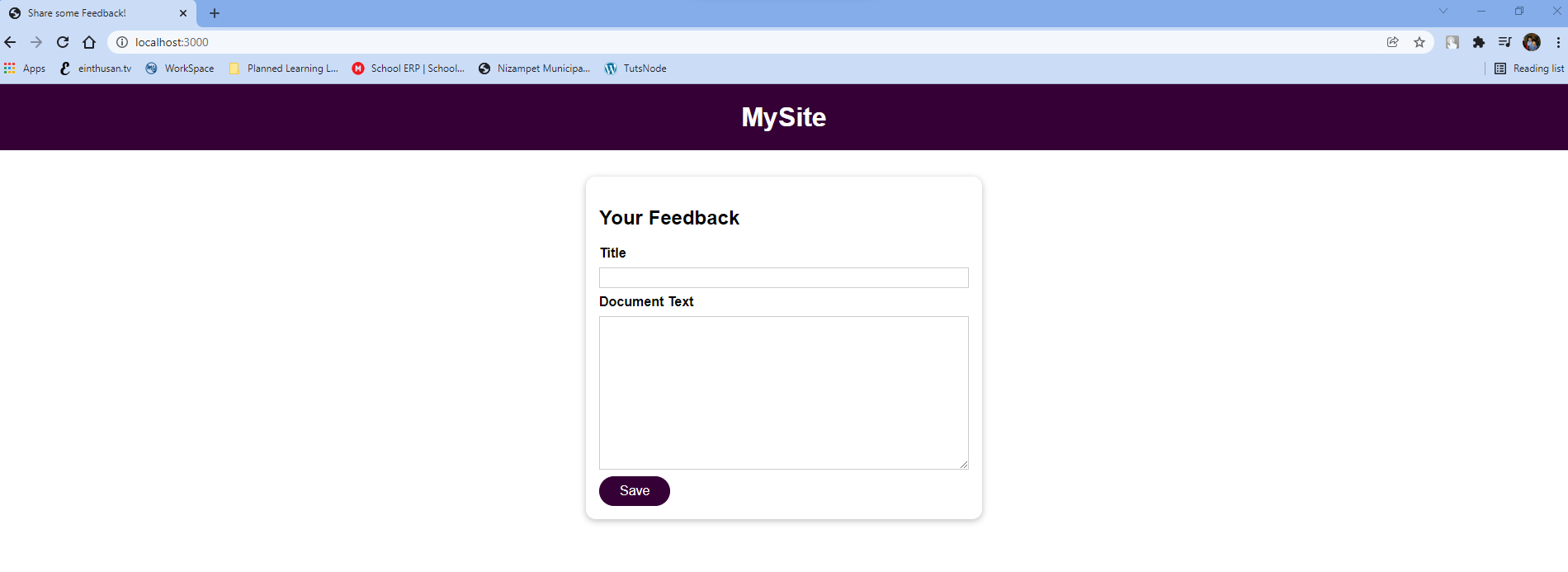


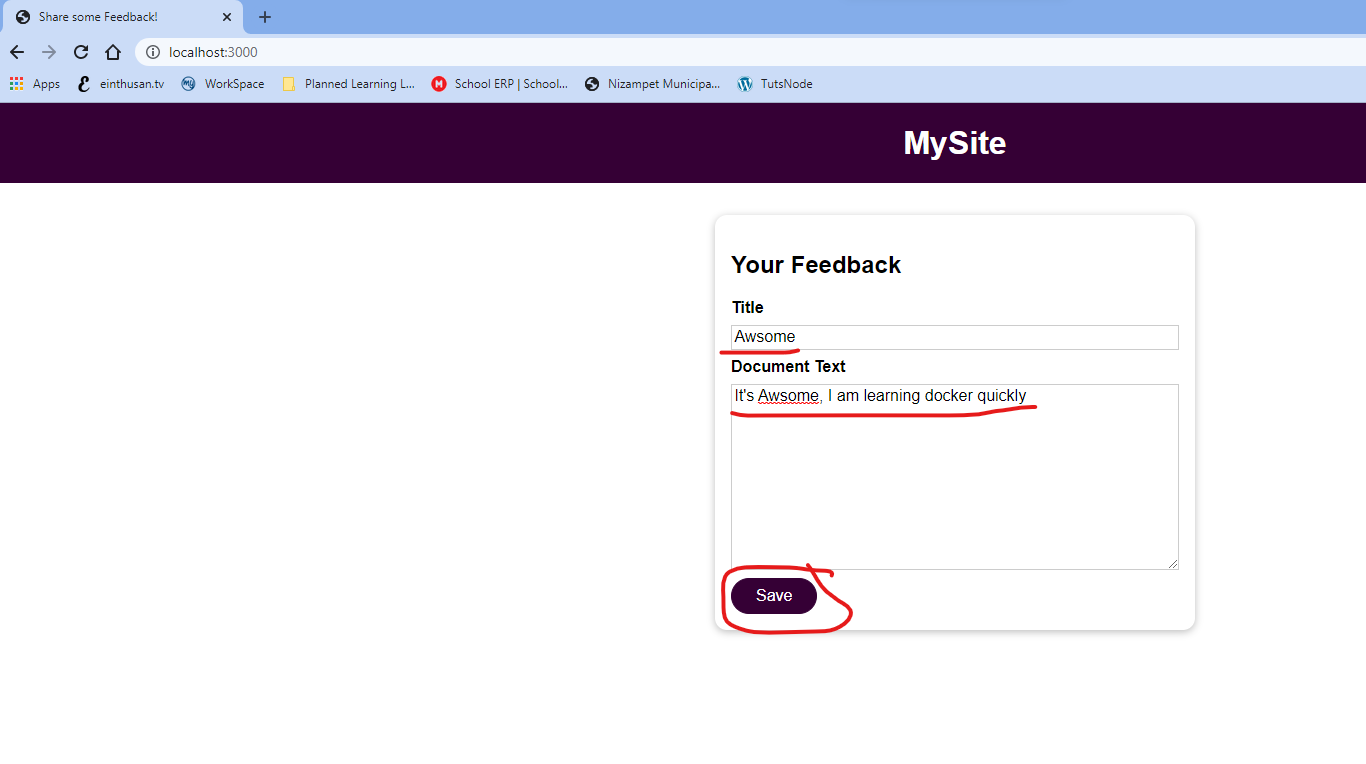
Now let’s get dockerize this app by creating a Dockerfile.









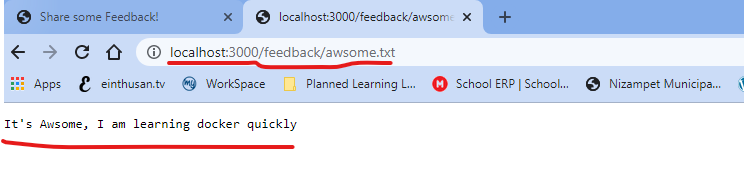


As per the NodeJS application, whatever you enter on title section it will create with that name so make sure you keep the title as simple don’t mention any special characters.

Note: if you enter any capital letters, as per the code it will convert all into small characters and save it.

As I enter awsome on text, if you want to view the feedback we need to browse the below url, there is no other way to browse it.

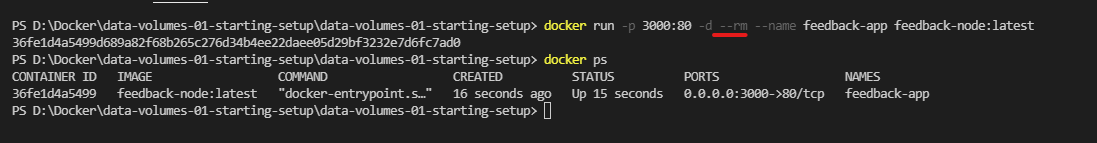
<http://localhost:3000/feedback/awsome.txt>



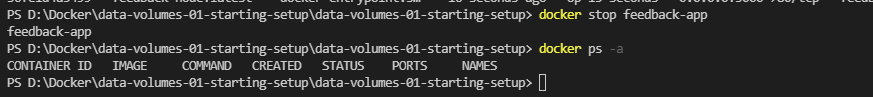
We can’t see the feedback in a files, because once the instruction of “COPY ..” on Dockerfile as there is no connection to your local file system to docker container, and the image is read only, we can’t be able to copy the files locally from your machine to the docker container / images. It only stored in a temporary files and be able to browser [http://localhost/feedback/<filename.txt](http://localhost/feedback/%3cfilename.txt)>

That is why on previously, whenever we change the code we re-builded the image to reflect it. This problem can be addressed in feature by using docker volumes to store the file permanently.

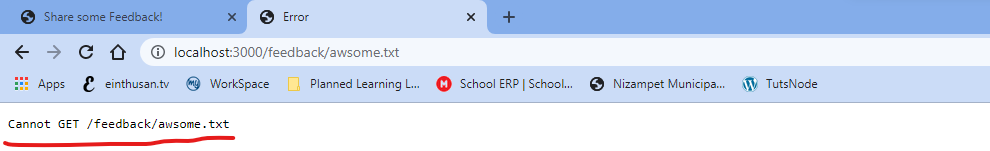
## Understanding the Problem now



As we are running the container by using the option --rm, if we stop the container it will automatically remove the container and you will loose the feedback.

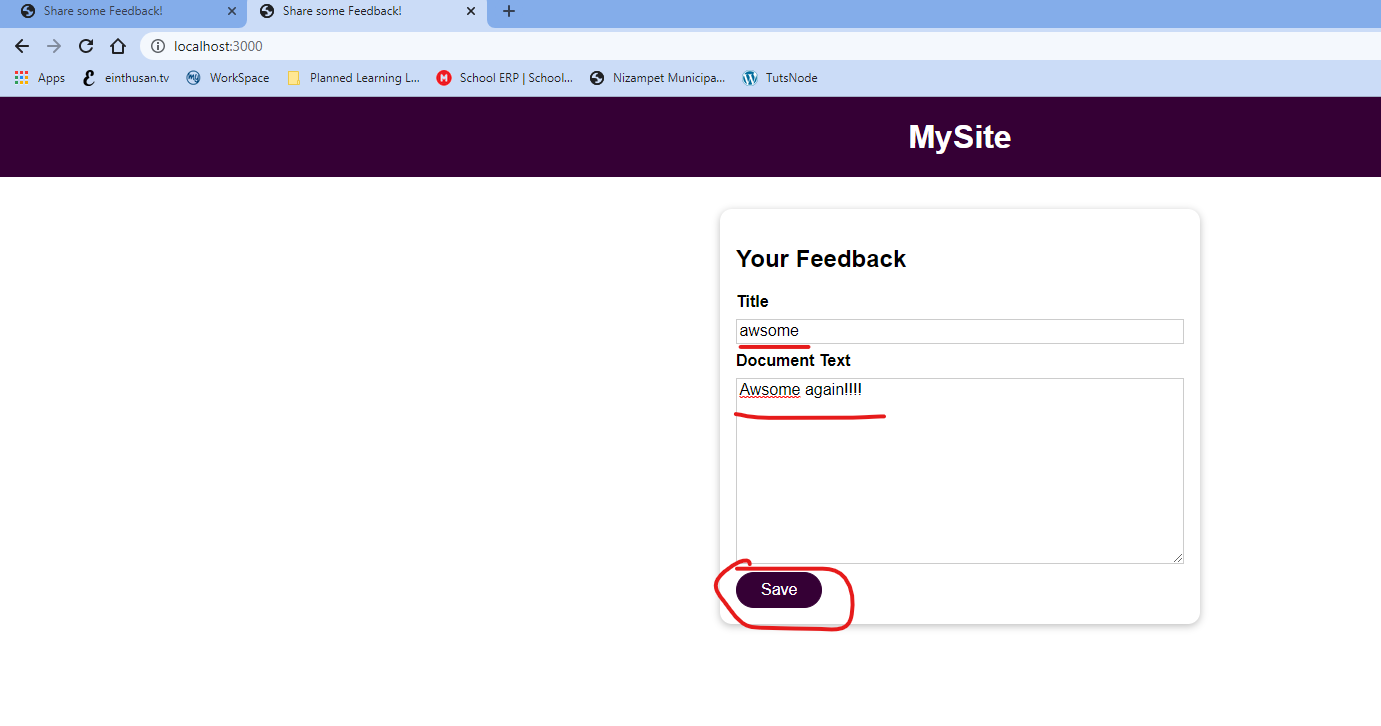


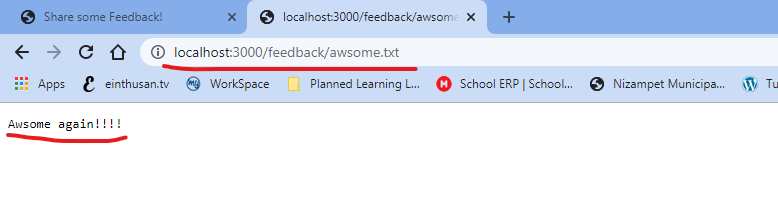




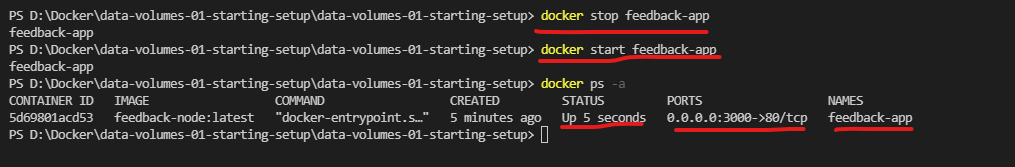
As you had re-created the container, all the feedback got erased now.

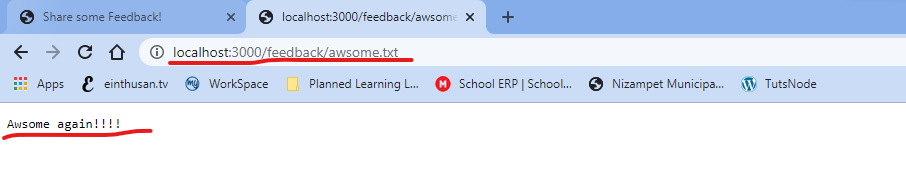
Suppose if you stop the container and start the container, it won’t erase the container filesystem. As container had read-write file system layer the feedback still exists.





Now let’s start and stop the containers and see if the feedback is available or not





As you can see after stop and start the container, the feedback still exists because the container file system layer is read-write and it won’t erase it start and stop the container. We will loose the feedback only if the container is lost or removed as container file system we have only an image and image filesystem is read only and fixed.

Now we understand the problem that we are facing, to deal with this problem docker had inbuilt feature called volumes, we will use this volumes and solve the problem.

Volumes are folder on your host machine, it’s not the folder on containers or images. Which basically means it made available on your local machine and mount that folder to your images / containers.

Volumes are connected inside your local machine to containers, suppose if you create a file on that volume from outside of your container it will reflect that file on the container and viceversa.

Therefore volumes persist even if we rebuild the containers, only thing is you need to mount the volume from your host to the container.

Volumes persist if a container shuts down. If a container (re-)starts and mounts a volume, any data inside of that volume is available in the container.

/app/user-data

Host (Your Computer)

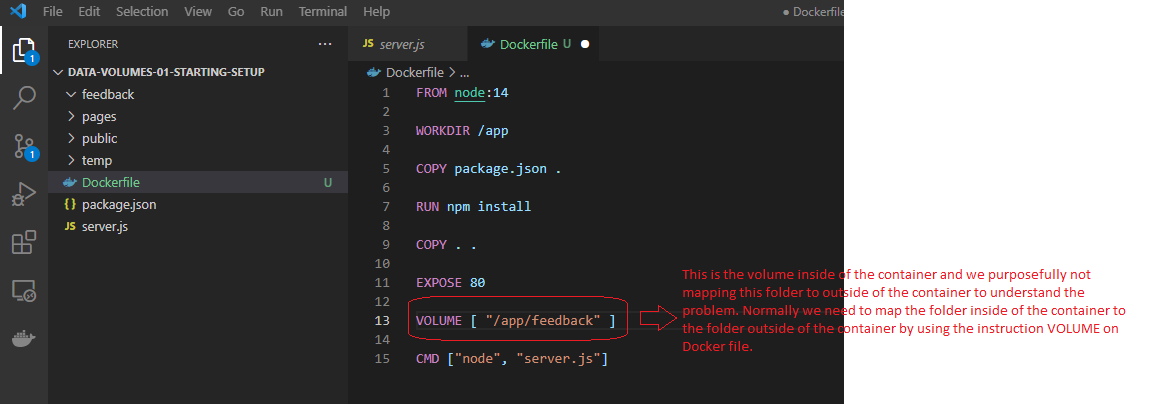
/some-path

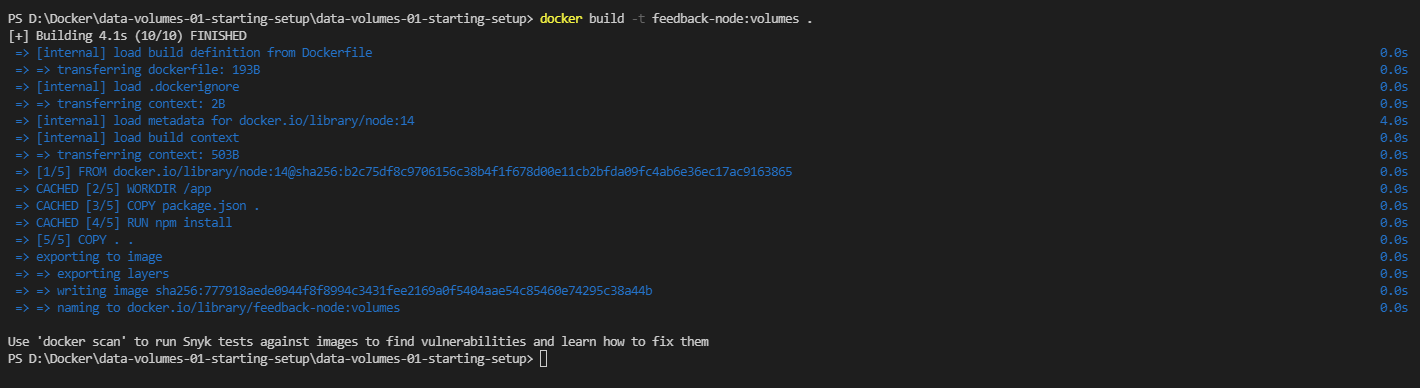
Volumes are folders on your host machine hard drive which are mounted (“made available”, mapped) into containers

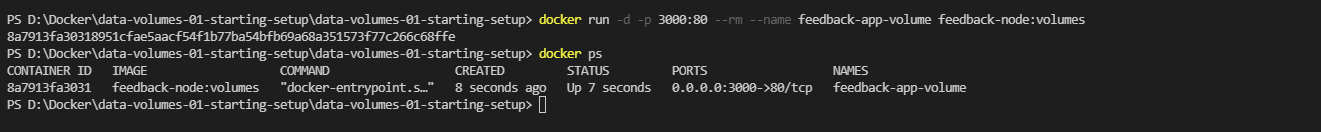
A container can write data into a volume and read data from it.

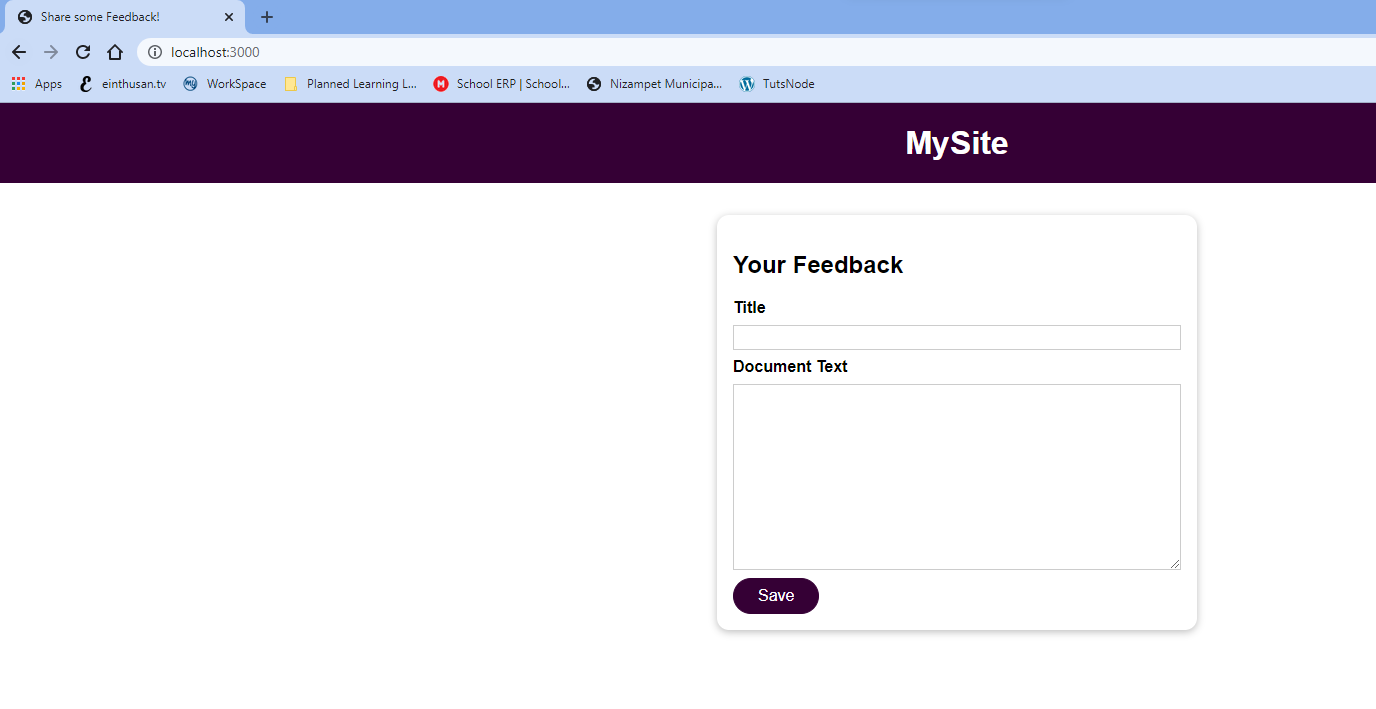
Now let’s try by adding volume to our container by modifying the Dockerfile.

As per the NodeJS Code, I am saving all the feedback files to a folder called feedback and temp files to another directory temp but these temp need not to be saved.

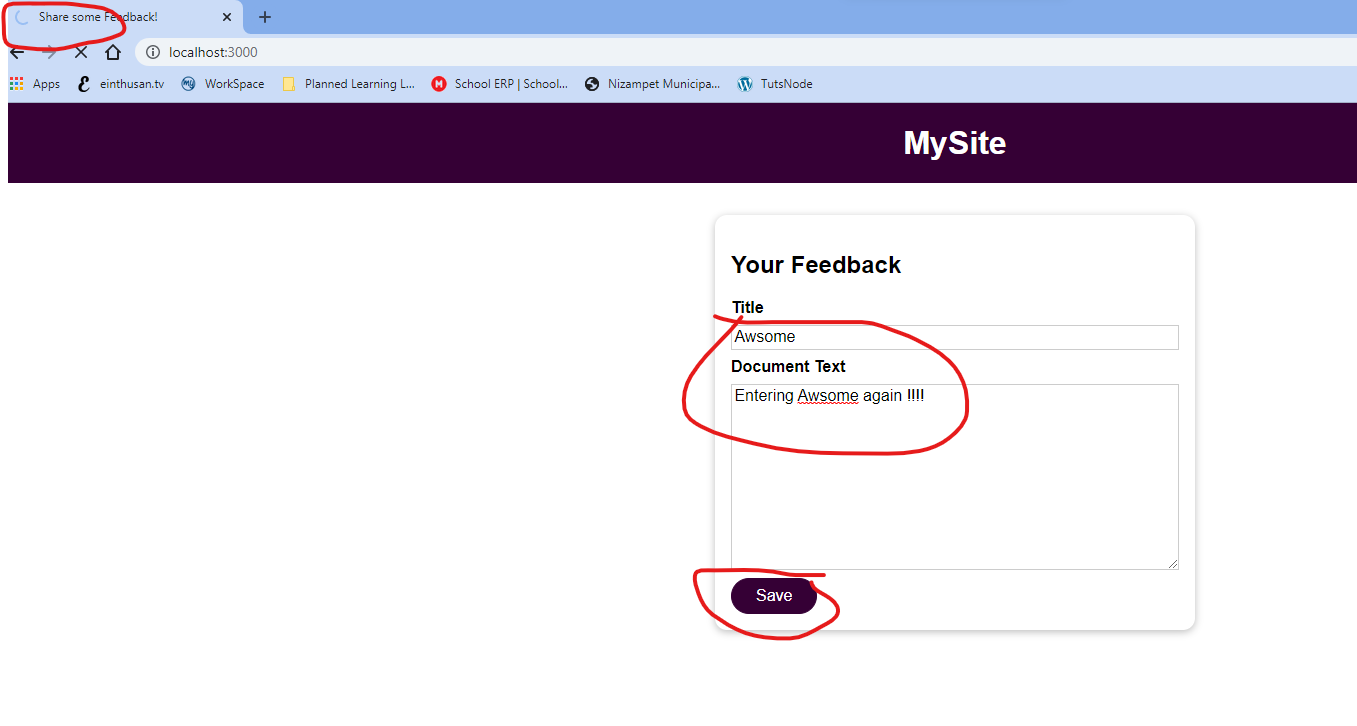
As our whole application resides under /app, I am choosing the volume inside my container as “/app/feedback”, and this folder should be mapped to outside of the container to some path inorder to store the data permanently.



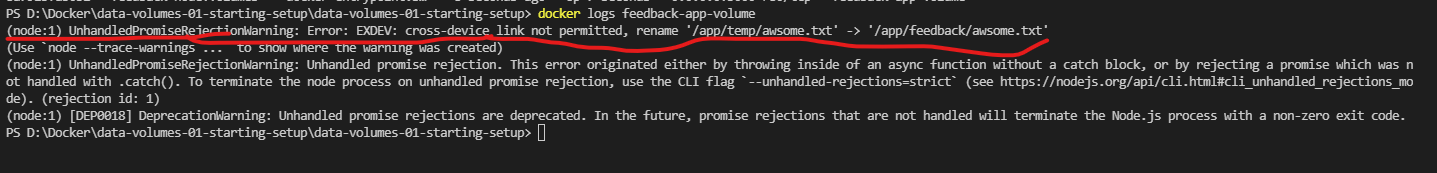




Now let’s try to save the feed and see if it will save or not, as we added the volume on the container.



It will not save the code, it will keep on try for some time and crash the node. In order to see what went wrong let’s issue the command “docker logs feedback-app-volume”

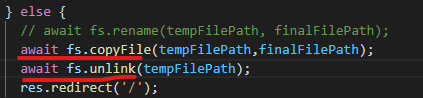


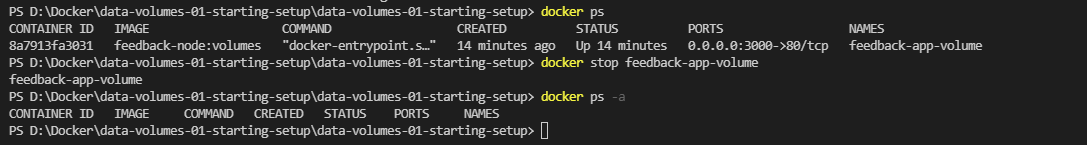
The error says cross-device link is not permitted, which means when it tries to copy the feedback from temp folder to /app/feedback folder this app crashed.

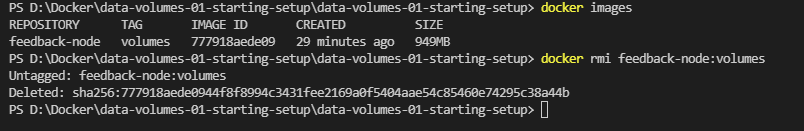
Here the problem is on the server.js rename method, this rename method doesn’t allows to copy the file from inside of the container to the outside path of the container (Volume).

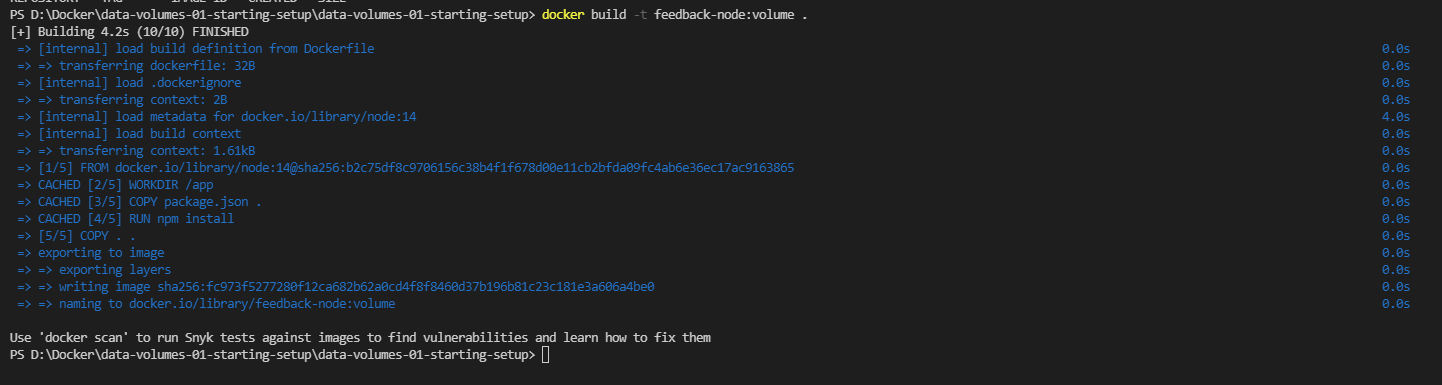


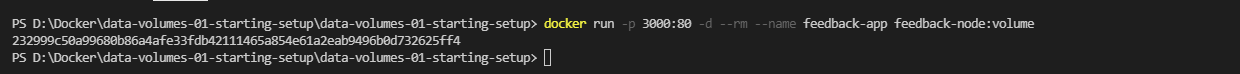
In order to remedy this we need to replace the rename method with copyFile method.

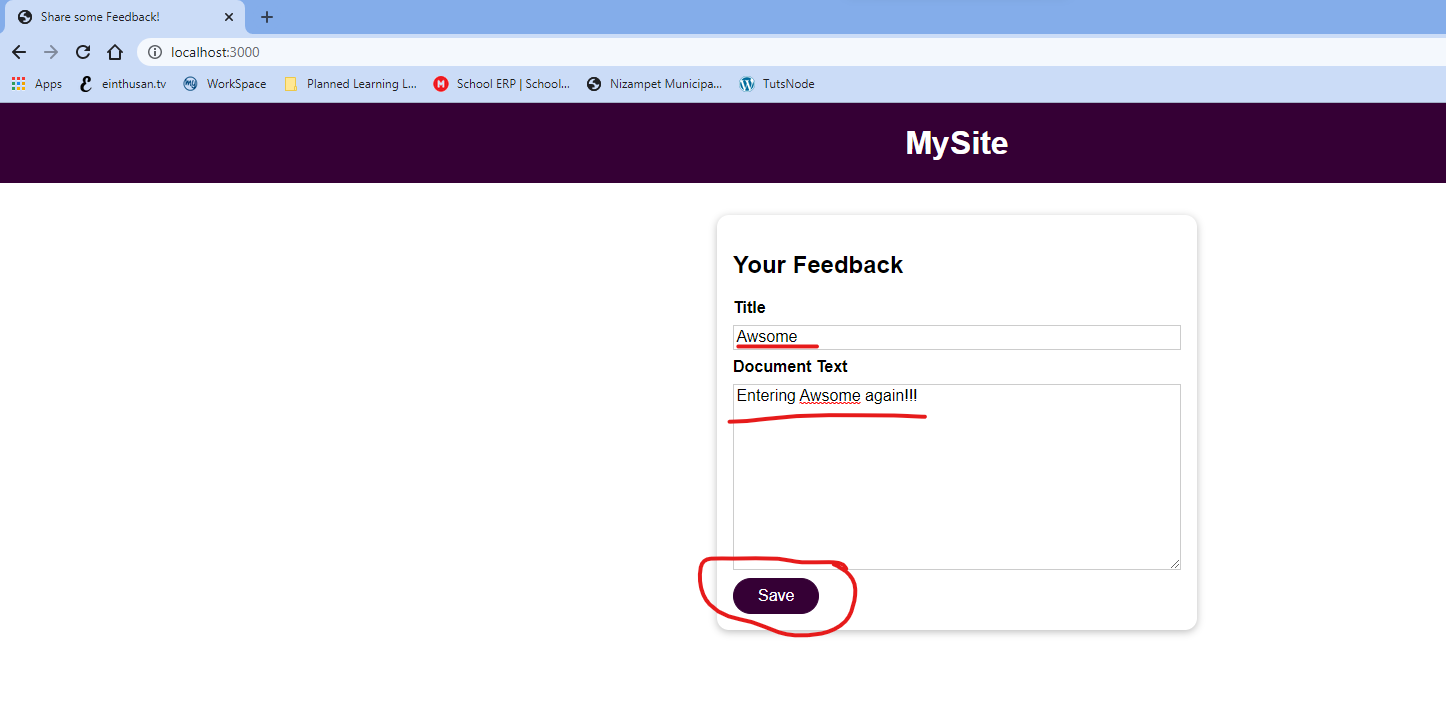




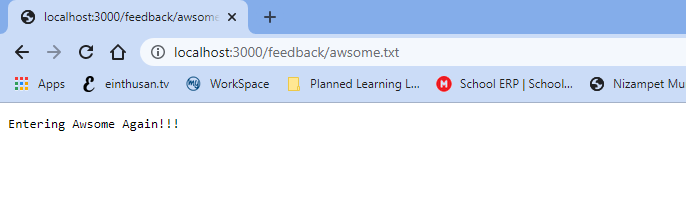




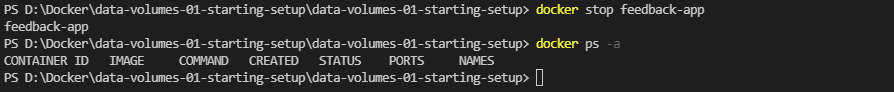




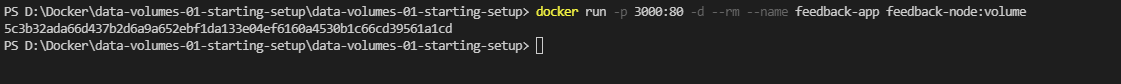
Now this time it will save the code. You can also verify the feedback by browsing the following url.

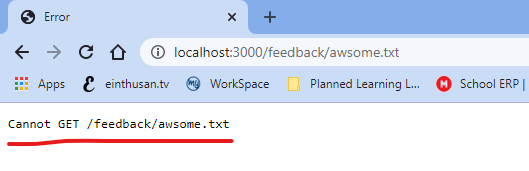


As you had used the option --rm previously it will remove the container and hopefully we will see the feedback still.



Now if I run the brand new container and verify it once we will not be able to find the feedback.





With docker we have multiple external data storage mechanism.

1. Volumes
2. Bind Mounts

Bind Mounts we will deal with later, now Volumes are of two types

🡪 Volumes

* Anonymous Volumes (Which will created by mentioning instruction on Dockerfile)
* Named Volumes (which will create by dynamically by using docker run command)

Each of them will have their own purpose, up to now we had create Anonymous Volumes by mentioning the instruction “VOLUME” on Dockerfile.

In either case Anonymous or Named volumes, docker will map the folder inside to the container with some path on your host which we don’t know where it is on your host system.

We can only see these volumes by using docker command “docker volume <option>”.

Volumes

(Managed by Docker)

Bind Mounts

(Managed by you)

Named Volumes

Anonymous Volumes

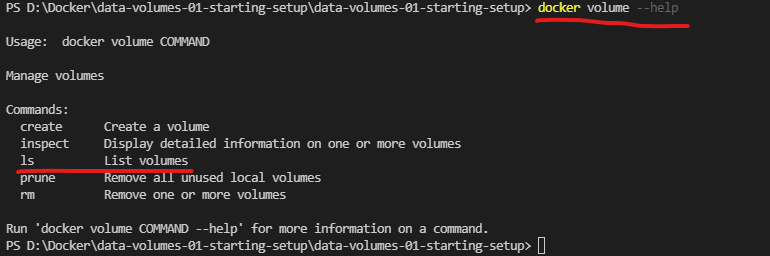
**You define a folder / path on your host machine**

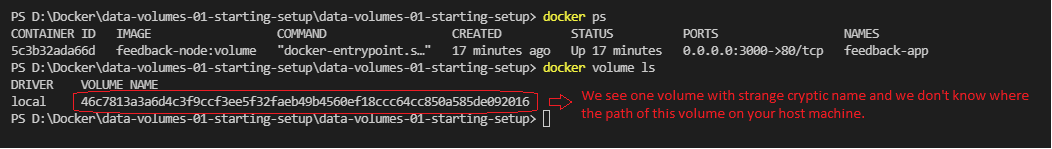
A defined path in the container is mapped to the created volume / mount. e.g. /some-path on your hosting machine is mapped to /app/data

Docker sets up a folder / path on your host machine, exact location is unknown to you (=dev). Managed via docker volume commands.

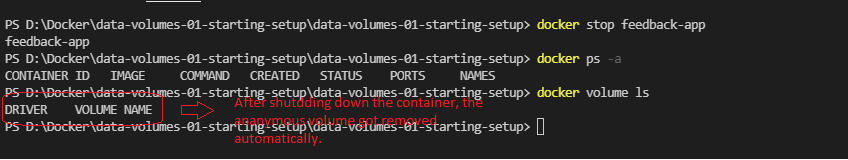
Great for persistent, editable (by you) data (e.g. source code)

Great for data which should be persistent but which you don’t need to edit directly.



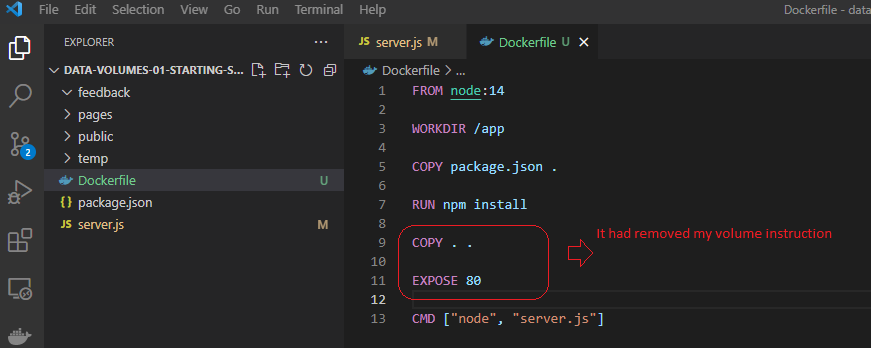


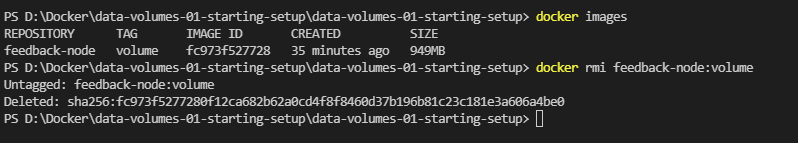
Now if we shutdown the container, as we had used the option --rm it will remove the container and see if the anonymous volume exists or not.

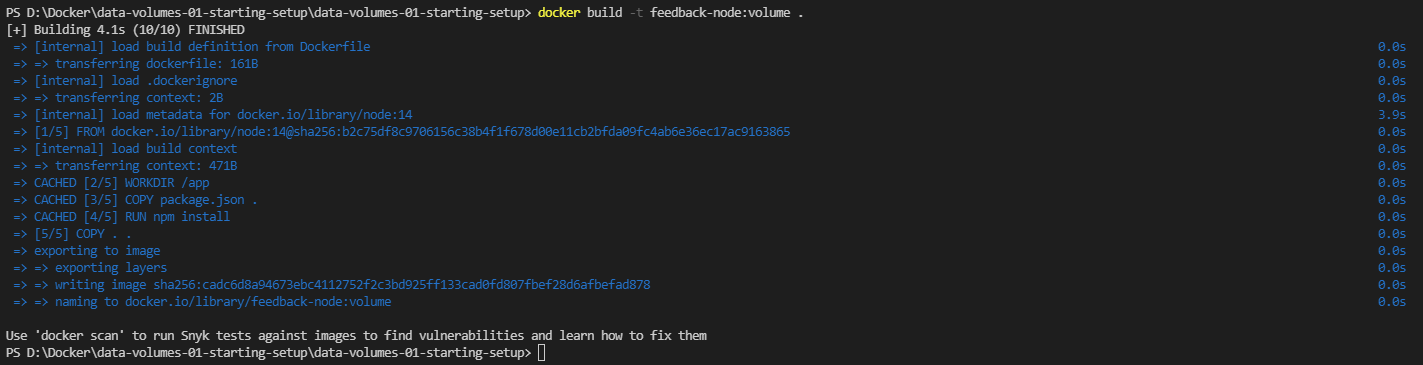


Now let’s see if we create a Named volume and if we shutdown and recreate the container whether the volume will exists?

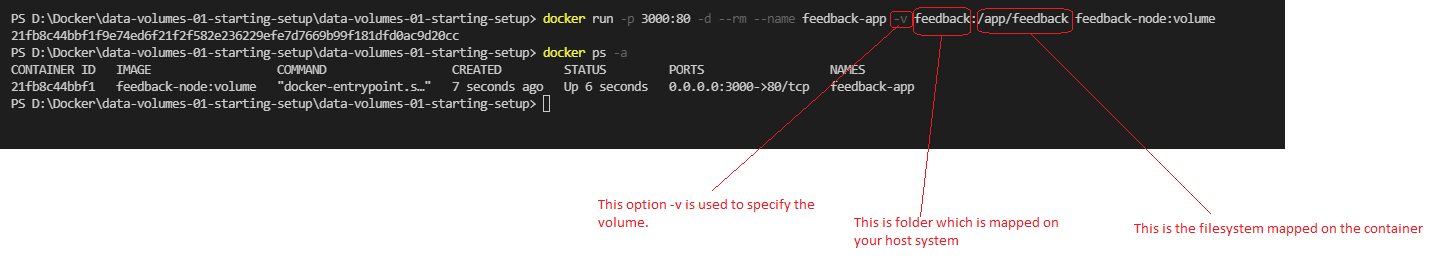
As we know to create a named volume, we need to mention at docker run command by using the option -v

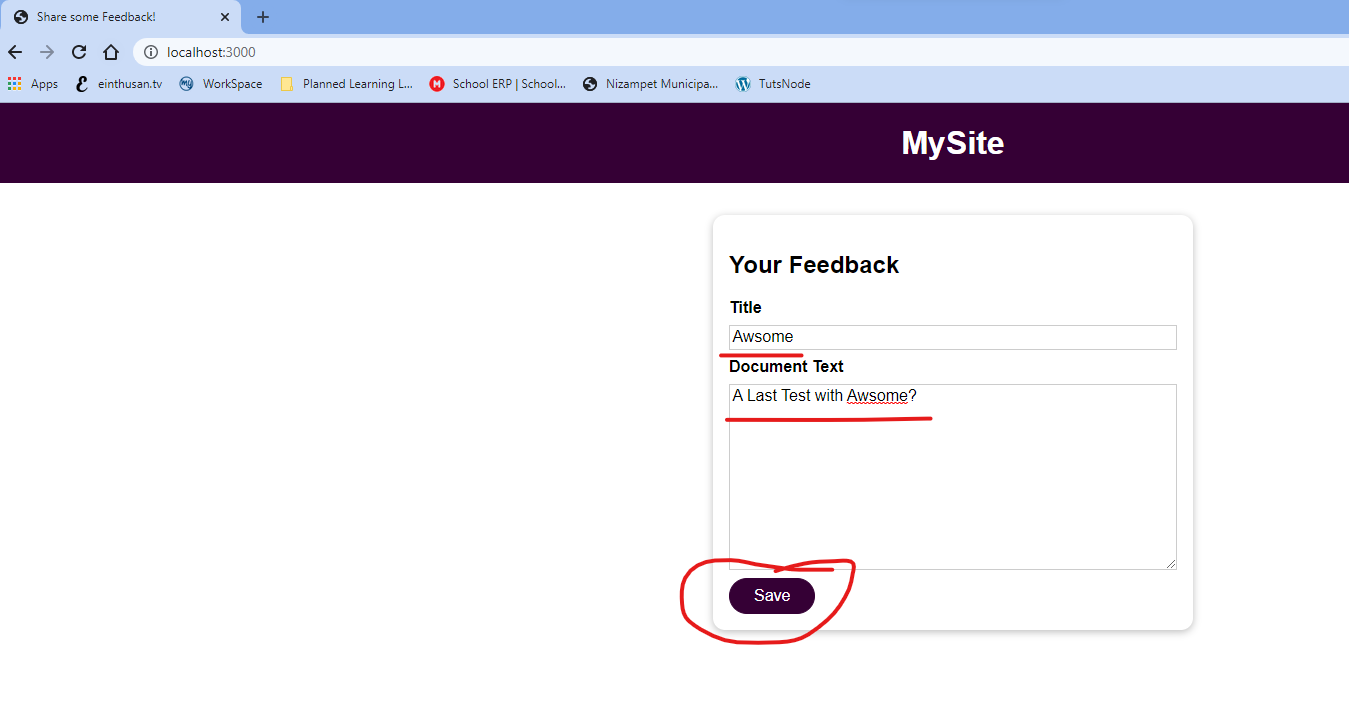


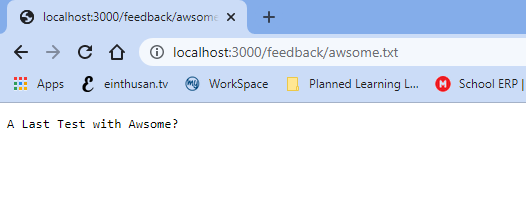


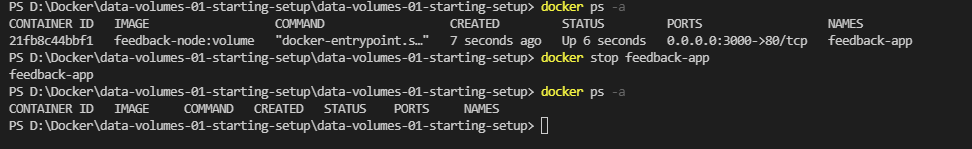


Note: Anonymous volumes are closely attached to one specific container, named volumes are not attached to container.

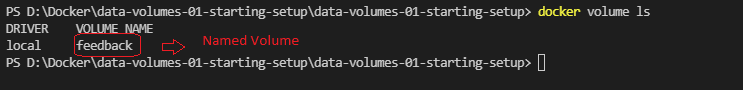






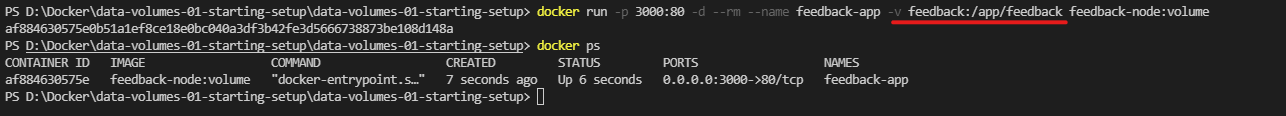


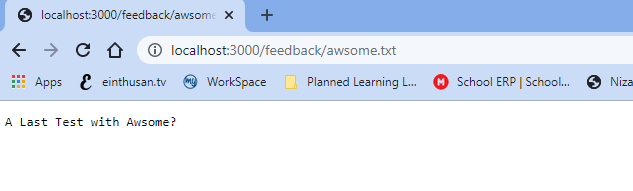
Now check whether the named volume exists or not.



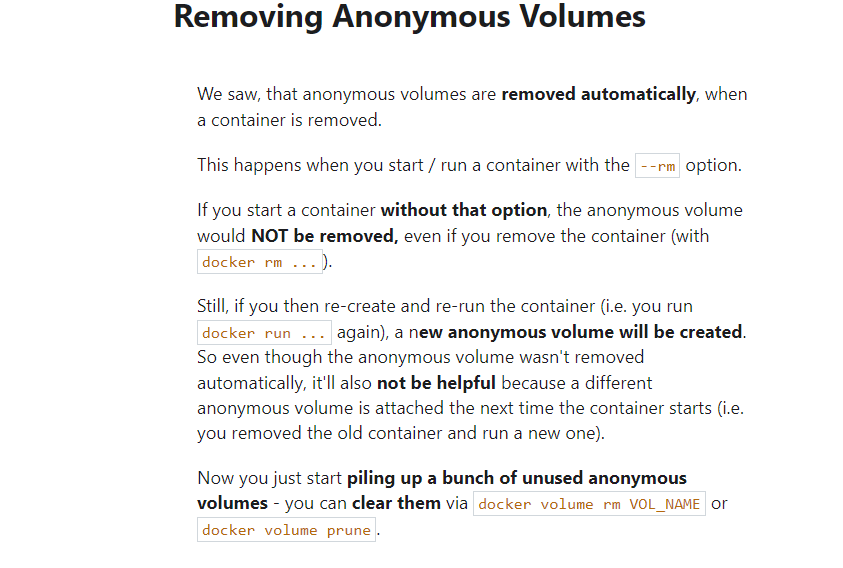
As you can see the volume still exists, we had option --rm previously, it will remove the container.

Now we if start the new container by attaching the same named volume, you will find the feedback, which was saved on that volume.





Finally we managed to persist data with help of named volumes.



Now let’s discuss about the Bind Mounts for docker volumes, these volumes are managed by you not by docker.

Bind Mounts can help to deal with different kind of problems, by using anonymous / Named volumes when you changed your source code on your host system it won’t reflect it to container until we rebuild the container again to create a new image.

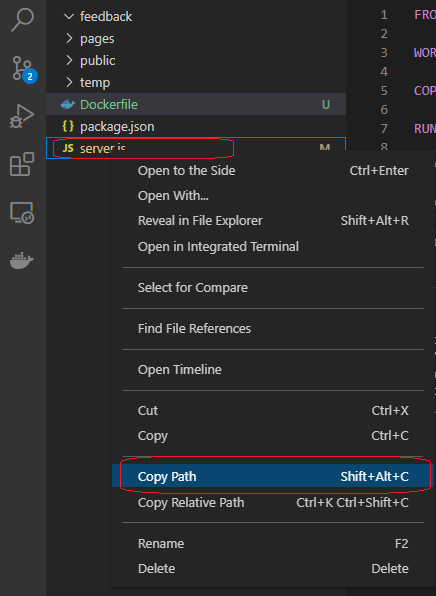
This is happening because the instruction that you are giving “COPY . .”, is not copying entire files to the image instead it will only copy the snapshot of that project files to the image. So whenever you are making changes it won’t reflect at the image is read-only file system. This problem will be solved by bind mounts, as in bind mounts we are directly attaching our host folder to the container folder, so whatever we changed on your host directory it is reflected on your running container.

Now in order to mount the bind mount volume, you should not mention on Dockerfile, instead you need to provide the path to docker run command by using the option -v <Some path at your host>:<App container file system>

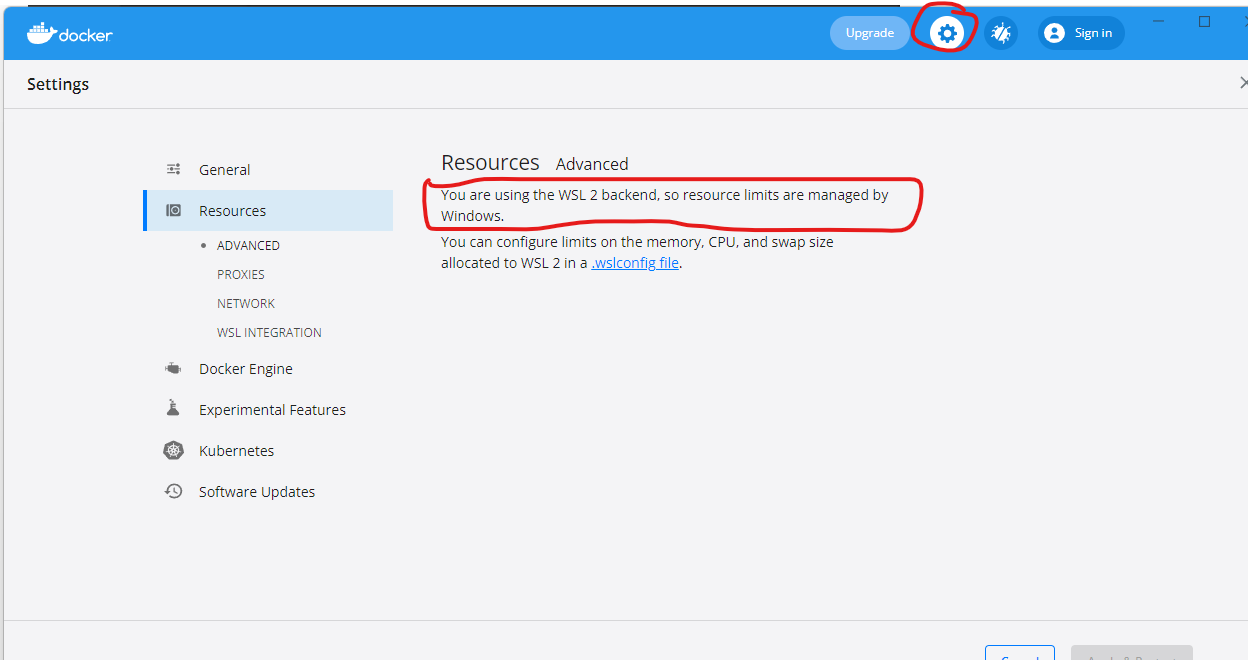
You can get the path of your code on your system by right clicking on server.js on visual studio and select the option “Copy path” as shown below.

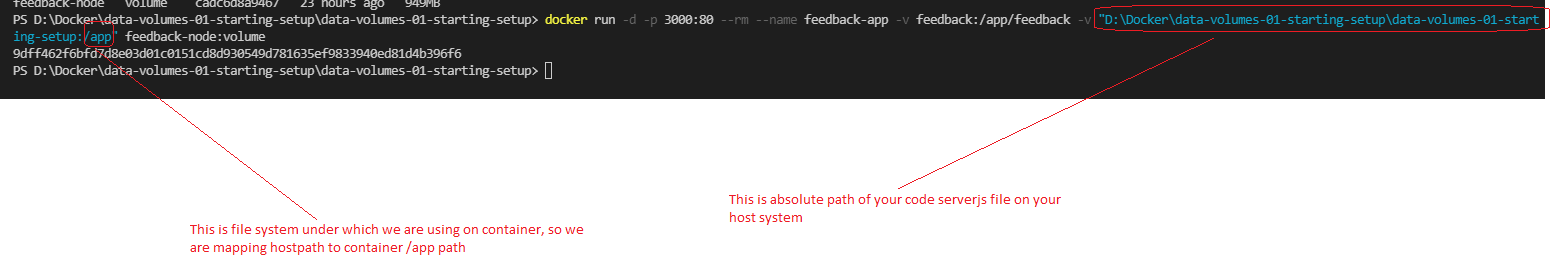
Note: if you have any special characters in the path please mention the path on quotes

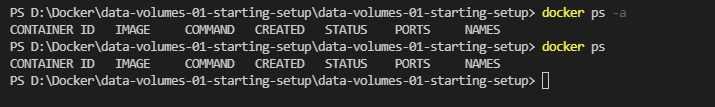
Here my server.js path is “D:\Docker\data-volumes-01-starting-setup\data-volumes-01-starting-setup”.



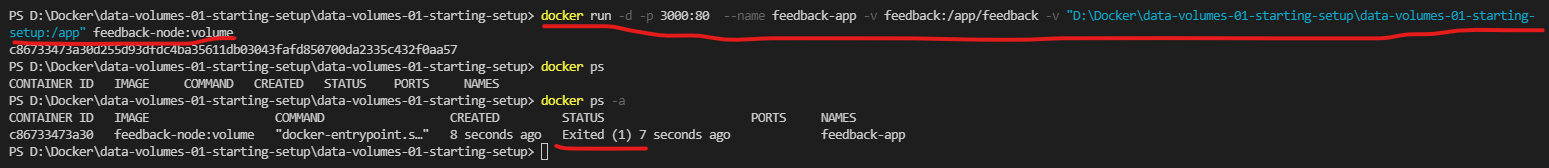
Note: If you using MAC machine, before making a Bind mount, you should make sure that docker has access to that bind mount. For click on Docker icon at system tray and select preferences. Once the settings are opened click on Resources 🡪 File Sharing and add the parent root folder that are using for bind mount. Here I am using windows and I am using docker with WSL integration I need not to be worry.

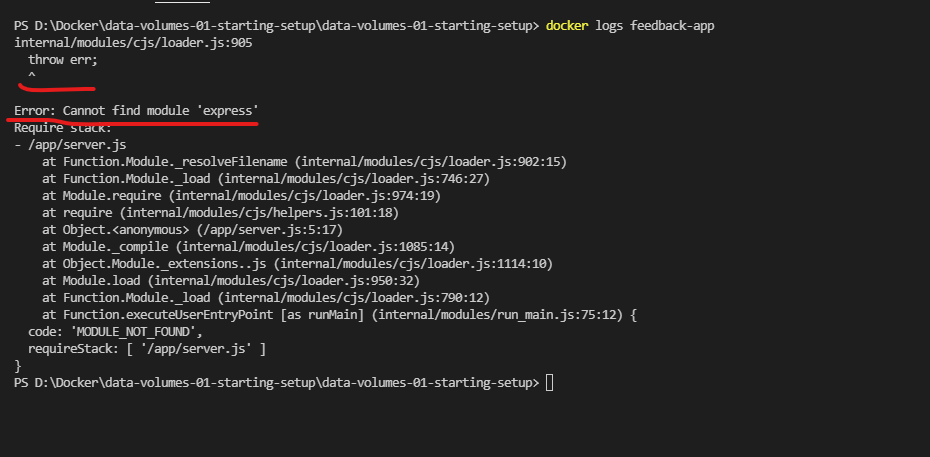






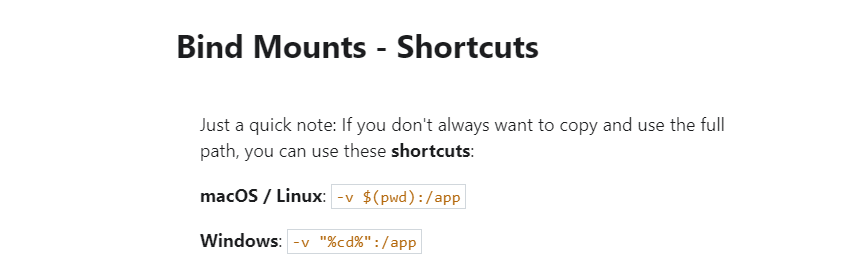
Looks like the container started and shutdown immediately, as we had used option --rm it deleted the container. So we will start the container by removing the option --rm and verify the docker logs.

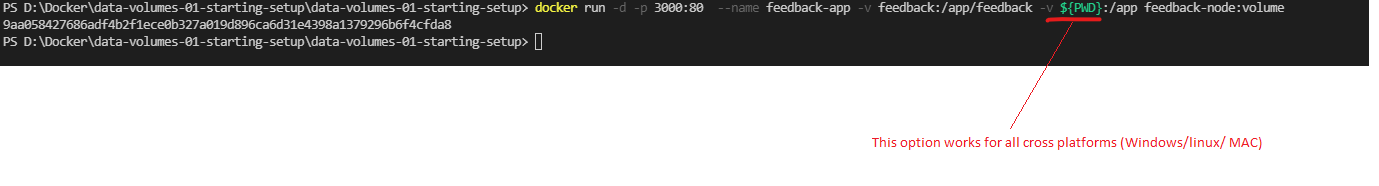




The above error states that, it even tried to bind mount and it failed while installing dependencies with the command “RUN npm install” on Dockerfile.

Let’s debug further why it is failing.





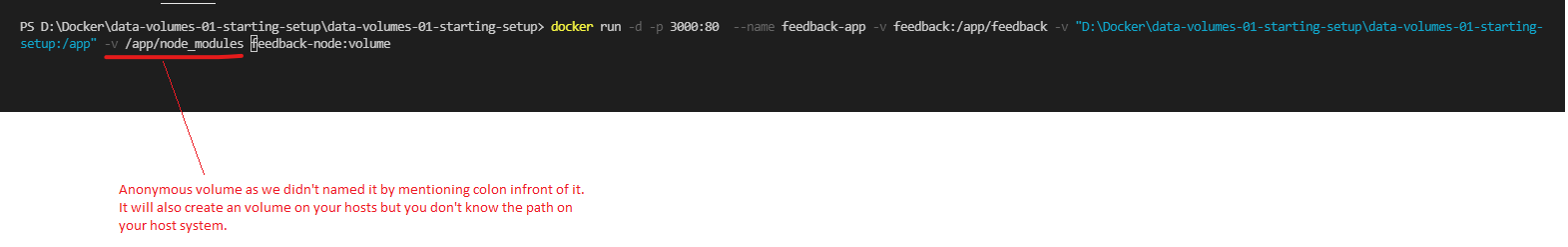
To understand what could be the problem, keep in a mind that everything on this folder on your host system “D:\Docker\data-volumes-01-starting-setup\data-volumes-01-starting-setup” is trying to override the folder /app inside the container.

As the file system from our host overrides to your container file system, all the COPY instructions mentioned on the Dockerfile will be invalid, because our host file system overrides to container /app, while starting NodeJS application, it requires some dependencies like express etc.., which was not installed on our local host system throws error when starting the NodeJS application on the container.

Now we had files inside of the container because of the COPY instructions and we had files on the Bind Mount point on our host system, good thing is it only overrides the host file system with your container file system and it won’t happened vice versa.

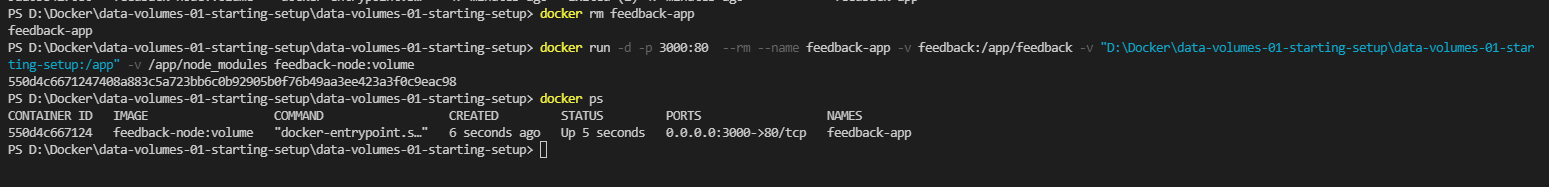
To solve this problem we need to inform to docker that certain parts of the file system should override with our local host file system on the container. This can be achieved by adding anonymous volume.

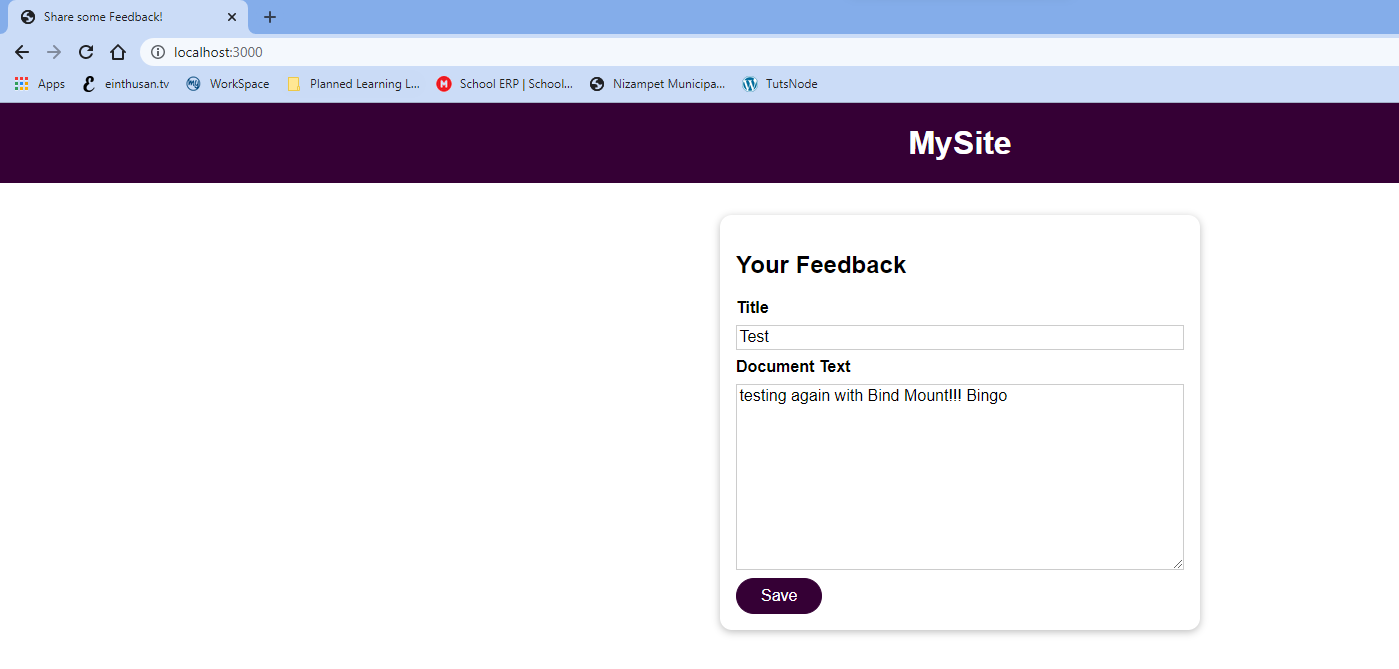
(Note: Anonymous volume can be created if you mention “:” infront of the volume, also this is same if we can mention -v [<volume name] to docker run or by writing VOLUME [<volume name>] on Dockerfile. We will prefer to use -v option as it uses on run time and no need to rebuilt)

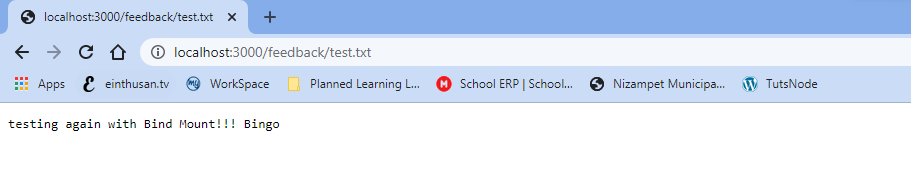


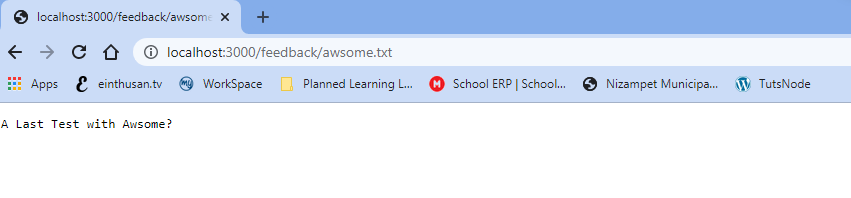
So here have a clash as Bind volume to /app and anonymous volume to /app/node\_modules, in case docker will simply follow the longer path and here the winner is /app/node\_modules instead of short path /app. So as the longer path wins while running the instruction “RUN npm install” it will install take the anonymous filesystem first with all the dependencies and later by mounting the Bind Mount folder, it overrides the whole host file system with the /app container file system and proceed further without any issues.

So simply stating that while building the image it uses the folder /app/node\_modules and later it will override with host file system with the Bind Mount

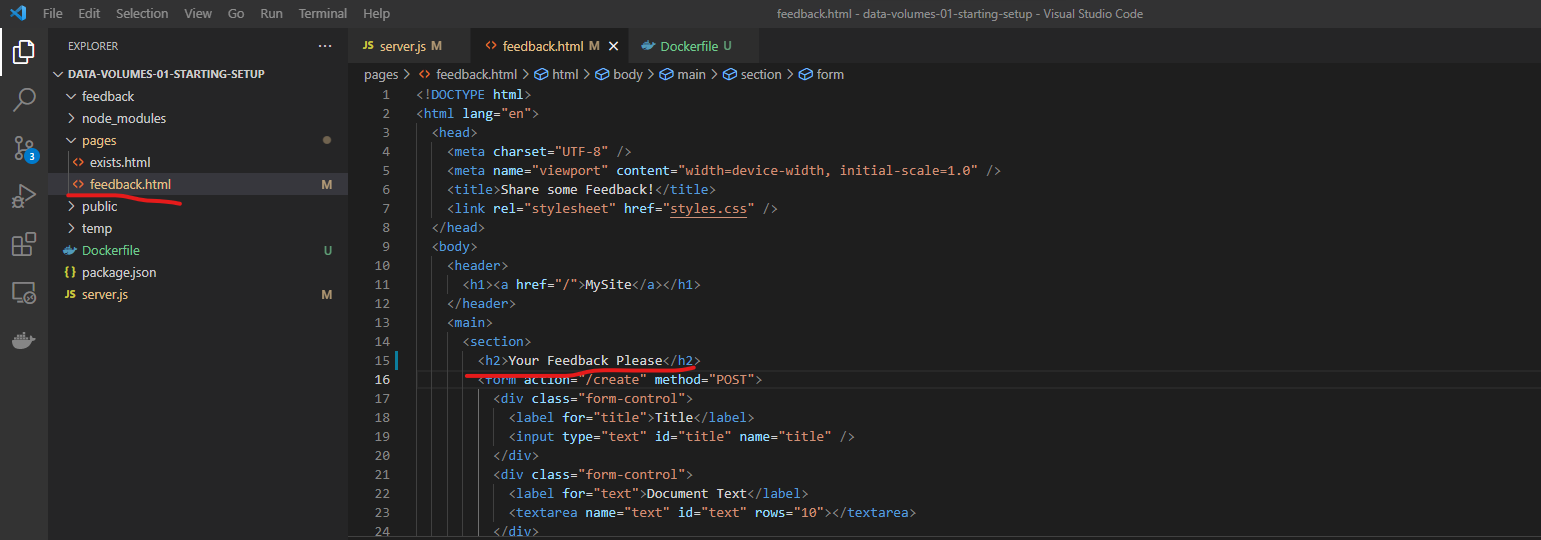


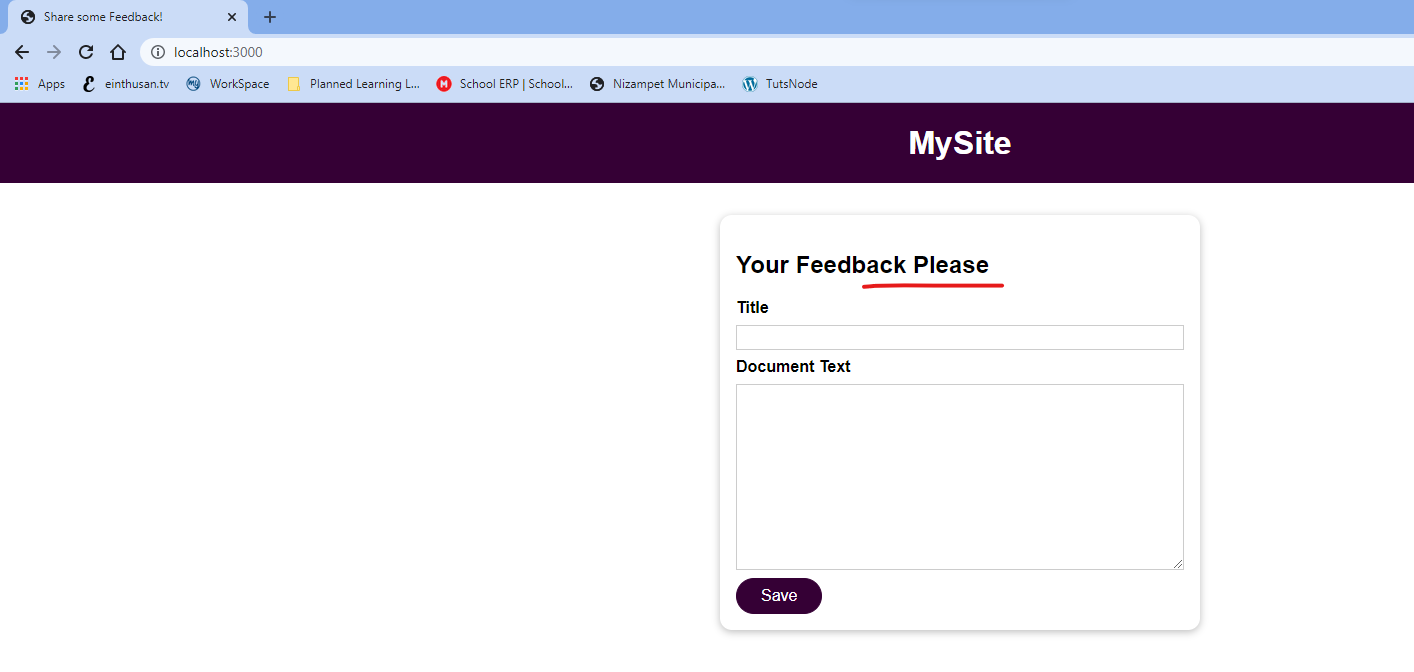




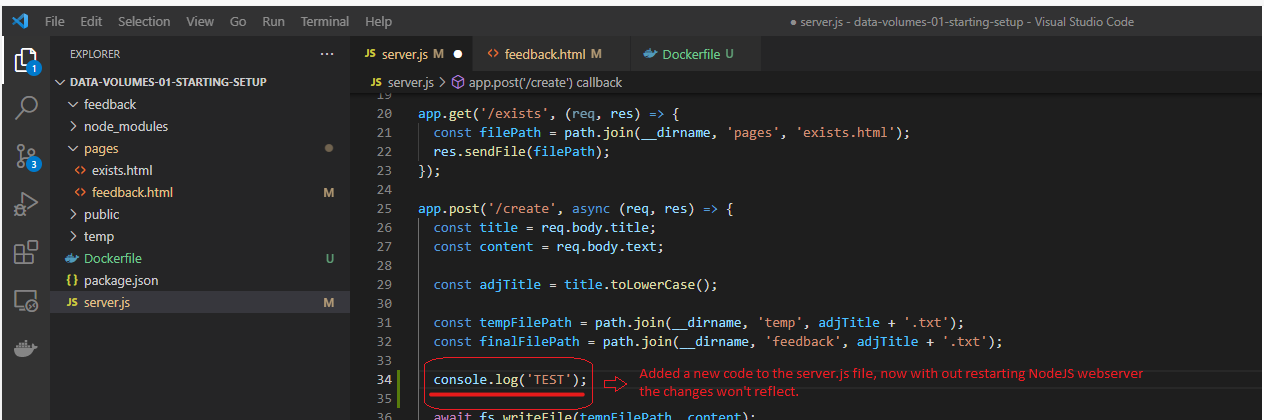


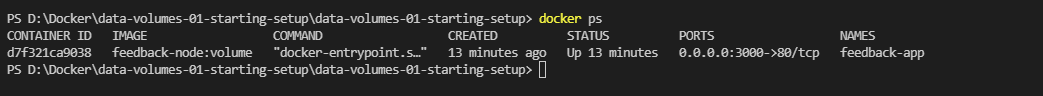
Now one additional benefit we got if we change html files on our host file system it will immediately effect on container without rebuilding.

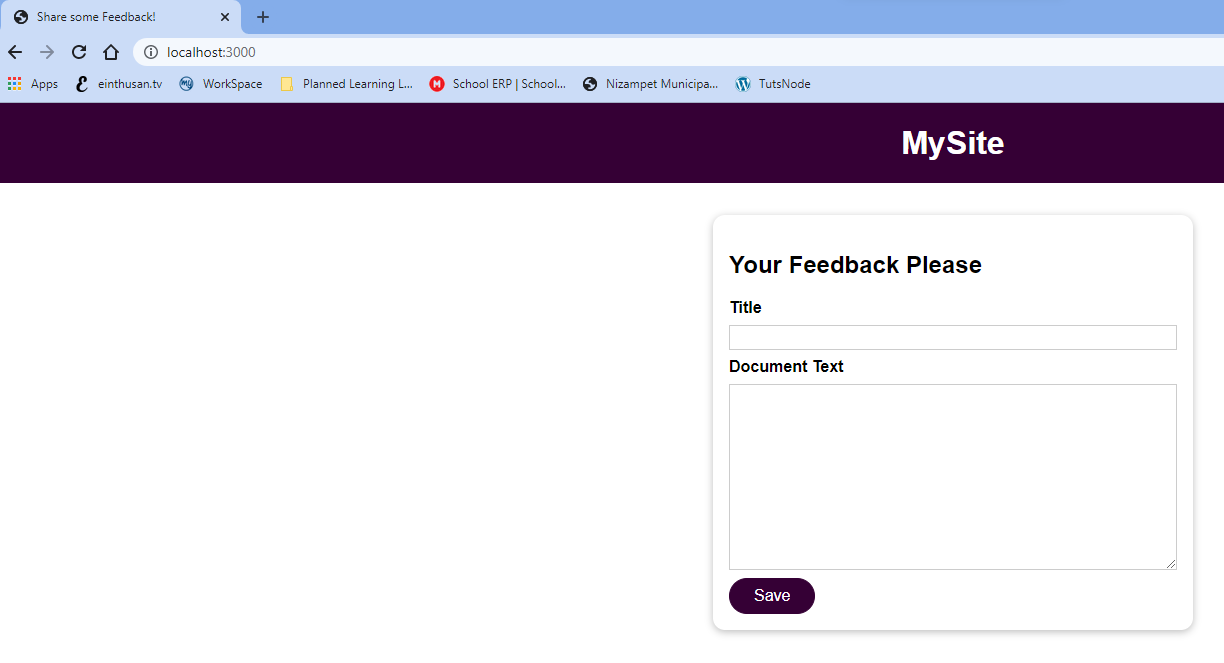




Now we have another problem, that is whenever we changes normal files like html and other files the changes are reflecting automatically. But whenever you change the source code of NodeJS server.js the changes won’t reflect because the Nodejs files are maintained by NodeJS server, we need to restart that process or restart the container to reflect it.



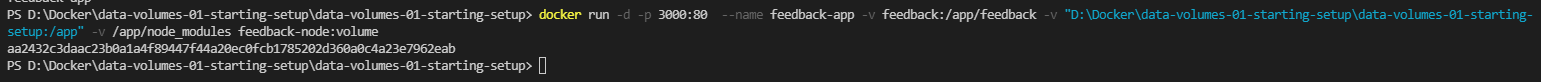


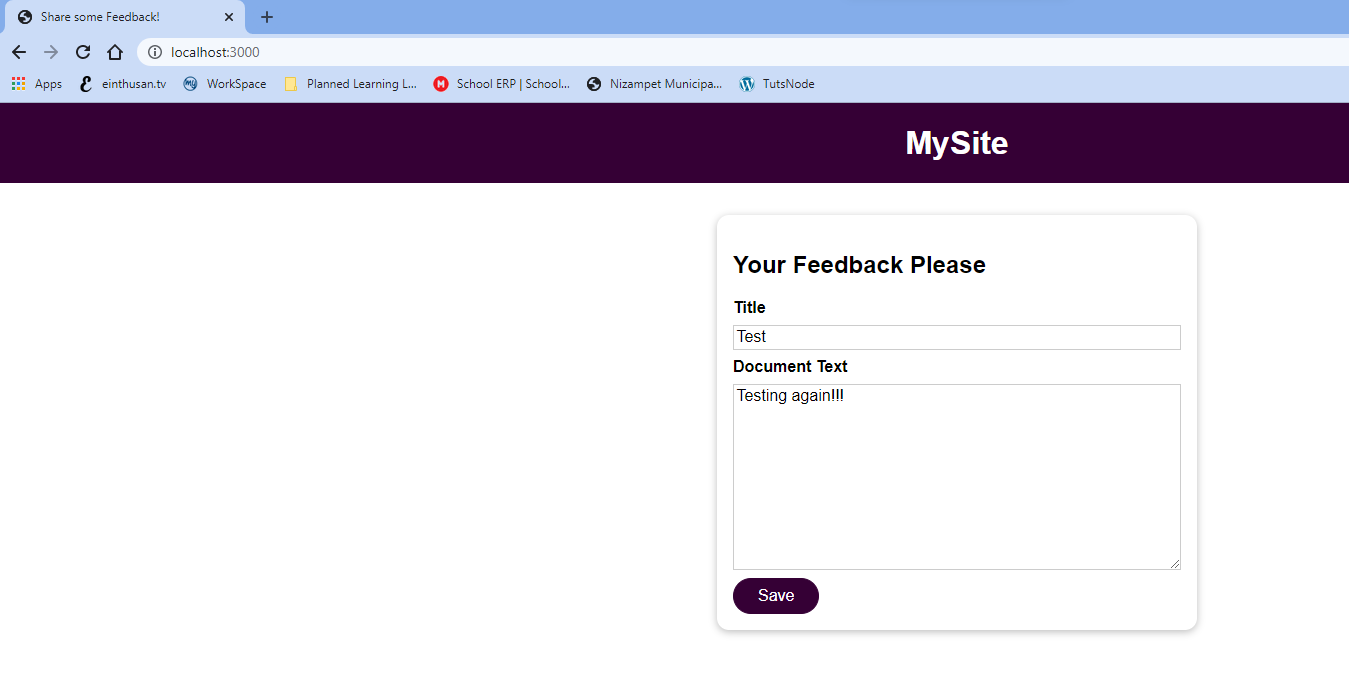


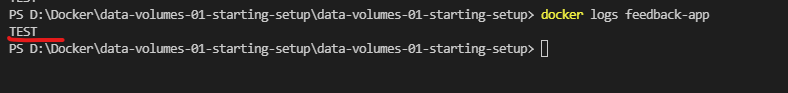
But when we verify the logs with docker logs, the changes won’t be viewed.



In order to reflect this, let’s restart the container once and re-run by using docker run command as I had used --rm option.

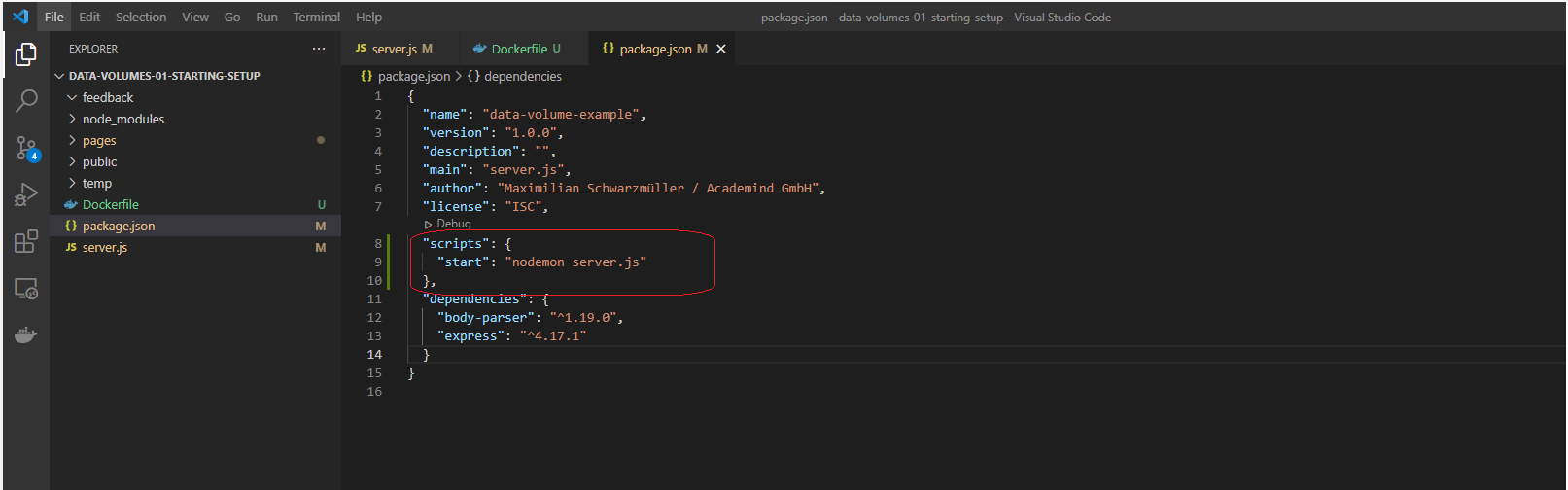


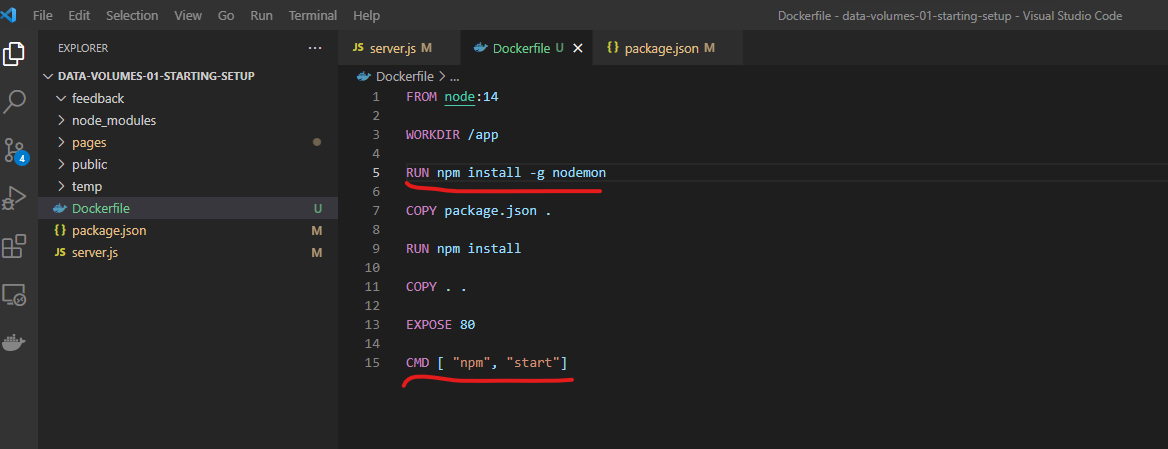


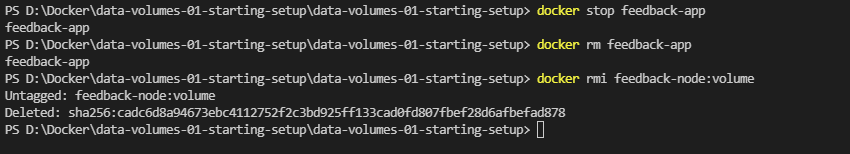


For NodeJS, there is extra useful package which will watch the filesystem and will restart if anything changed which is called nodemon.

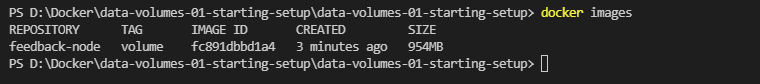
To install this dependency for nodemon and start the process, I need to modify the following changes.

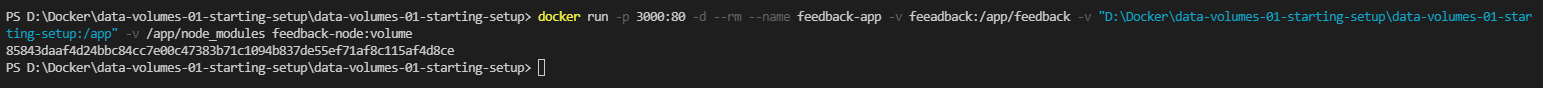


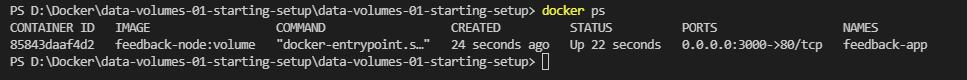


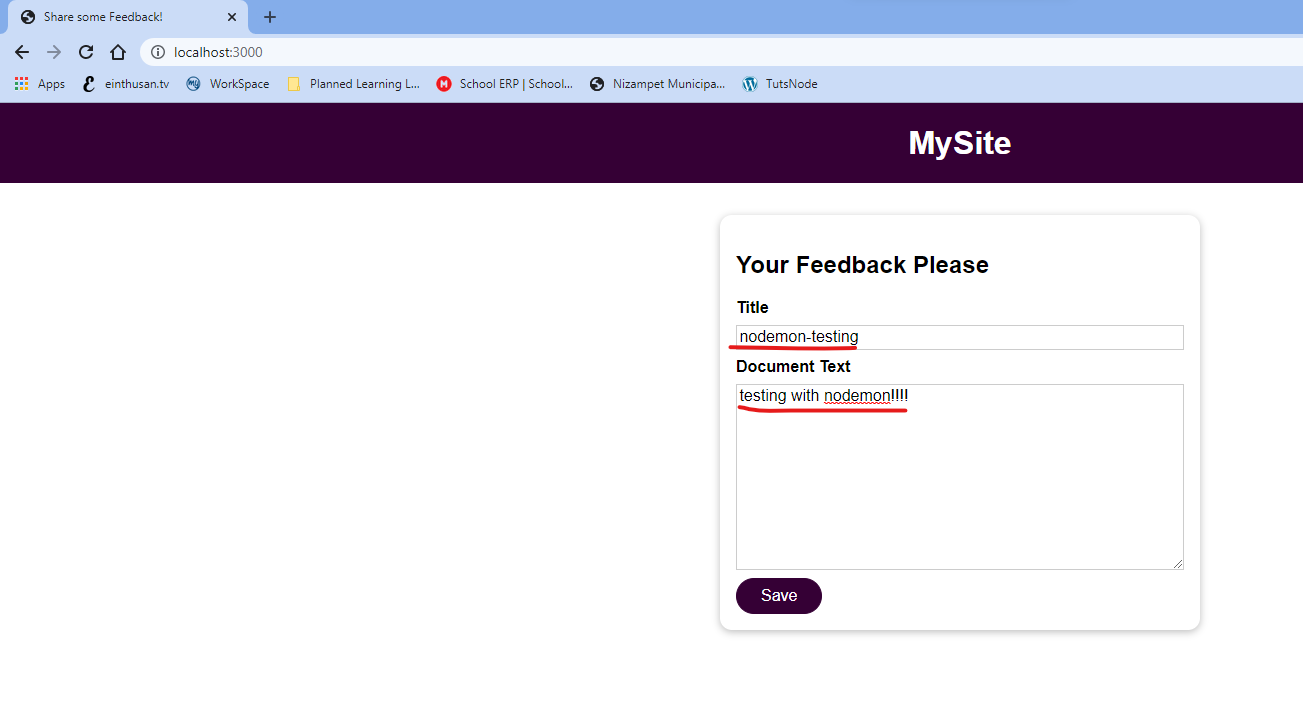


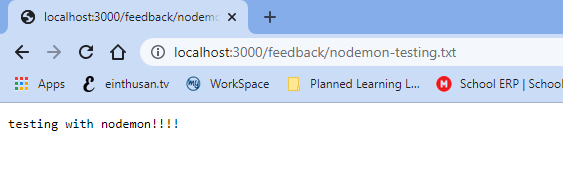


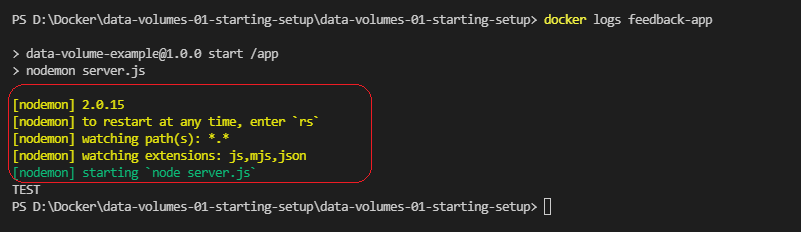




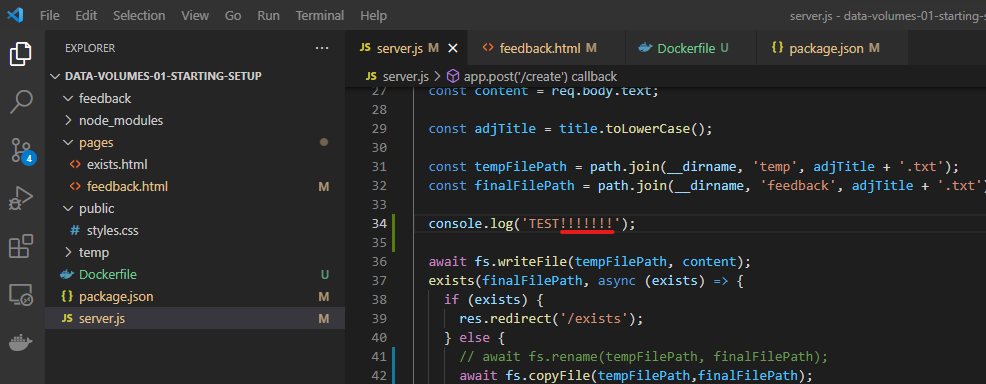


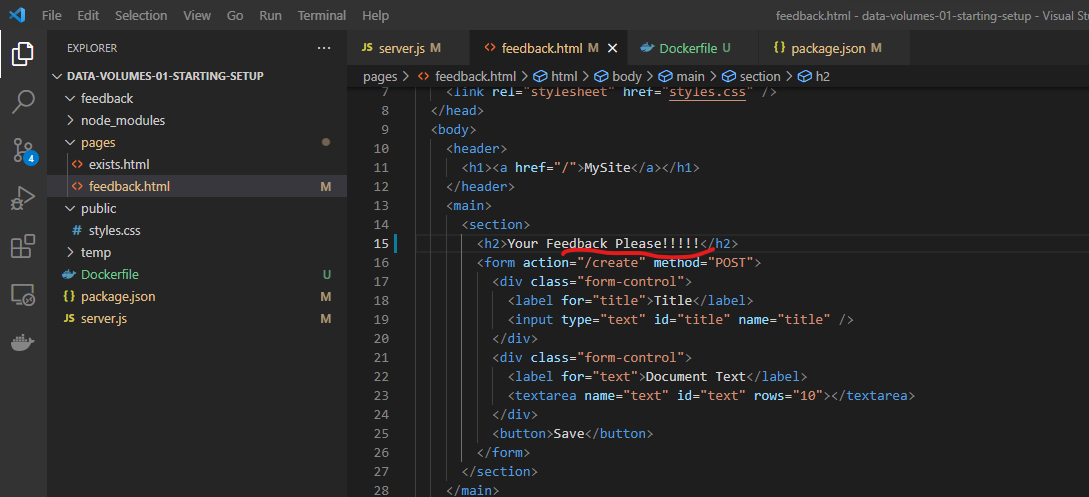




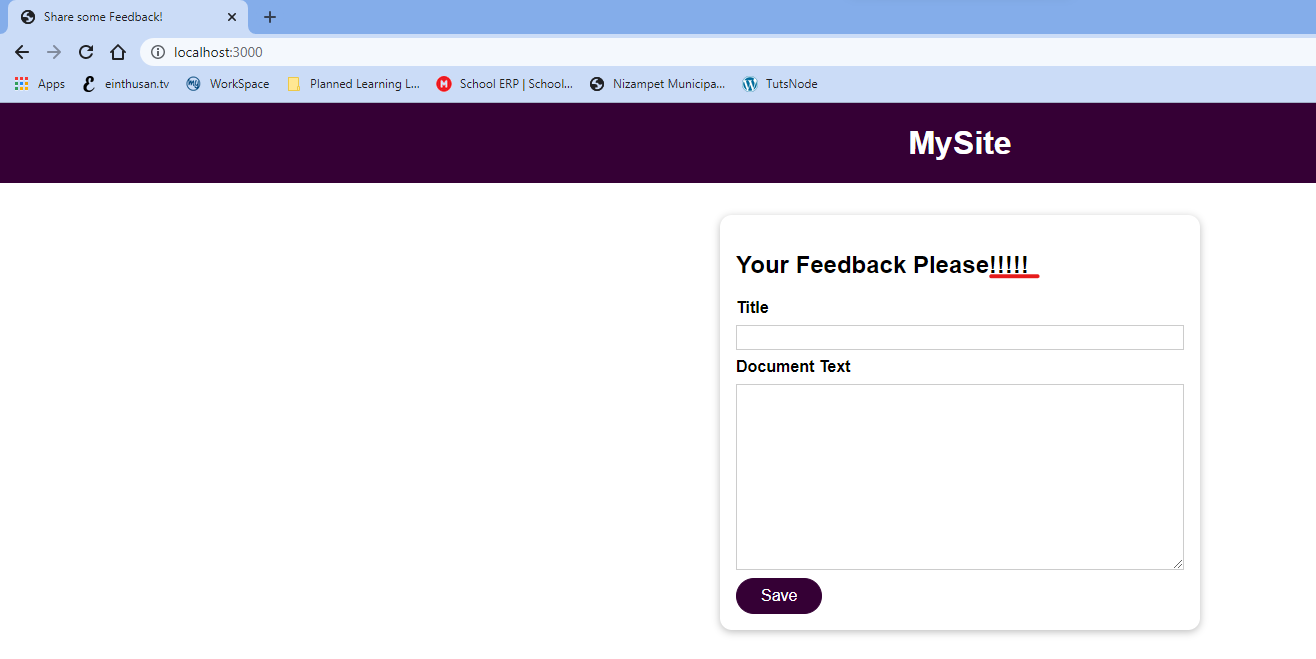


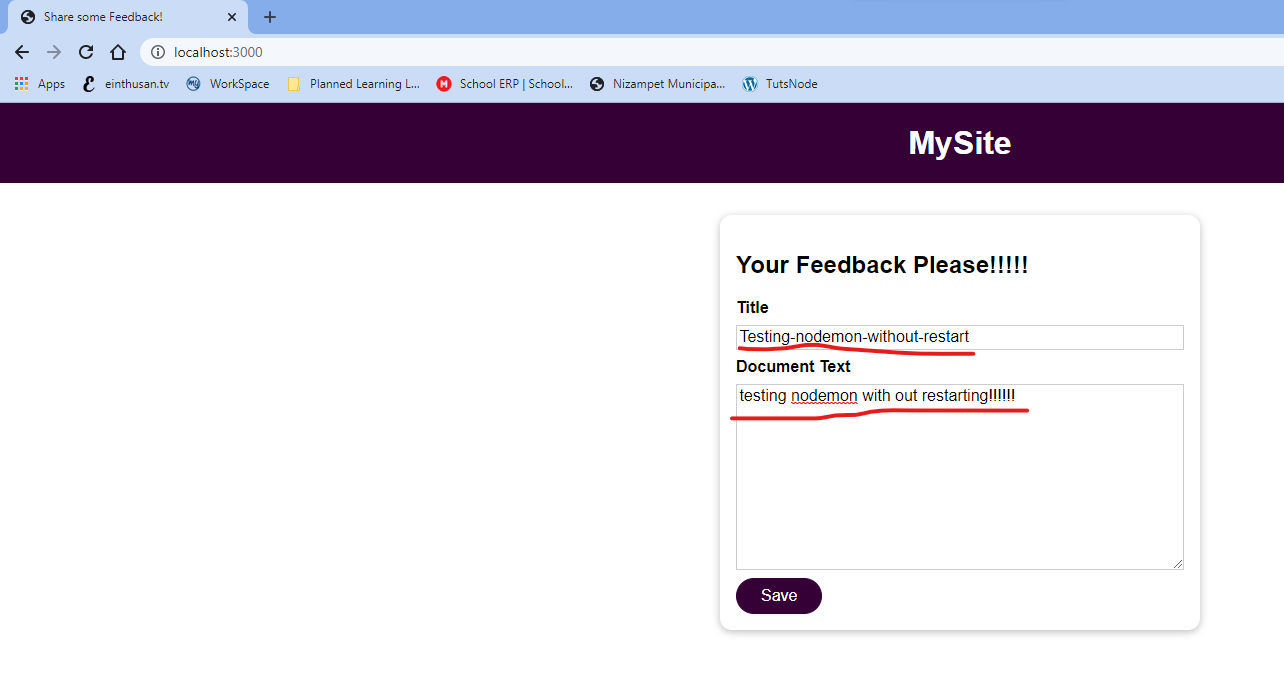
Now let’s change the code and html files without restarting the container it will start.

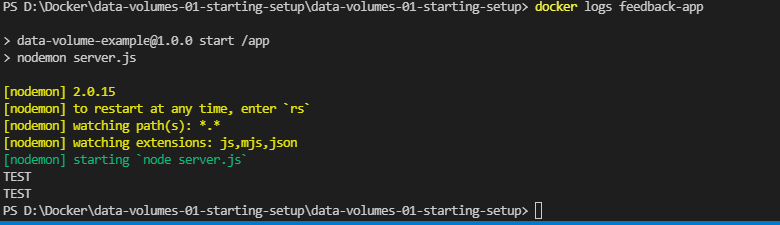


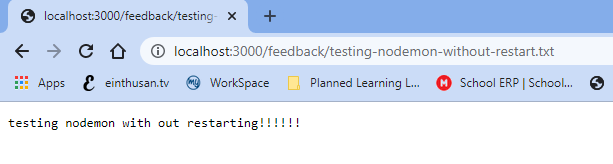


Now I am not restarting the container.









Here is a quick overview of volumes on the docker

Anonymous Volume

Bind Mount

Named Volume

docker run -v /app/data …

docker run -v data:/app/data …

docker run -v /path/to/code:/app/code …

In Anonymous volume, we don’t assign any name, docker will create some path at somewhere and also it managed by docker.

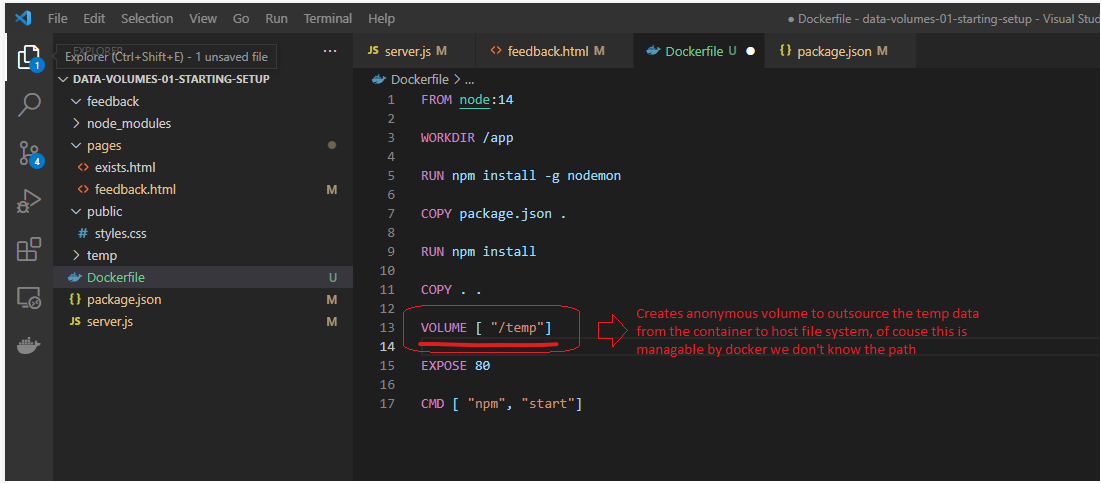
In Named volume, we mentioned <file system on host>:<file system on container>, this is also managed by docker itself.

In Bind mount we mentioned <Absolute path of file system on host>:<file system on container>, this volume can be managed by us.

Note: Anonymous volumes can be created by using the VOLUME instruction on Docker file or -v option at run time.

Also Anonymous volumes can be useful for locking in certain data which already exists in the container. They can also be useful for avoiding that disk data then gets overwritten by another module, that’s get something that anonymous volume can save the data. In addition anonymous volumes can still also creates a counterpart folder on your host machine, of-course that removed when the container is removed, that exists as long as the container is running. Which this doesn’t docker can’t store all the data as read-write inside the machine, it can outsource the data to your host machine filesystem, this can also help with performance and efficiency.

Here we can also add another anonymous volume for /app/temp to store temp files, this works to store the temp data and shared with outside of the container also.



Note: Named volumes and Bind Mounts can’t be created by using the VOLUME instruction on Dockerfile.

Note: On Bind Mounts, you can’t delete the data using docker commands, you need to remove the data on your host file system physically.

Can be re-used for same container (across restarts)

Can be shared across containers

Survives container shutdown / restart – removal on host fs

Location on host file system, not tied to any specific container

Can be re-used for same container (across restarts)

Can be shared across containers

Survives container shutdown / restart – removal via Docker CLI

Create in general – not tied to any specific container

Since it’s anonymous, it can’t be re-used (even on same image)

Cannot be shared across container

Survives container shutdown / restart unless --rm is used

Created specifically for single container

Named Volumes

Bind Mounts

Anonymous Volumes

There is another type of volume which exists is read-only volume, Up to now we had mounted the following volumes.

feedback:/app/feedback 🡪 Name volume , here we need this volume as read-write, because inside out container user feedback need to be stored here.

/app/node\_modules 🡪 This is anonymous volumes, which is used to lock the file system and will save from overwriting the host filesystem with container and support to install the dependencies. This also need to be read-write.

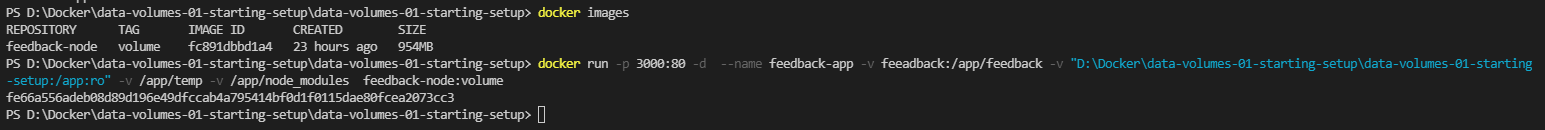
But let’s consider the Bind mount point, the idea is here on bind mount point we only change the source code or html files from host file system, the same will be reflected on container and the container should not be modify the filesystem, so In order to achieve this we can make this filesystem as read-only.

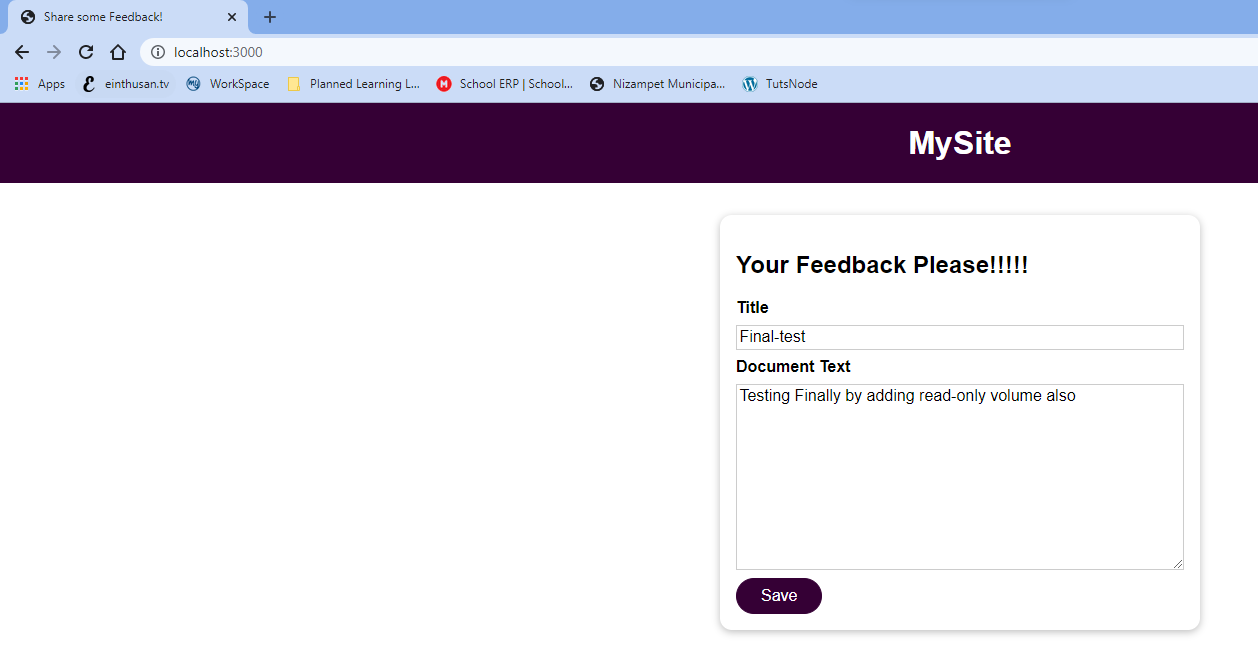
Note: by default volumes are read-write.

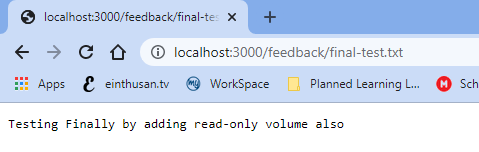
-v "D:\Docker\data-volumes-01-starting-setup\data-volumes-01-starting-setup:/app:ro"

As we made the bind volume as read only, the entire folder inside the project on host filesystem will mounted as read-only, then it is not possible that container can write the files to temp directory also.

To avoid this situation, we add another anonymous volume with the name /app/temp, we know the greater path always wins it will override that read only directory and write the temp data to store the temp folder.



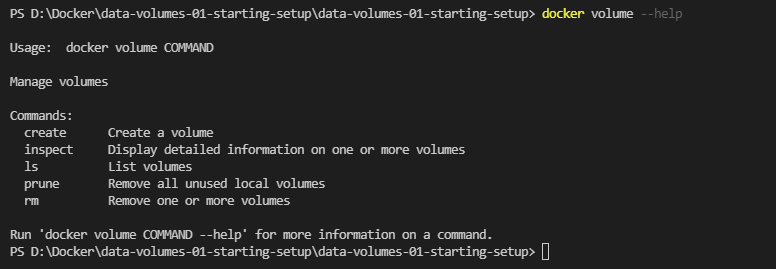




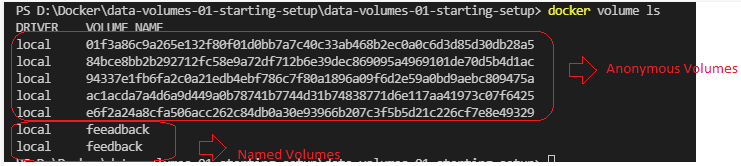
## Managing volumes

As we know docker will manage all the volumes for us somewhere at some path except the Bind Mounts. Bind Mounts can be managed by us.

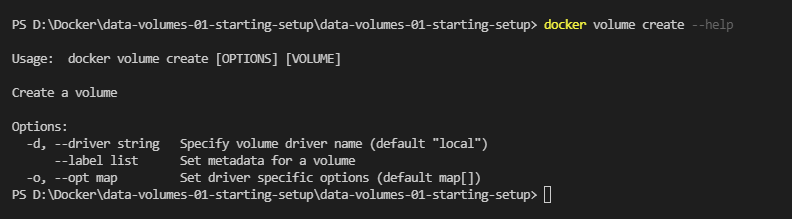
We will find that all the options that are resides with the volumes are listed below.

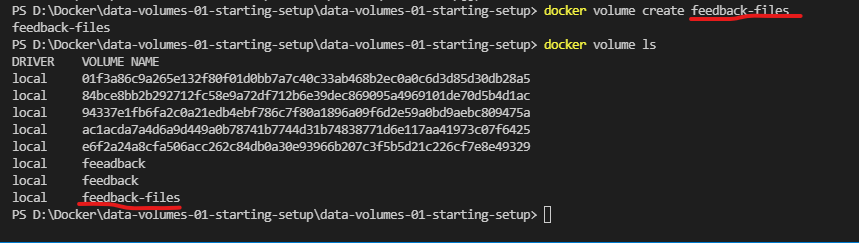


If we want list all the volumes that are present as of now, use the below command.



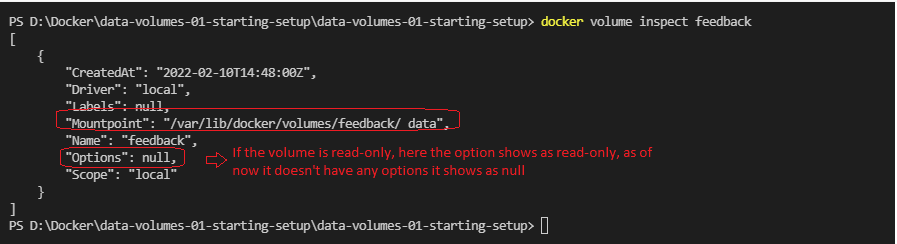
Normally docker will create the volumes if it doesn’t exists, but you can also create volume manually using the following command.



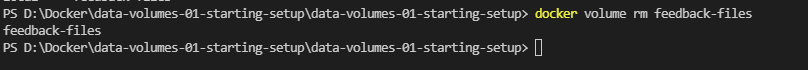


Now we can also use this created volume to another container also.

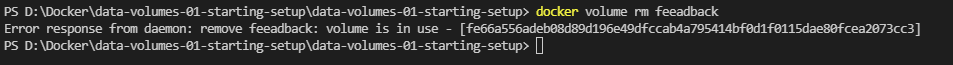
There are options exists with docker volume, one of important option is inspect, it will inspect the volume and provide useful information.



We can also remove volumes, by using the following command:



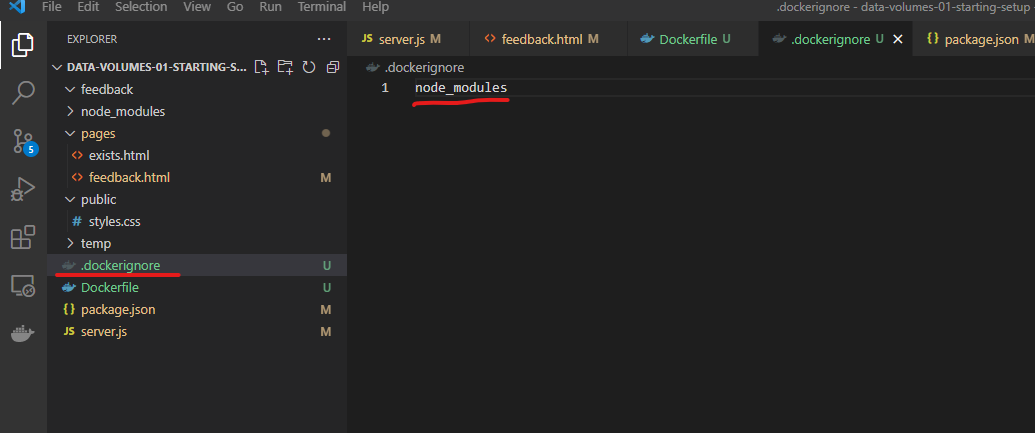
If I try to remove any volume which is mapped to running container it will throw an error.

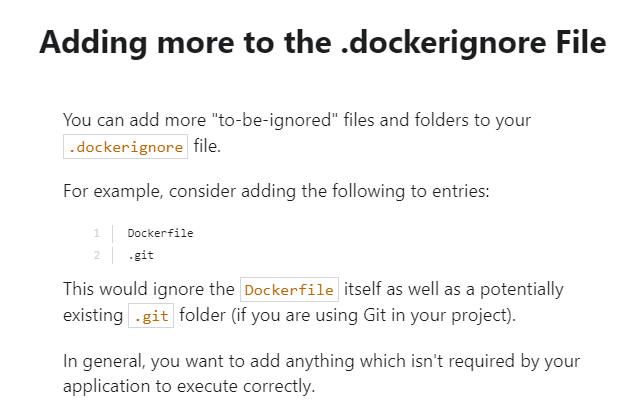


Now we are using the Bind Mount volume, which means all the host filesystem is mapped to container so everything on the volume will be available. So the Instruction “COPY . .” is necessary to copy the whole project directory to container?

The application will work fine without any issues, even if we remove that option on Dockerfile, but that is necessary these bind mount points we are mapped while developing the application, once it is finished if we mapped the code to a production server, there the mount point differs so it is always suggestable to make copy the snapshot of the entire code by using the option “COPY . .”.

Also we can restrict to copy some files from host file system to container by creating a “.dockerignore” file. If you mention any folder in that file the docker won’t copy anything while building the image.





## Docker Arguments & Environment variables

Today we will discuss about the Docker Environment Variables & Arguments and why those are helpful.

Docker supports build-time Arguments and runtime Environment Variables

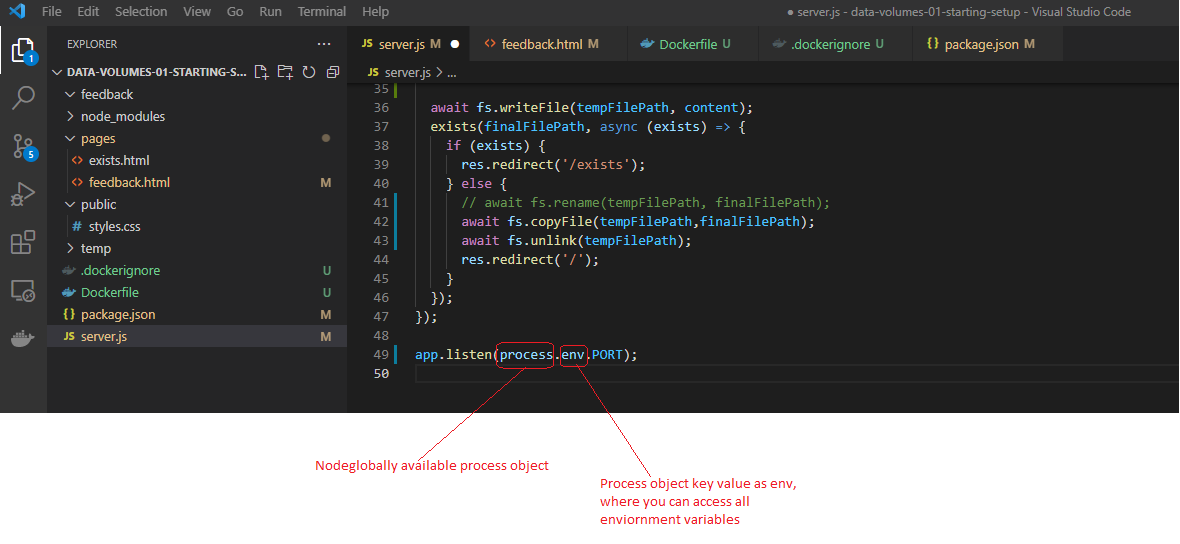
Arguments will allow you set flexible bits of data which are called variables in your docker file and you can use different values on certain Dockerfile instructions based on the arguments that are provided with --build-arg option when you run docker build.

Environment variables on the other hand are available inside of the Dockerfile but or also available in your entire application code. You can set them by using the option --env inside the Dockerfile telling docker that you are expecting this environment variable to be existent and then provide complete values using --env on docker run.

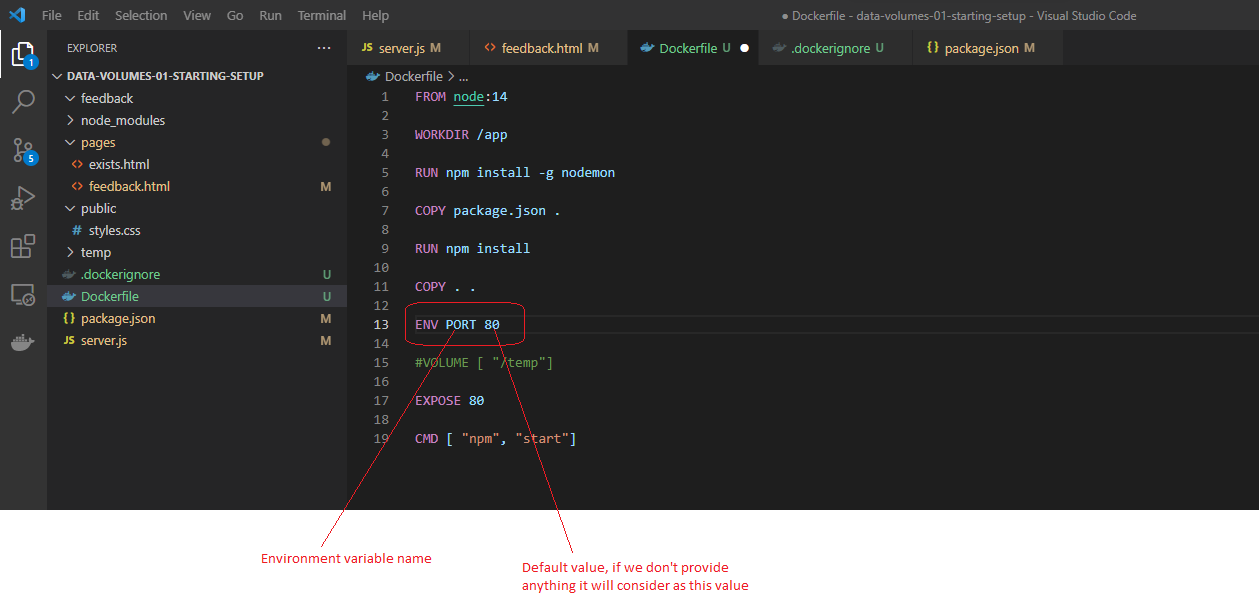
This Arguments and environment variable allows docker to build flexible images, because you don’t need to hard code anywhere and you can build an image dynamically while running container.

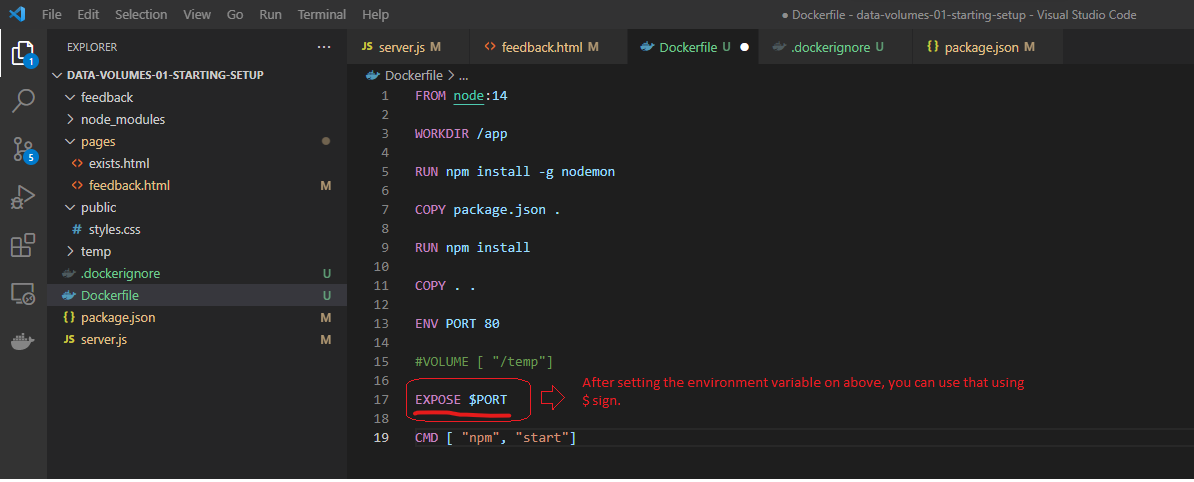
Now let’s make the Port as environment variable and provide those via dynamically via docker run.

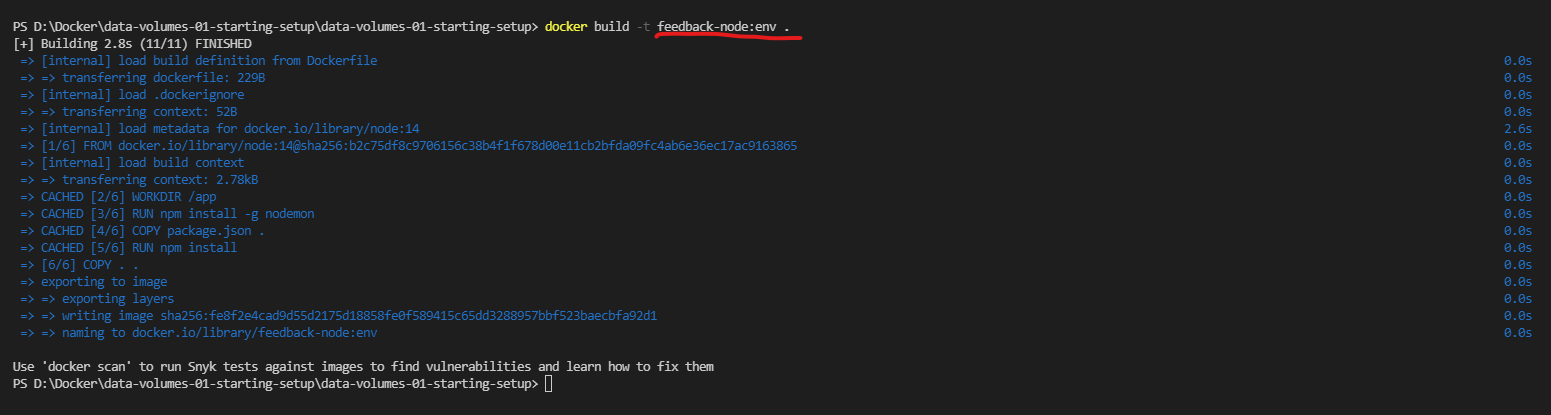
Also please note that Node application embraces the concept of environment variables and you access them in your node code on a globally available process object, on their you have a env key in their you can access environment variables you can set for.

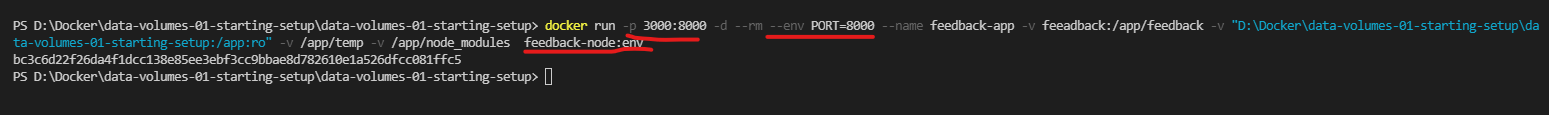


Now we can set a docker related environment variable inside the Dockerfile. You can set the environment variables in Dockerfile by using the ENV <name of the environment variable> <default value>



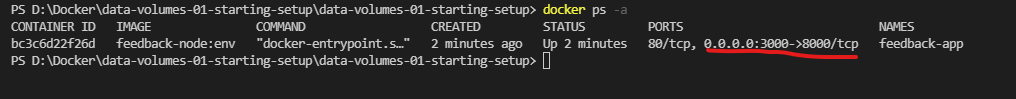


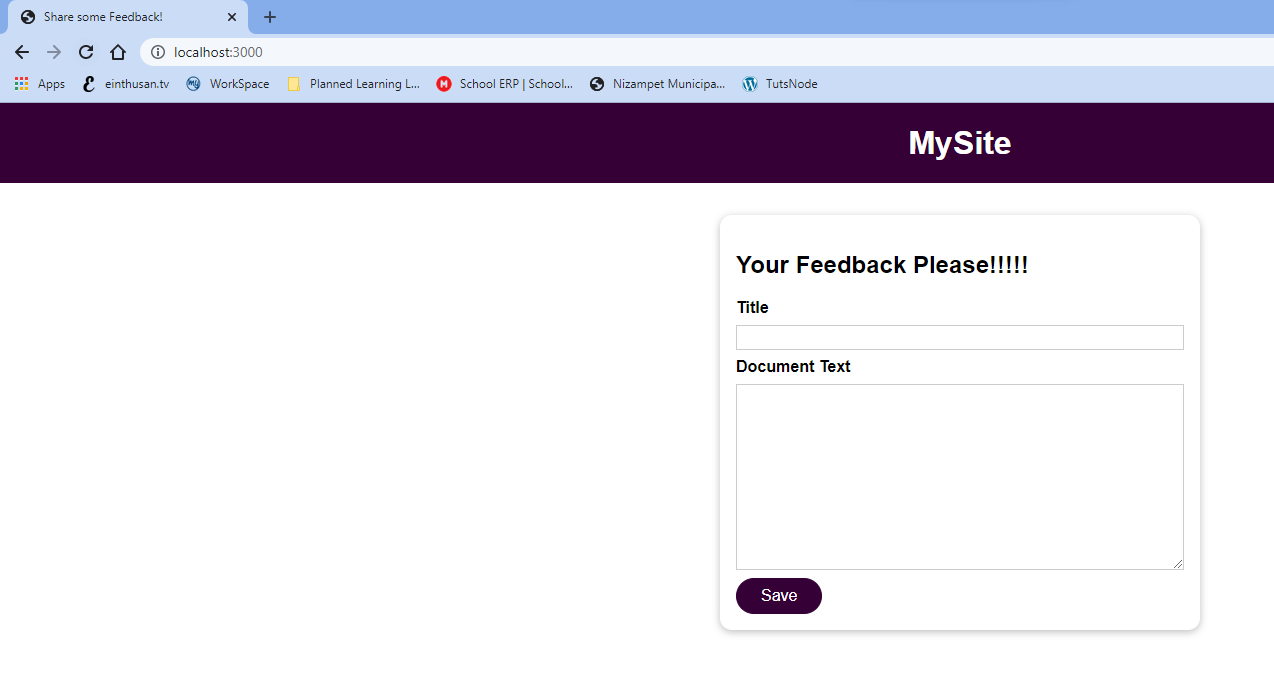




As you can see on the above command I had used the environment variable PORT while running docker run command itself, for me now no need to rebuild the image as we are taking the port from environment variables.

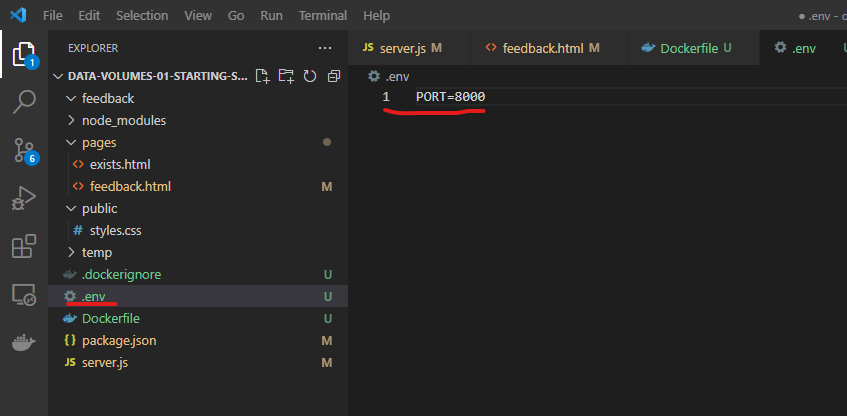
As we are asking docker to create a container using port 8000, we also exposing that port 8000 using -p option.

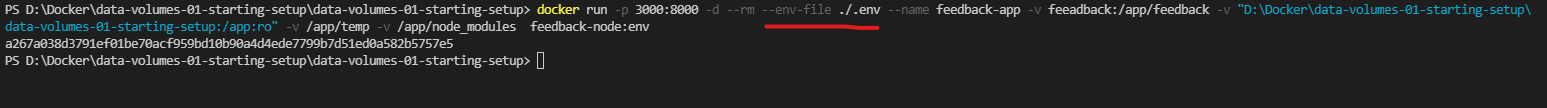


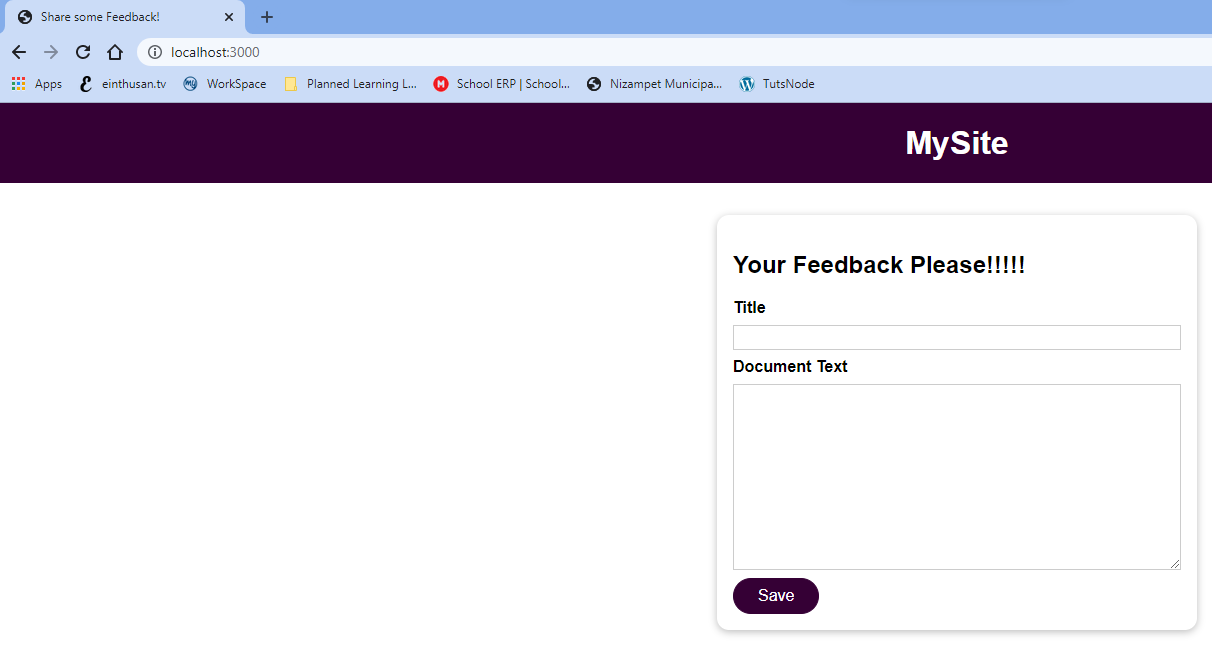


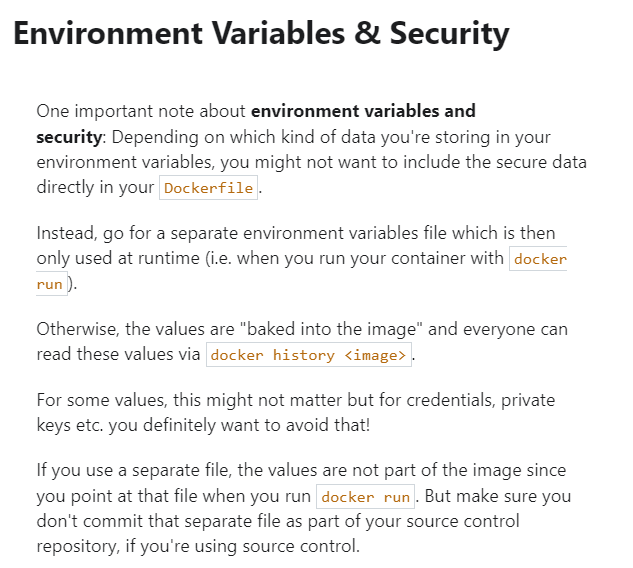
Note: Instead of providing option --env to docker run command, you can also use -e option, you can provide multiple -e options if you have multiple environment variables.

You can also specify the environment variable through a file, you can create .env file and mention the key value pair on that file. Once you create the file you can use this option --env-file to docker run command.



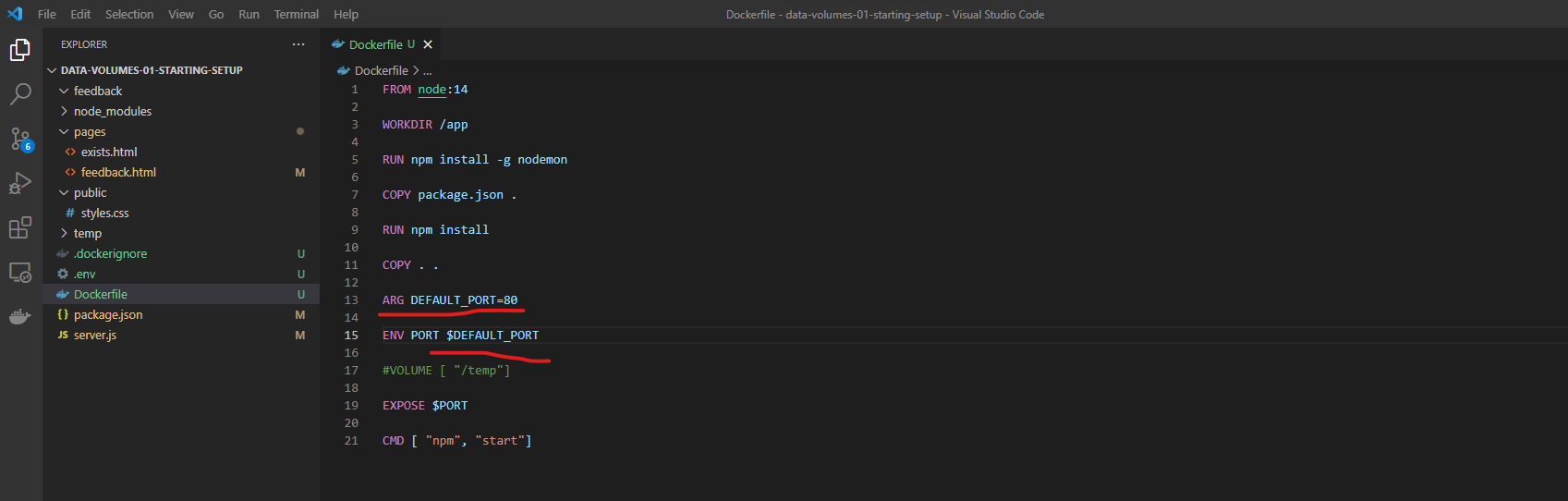


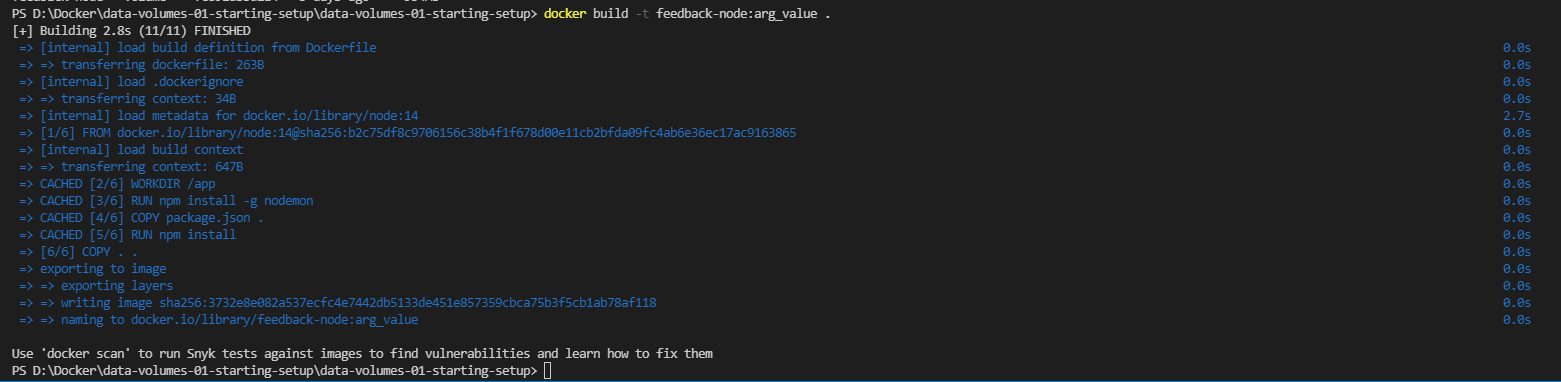




We can use arguments and supply those values while building an image, by providing arguments you will have a flexibility to change the values.

Now let’s consider while defining the port value as environment variable we defined default value, so we will change now to default by providing --build-arg.





You can also build by changing default value by providing the arguments at run time.

