Install Docker on Windows OS

Table of Contents

This document outlines the steps needed to install **Docker** and manage Docker containers with the help of Docker Compose and setup Docker cluster on Windows Operating system.

1. Overview:

Docker is an open-source software platform designed to build, deploy and execute applications more efficiently. Docker allows to separate applications from the underlying infrastructure to deploy applications quickly without the need of worrying if applications work on the deployed environment. Using Dockers methodologies for shipping, testing and deploying code, developers can significantly reduce the delay between writing the code and running it in production.

Docker is programmed in **Go** programming language. It was first released in **2013 by Docker, Inc** to work on **Linux** platform and later, it was extended to offer support on other platforms including **Microsoft Windows** and **Apple MacOS**.

Docker uses **containerization** technology using which one can develop and package the application along with all its dependencies into a standardized unit called "container". Docker containers are light-weight, portable and isolated from the underlying infrastructure so that they can be easily shipped and run consistently on different platforms such as Linux, Windows and macOS and across different locations including on-premises, public cloud and private cloud.

Docker is not a or does not work like a Virtual Machine:

- **Virtual Machine** is a system like a computer having a fully copy of operating system, application, binaries, libraries etc. Virtual Machine uses **hypervisor** technology which allows VM to run a guest operating system with virtual access to host operating system resources. VMs are heavy in nature since they have full OS with its own memory management with the overhead of virtual device drivers.
- **Container** is not a VM since it does not require a separate OS and is deployed on a physical machine with a host OS. Container consists of everything needed including application, binaries, libraries for application to work but it depends on the underlying **machine OS kernel** to run.

The key components of Docker include **Docker Host, Docker Engine** which comprises of *Docker Daemon, Docker API, Docker Client***, Docker Objects** which include *Dockerfile, Docker images, Docker Containers, Docker Storage* and *Docker Network***, Docker Registry, Docker Desktop, Docker Compose** and **Docker Swarm**.

Due to Docker's popularity, many cloud platforms such as **AWS** and **Azure** support Docker containers to package, deploy and run applications.

2. Docker Installation on Windows:

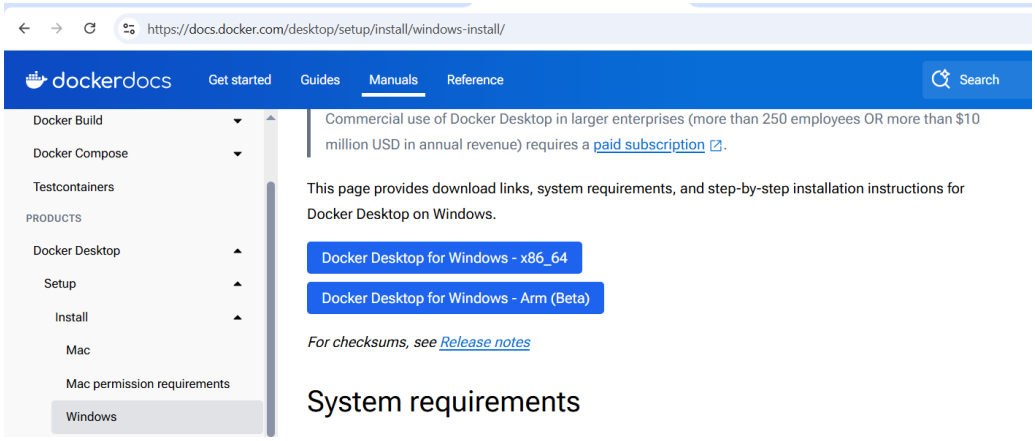
On a Linux host, Docker is installed directly on the operating system but on Windows, Docker is installed inside a Linux VM *(created using either Hyper-V or WSL2)* and Docker client is used to interact with Docker Engine.

Installing Docker on Windows will actually install the following components inside a Linux VM on Windows system.

- **Docker Server**
  - **Docker Engine** – Docker daemon, `dockerd`
  - **ContainerD** — Container runtime, `containerd`
  - **RunC** — Low level container runtime, `runc`
  - docker-init
- **Docker Client** – Docker CLI, `docker`

2.1. Install Docker:

Go to Docker Desktop Windows Install page and click on **the Docker Desktop for Windows - x86_64** button which downloads `Docker Desktop Installer.exe` file into **Downloads** folder.



Note: By default, `Docker Desktop Installer.exe` file installs at `C:\Program Files\Docker\Docker` location and this location cannot be changed from the GUI. As an alternate, install it from the command line by passing the installation directory and using the default WSL2 backend for Docker Desktop.

Open **Command Prompt** application in **Administrator** mode and run the following commands:

```
cd %USERPROFILE%\Downloads
```

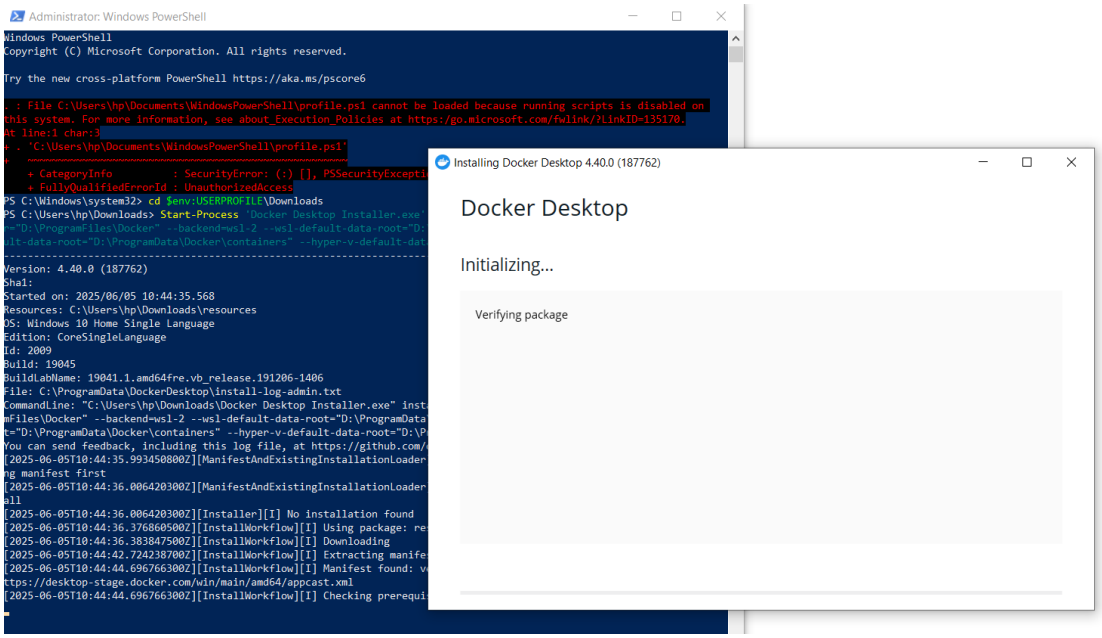
```
start /w "" "Docker Desktop Installer.exe" install --accept-license --installation-dir="D:\ProgramFiles\Docker" --backend=wsl-2 --wsl-default-data-root="D:\ProgramData\Docker\wsl" --windows-containers-default-data-root="D:\ProgramData\Docker\containers" --hyper-v-default-data-root="D:\ProgramData\Docker\vm"
```

To run from **PowerShell** application in **Administrator** mode, use the following commands for Docker installation:

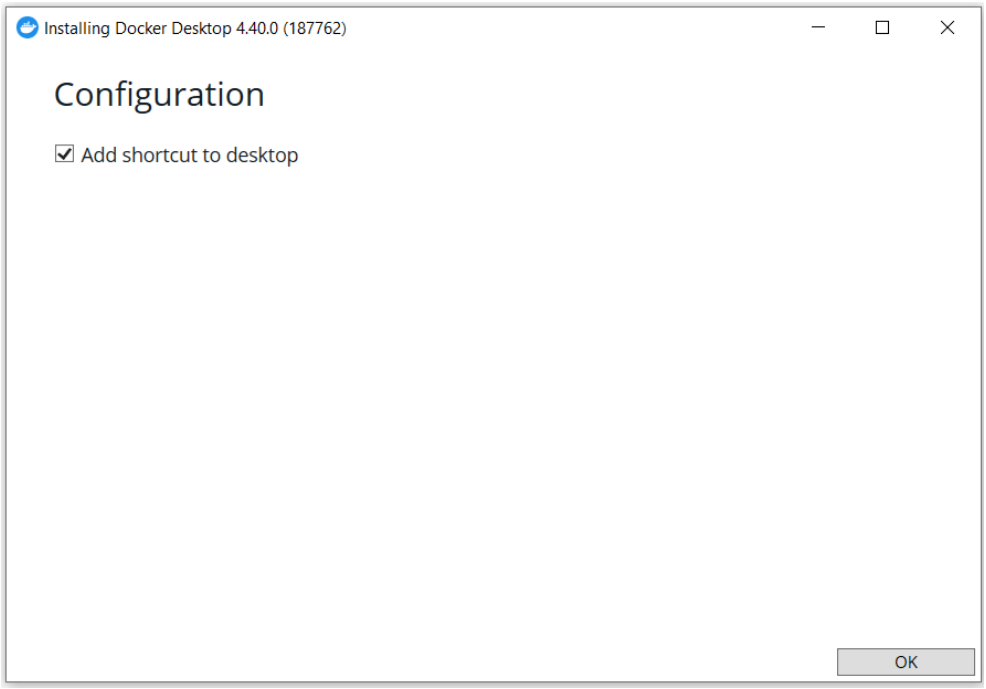
```
cd $env:USERPROFILE\Downloads

Start-Process 'Docker Desktop Installer.exe' -Wait 'install --accept-license --installation-dir="D:\ProgramFiles\Docker" --backend=wsl-2 --wsl-default-data-root="D:\ProgramData\Docker\wsl" --windows-containers-default-data-root="D:\ProgramData\Docker\containers" --hyper-v-default-data-root="D:\ProgramData\Docker\vm"'
```

In the above commands, the installation directory is specified as `D:\ProgramFiles\Docker` to install at this location and backend as `wsl-2` to use *(even if `--backend` flag is not specified, it uses `wsl-2` by default but we can also specify `hyper-v` or `windows` values to use a different backend)*.

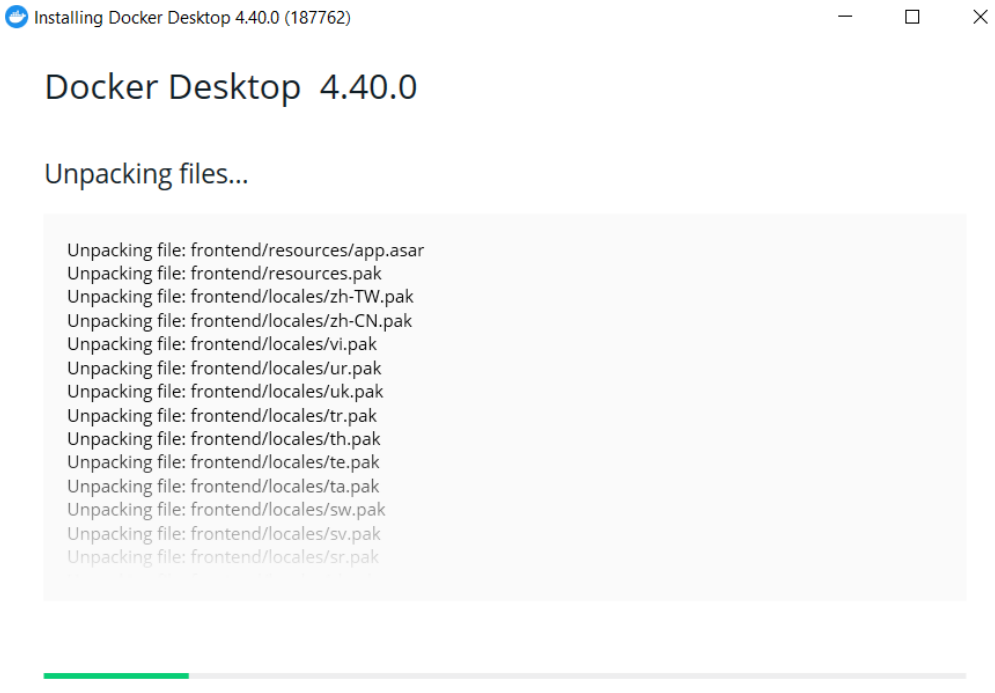
The installer then starts extracting and checks prerequisites which opens the installer wizard.



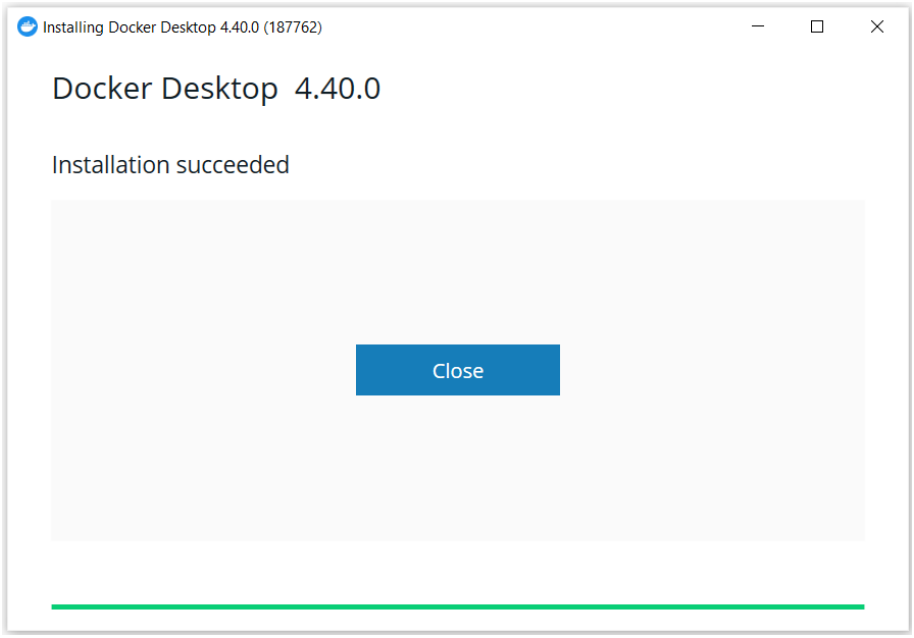
Once all prerequisites are validated, it shows the **Configuration** page where select the **Add shortcut to desktop** checkbox and press **OK** button.



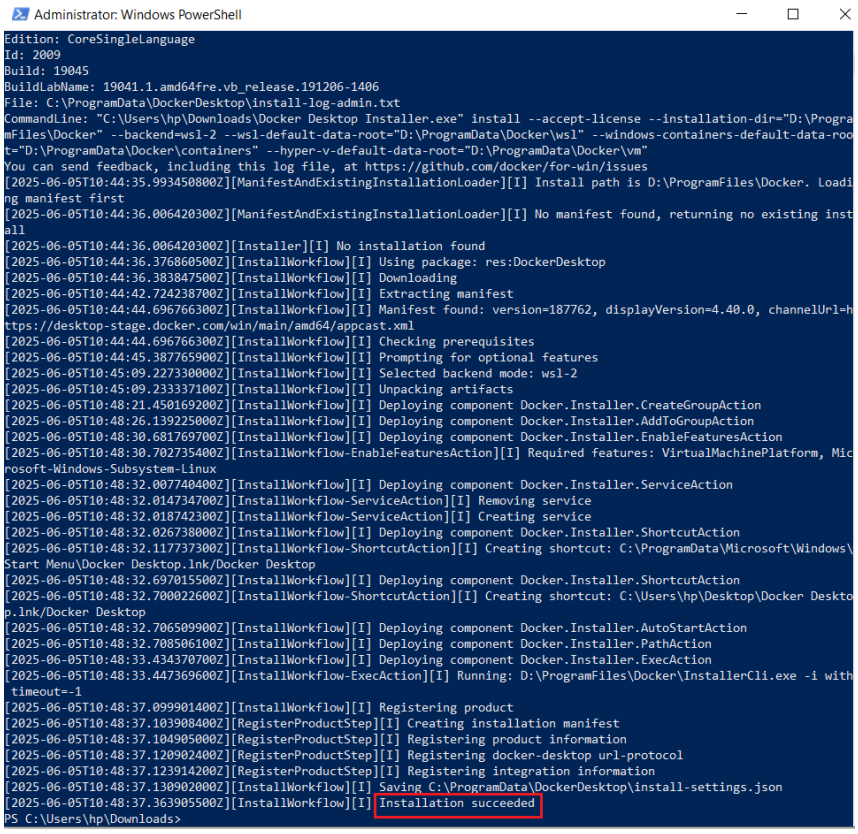
Then, it starts with the installation by unpacking files and takes few minutes to complete.



It gives **Installation succeeded** message once done. Click on **Close** button to exit the installer.

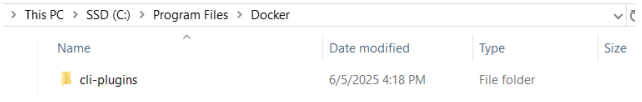


On the **Command Prompt** also, it displays **Installation Succeeded** message.

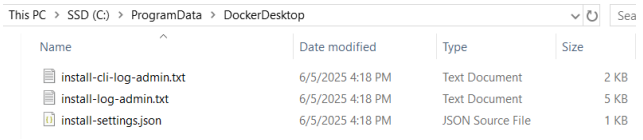


Note that though we installed Docker at `D:\ProgramFiles\Docker` location, some files are still being referenced from the default locations which cannot be avoided.

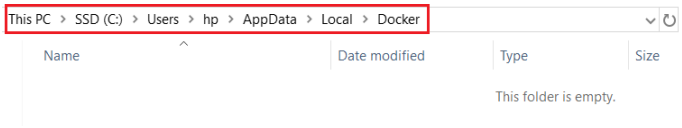
`C:\Program Files\Docker`



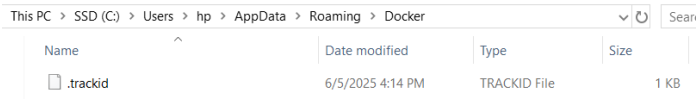
`C:\ProgramData\DockerDesktop`



`C:\Users\<username>\AppData\Local\Docker`



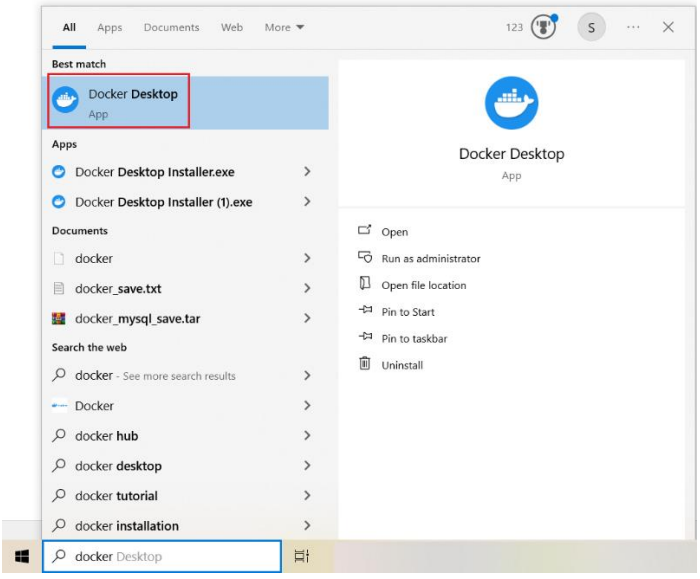
`C:\Users\<username>\AppData\Roaming\Docker`



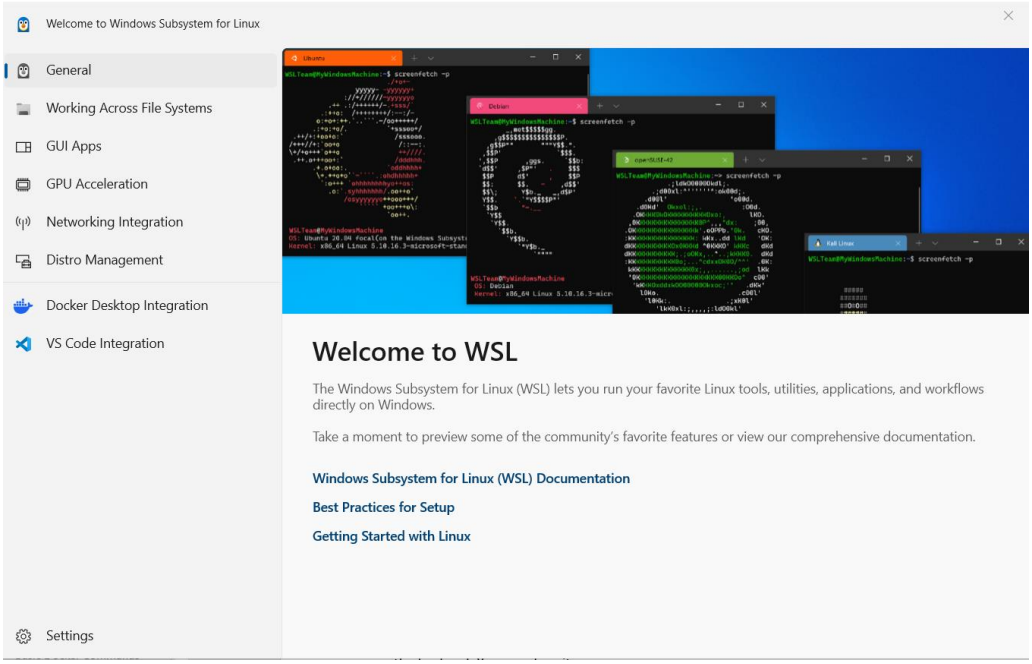
2.2. Start Docker Desktop:

After Docker installation, Docker Engine does not start automatically. **Docker Desktop** application needs to be started which automatically starts the Docker Engine.

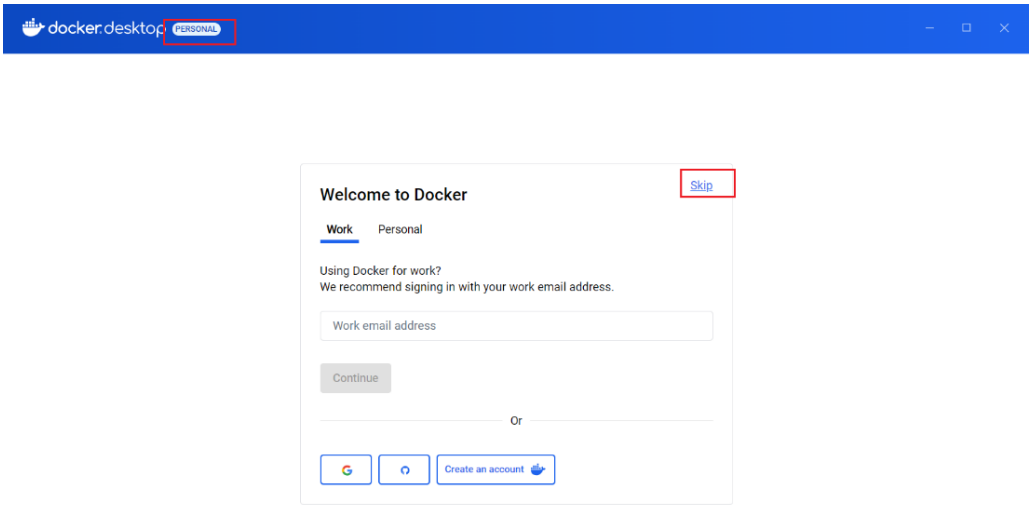
- Search for **Docker** in the search box of the task bar and select **Docker Desktop** application.



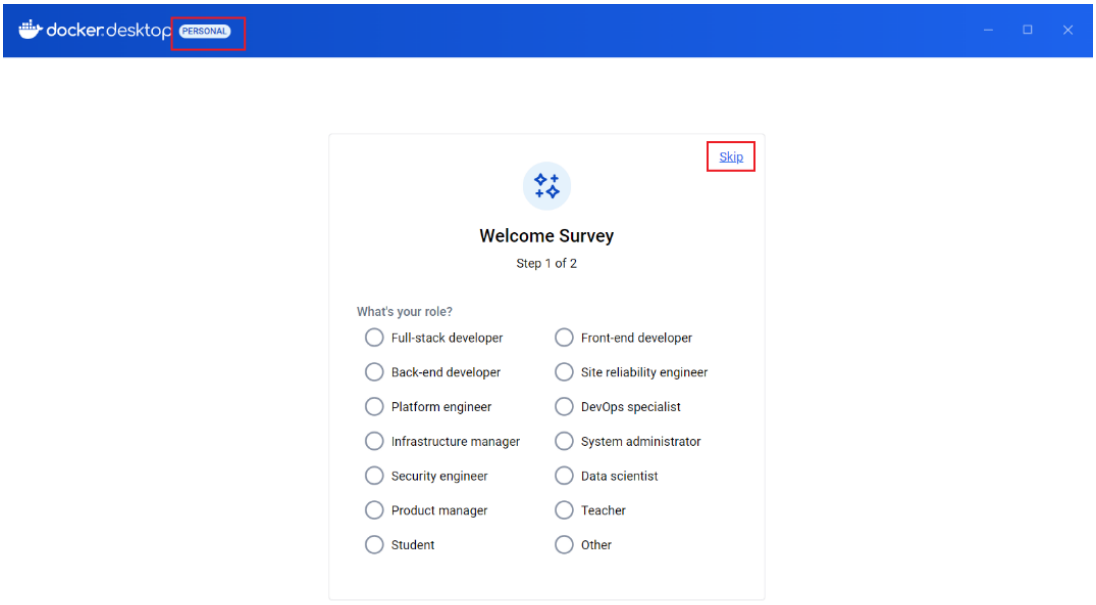
- In few seconds, it might launch **Windows Subsystem for Linux** (WSL) app since Docker uses this at the backend. This can be closed.



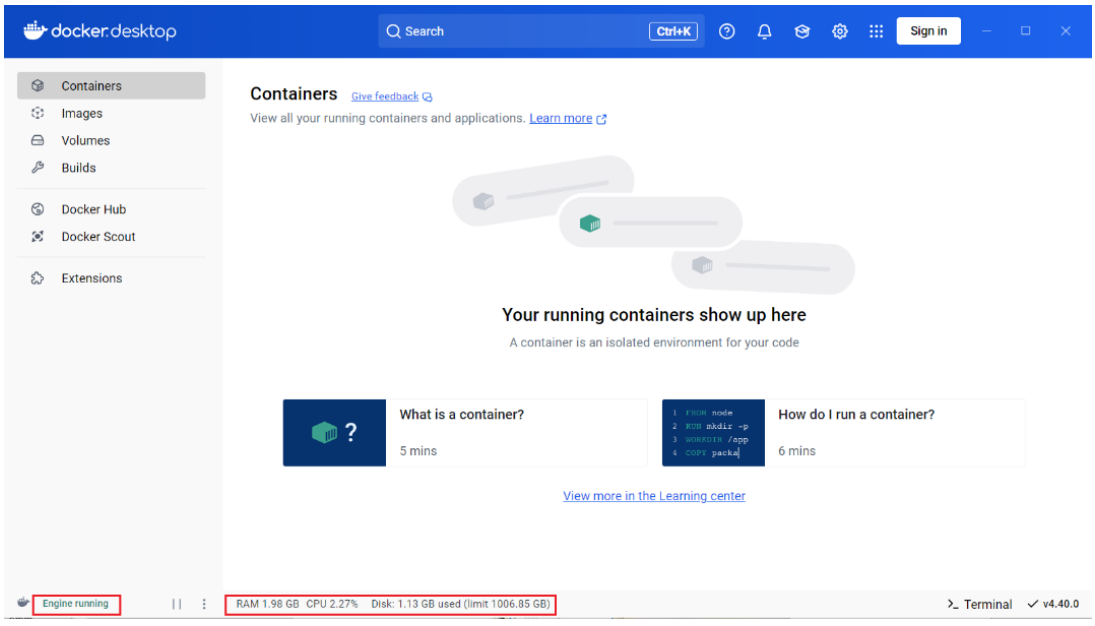
- In few seconds, it launches **Docker Desktop** application with a **Welcome** page. At the top, it displays the Docker Subscription Service Agreement. Since we are using Docker for personal use, it displays **PERSONAL** subscription. Click on **Skip** link to skip it for now.



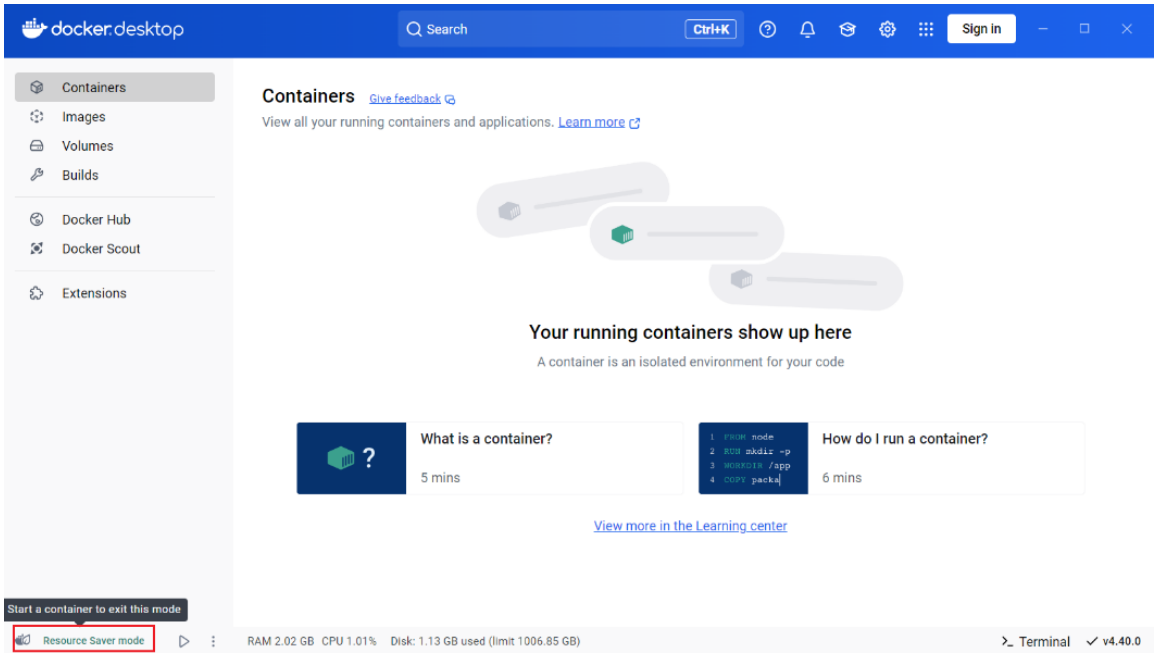
- Then, it displays a **Welcome Survey** page. Click on **Skip** link to skip this survey for now.



- After that, it displays Docker Desktop Dashboard with various tabs such as **Containers, Images, Volumes, Builds, Docker Hub**, etc. At the bottom, we can see that **Engine running** with **RAM, CPU** and **Disk storage** being used.



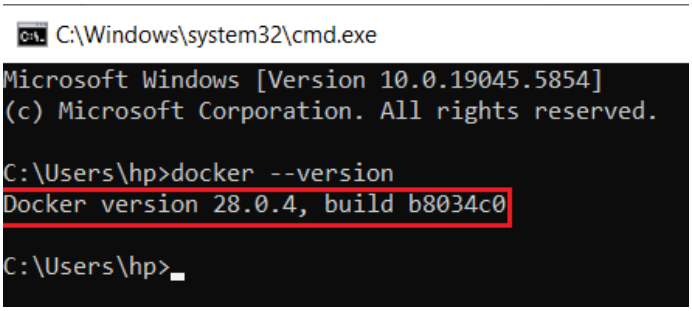
- Since no container is running at this moment, Docker engine goes into **Resource Save mode** in few seconds. It comes into running state again when a container is started.



2.3. Verify Docker:

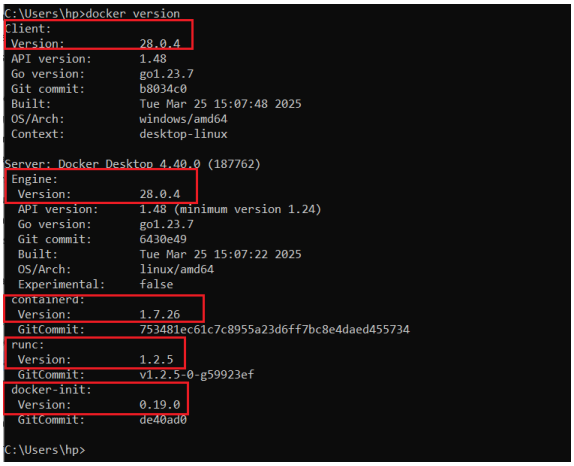
Run the following command to get the installed Docker version:

```
docker --version
```



Use the below command to verify the version of all Docker components

```
docker version
```



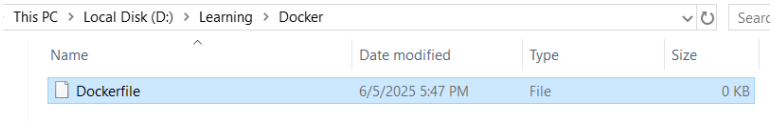
Now that Docker has installed successfully, it is important to get hands on Docker commands to work with Docker. Follow this link to understand the basic Docker commands.

3. Docker Image from Dockerfile:

A Docker image can be created from a `Dockerfile` starting with the base image available in the Docker registry and the resulting image is structured in the form of layers. Refer here to know more about `Dockerfile` instructions.

3.1. Create Dockerfile:

Create a file named `Dockerfile` (without any extension) in the desired directory. By default, Docker searches for the exact file named `Dockerfile` while building it. However, `Dockerfile` naming convention is not compulsory and can have a different name which can be provided to Docker while building but for now, let's go with `Dockerfile` naming.



Here, I created `Dockerfile` in my `D:\Learning\Docker` folder.

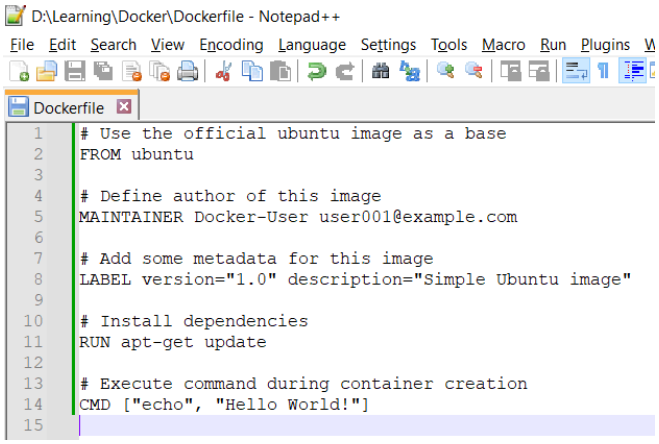
3.2. Add Instructions:

Open `Dockerfile` and add the below instructions for a simple **Ubuntu** image.

```
# Use the official ubuntu image as a base
FROM ubuntu

# Define author of this image
MAINTAINER Docker-User user001@example.com
```

```
# Add some metadata for this image
LABEL version="1.0" description="Simple Ubuntu image"

# Install dependencies
RUN apt-get update

# Execute command during container creation
CMD ["echo", "Hello World!"]
```

Save the file after adding instructions.



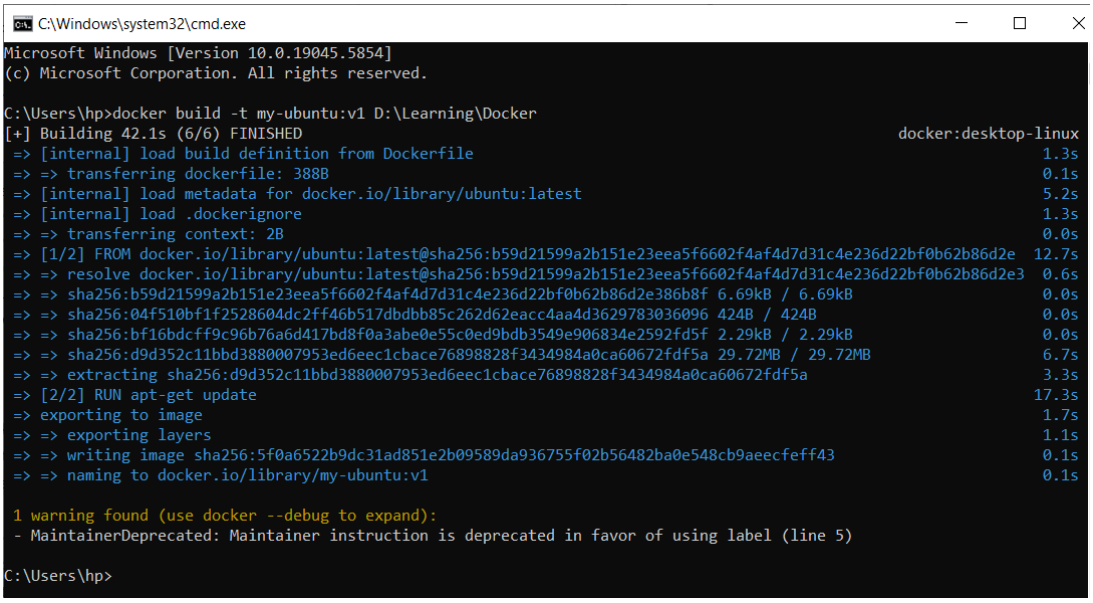
3.3. Build Docker Image:

Open a new **Command Prompt** and run the following command to build the image named `my-ubuntu` and tagged `v1`:

```
docker build -t my-ubuntu:v1 D:\Learning\Docker
```

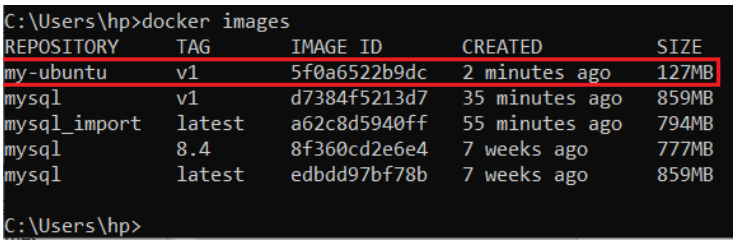
Note that my `Dockerfile` is present in `D:\Learning\Docker` directory. If your Dockerfile is present with a different name (say `Dockerfile.txt`) in the current directory, then use the below command where the `.` represents current directory from wherever this command is executed.

```
docker build -t my-ubuntu:v1 . -f Dockerfile.txt
```



Now, run the below command to see if the newly created image is available

```
docker images
```

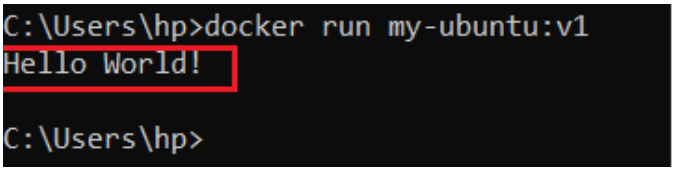


3.4. Run Docker Image:

Once the image is created, a container will be created by running the image.

Run the below command to create a container:

```
docker run my-ubuntu:v1
```



When the container is created, it executes the command that was specified in `Dockerfile` and exits.

Run the below command to see the list of all containers:

```
docker ps -a
```

```
C:\Users\hp>docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS                      PORTS                  NAMES
43abca6a58a8   my-ubuntu:v1   "echo 'Hello World!'"    28 seconds ago   Exited (0) 25 seconds ago                          quirky_johnson
7a931bb4e7dd   mysql          "docker-entrypoint.s…"   51 minutes ago   Up 27 minutes               3306/tcp, 33060/tcp    mysql

C:\Users\hp>
```

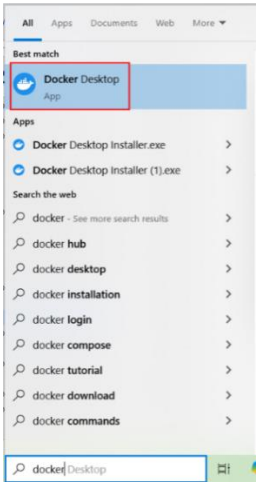
4. Docker Desktop:

Docker Desktop is a Graphical User Interface (GUI) provided by Docker to manage (build, run, share) images, containers, networks and volumes visually from the host machine. Docker Desktop takes care of default settings such as port mappings. It also allows to integrate various development tools and languages that saves development time, to automate builds and to collaborate securely with shared repositories. It includes monitoring tools to track container performance metrics, health checks and offers logging capabilities to capture container logs for debugging purposes.

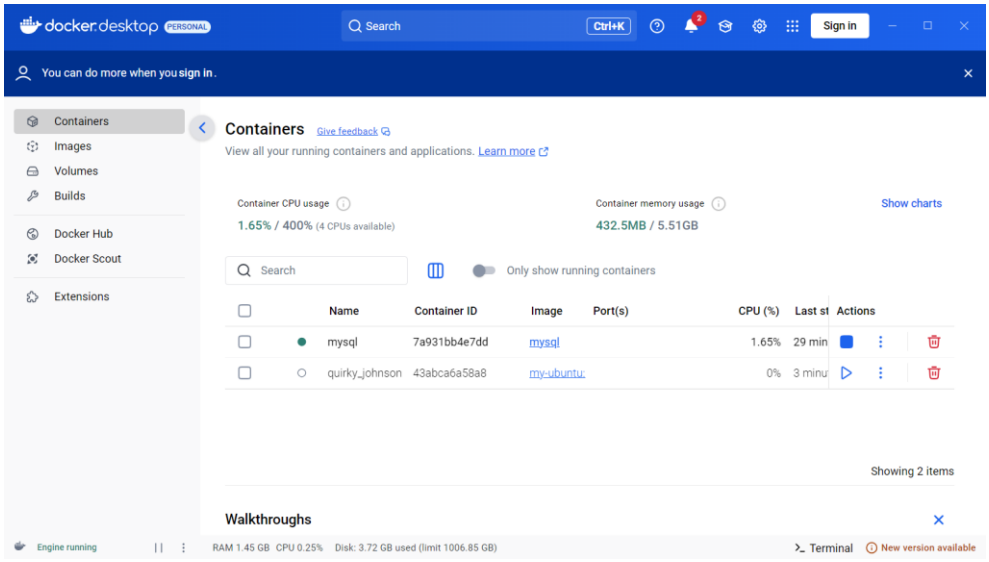
Docker Desktop includes components such as **Docker Engine**, **Docker CLI**, **Docker Build**, **Docker Compose**, **Docker Extensions**, **Docker Content Trust**, **Kubernetes**, **Credentials Helper** and **Ask Gordon** *(a personal AI assistant)*.

4.1. Launch Docker Desktop:

Open **Docker Desktop** application by searching for **Docker** in the search box of the task bar in your Windows system and selecting the **Docker Desktop** application.



In few seconds, it launches **Docker Desktop** application and displays the Docker Dashboard with various tabs such as **Containers, Images, Volumes, Builds, Docker Hub**, **Docker Scout** and **Extensions**.



The Docker Desktop header displays various icons including:

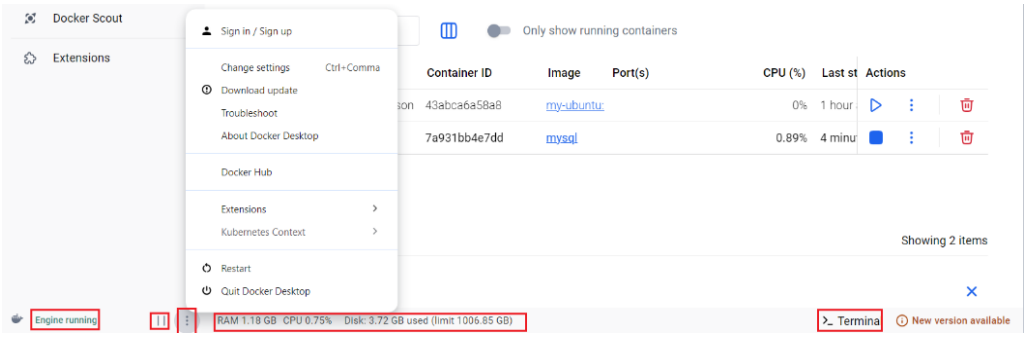
- **Quick Search** icon to search for local containers, local images, public Docker Hub images, images from remote repositories, extensions to install or open, volumes and official Docker documentation.
- **Notifications** icon to get notified of new releases and installation progress updates,
- **Troubleshoot** icon to debug and perform restart operations.
- **Learning Center** icon get started with quick in-app walkthroughs and provides other resources for learning about Docker.
- **Settings** icon to configure Docker Desktop settings.

The Docker Desktop footer displays:

- **Docker Engine status** which shows Engine running. When no container is running, it can go into **Resource Saver mode** to save Docker Desktop's CPU and memory utilization. Select **Pause** icon after which all processes are frozen and current state of containers is saved in memory.
- **Action** icon to perform various actions such as Sign in/Sign up to Docker, Change settings, Check for updates, Troubleshoot, Switch to Windows containers (if Docker is running on Windows), About Docker Desktop, Docker Hub, Extensions, Kubernetes, Restart, Quit Docker Desktop.
- **RAM, CPU and Disk** currently in use by the Docker engine.
- **Terminal** icon to use the integrated terminal within the Docker Desktop. This integrated terminal can also be opened from the running container. The integrated terminpersists the

session when navigated to another part of Docker desktop dashboard and supports copy, paste, search and clearing session features. To use the external terminal, navigate to the **General** tab in **Settings** and select the **System default** option under **Choose your terminal**.

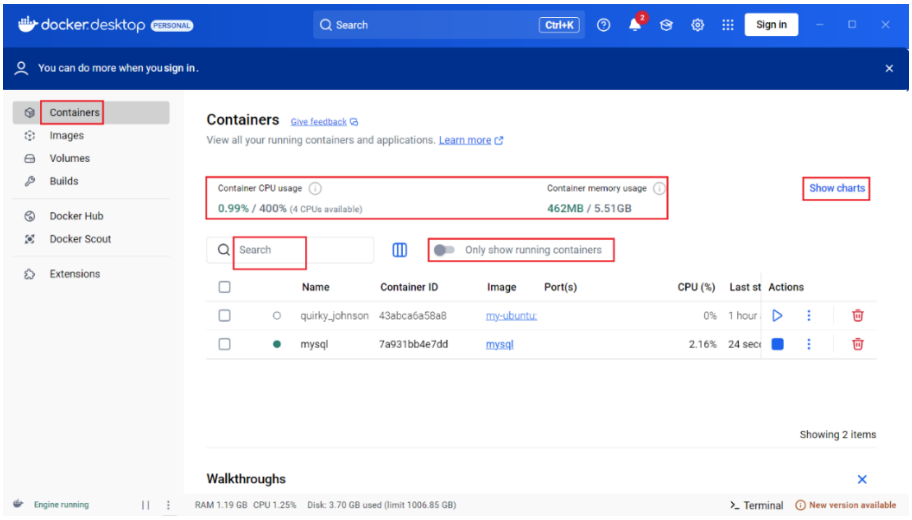
- **Current version** of Docker Desktop.



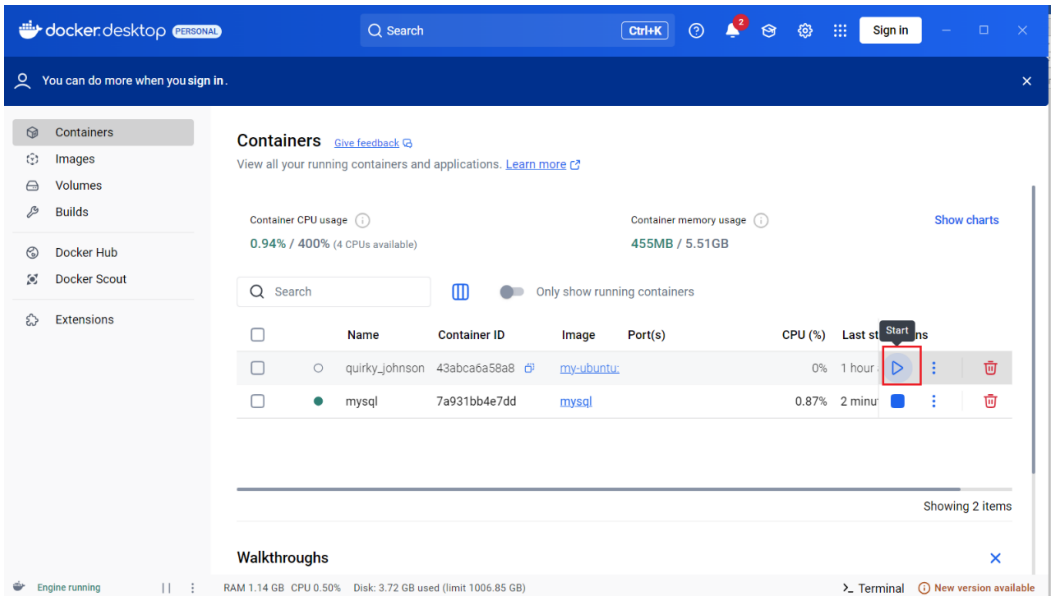
4.2. Containers View:

The **Containers** view in Docker Desktop provides a graphical interface for managing the life cycle of containers. It lists all running and stopped containers and provides a runtime view of all containers and applications. It allows to inspect, interact with and manage Docker objects including containers and Docker Compose-based applications. It also allows to start, stop, pause, resume, restart and delete containers, view image packages and CVEs, open the application in VS code, open the port exposed by the container in a browser and use the **Docker Debug** feature *(which is available with the paid Docker subscription only)*.

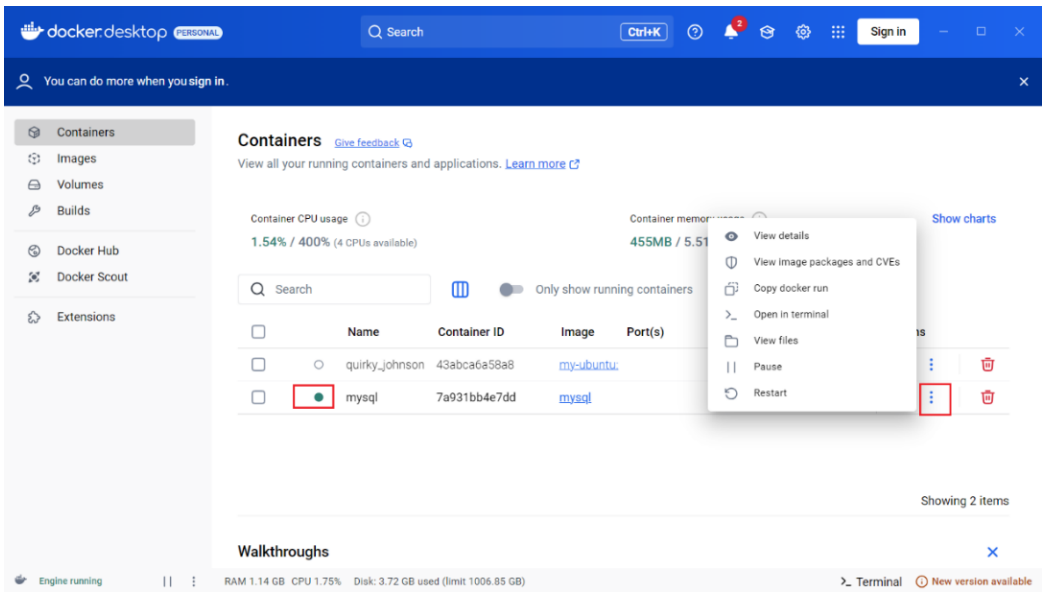
By default, the Containers view displays all stopped and running containers but the toggle option **Only show running containers** allows to display running containers alone. The **Containers status** bar displays the CPU and memory usage of all running containers which can also be viewed in graphical format by choosing **Show charts** option and the **Search** field allows to search for a specific container.



To start a container, select the container name, in this example it is container with `my-ubtunu` image, and click on **Start** icon under **Actions** menu:



Once the container is started successfully, it changes to **Running** state. Click on **Show Containers Actions** icon under **Actions** menu to view container details, view image packages and CVEs, copy docker run, open integrated terminal, view container files, pause the container and restart the container.

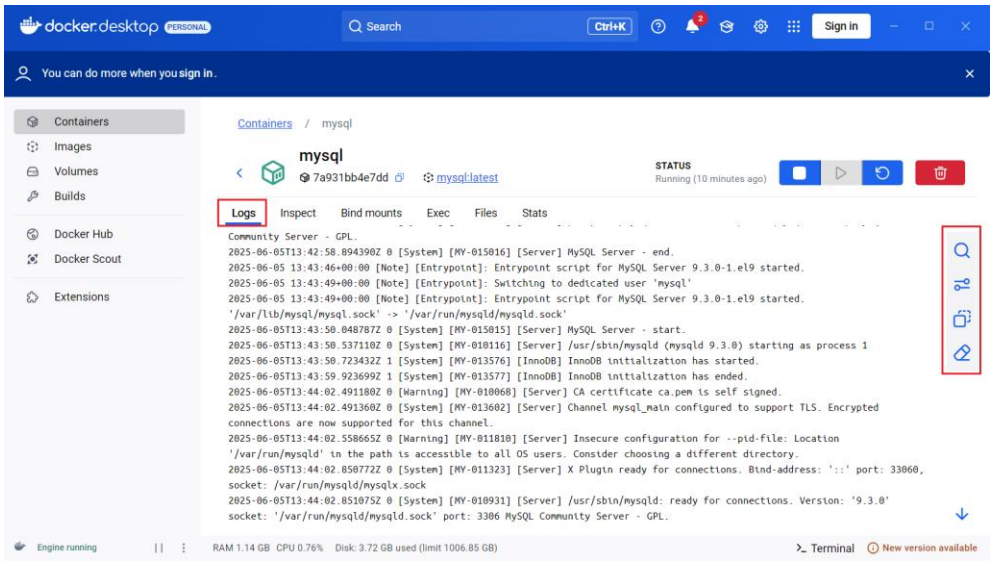


Click on `mysql` container that is running or select **View details** option under **Show Containers Actions** icon to inspect the container which displays various tabs:

- **Logs** tab allows to view the output from container in real time. This tab is the same as running the following command:

```
docker logs <container-id>
```

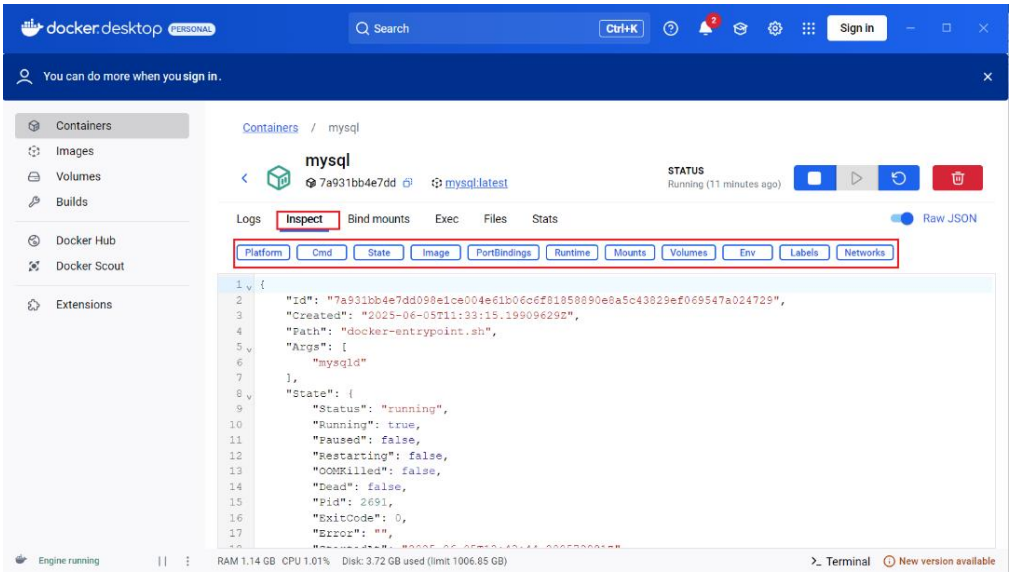
The **Logs** tab provides various log options such as **Search terminal** to open specific log entries, **Settings** to define how log entries can be displayed, **Copy** to copy all logs to clipboard, **Clear terminal** to clear the logs terminal.



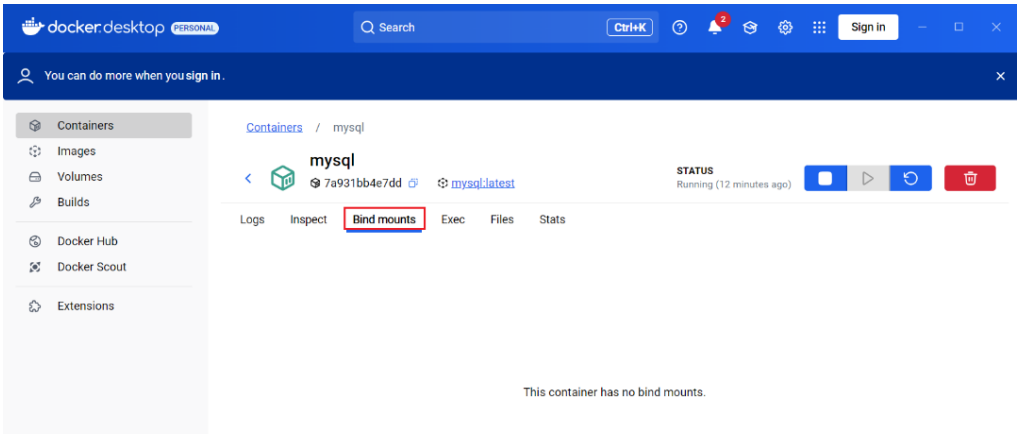
- **Inspect** tab allows to view the low-level information about the container including local path, version number of the image, port mapping, and other details. This tab is the same as running the following command:

```
docker inspect <container-id>
```

For ease of use, the **Inspect** tab provides various sub-tabs such as **Platform, Cmd, State, Image, PortBindings**, etc. to get the specific details of the container.



- **Bind mounts** tab allows to view the volume mounts that are bounded to this container.



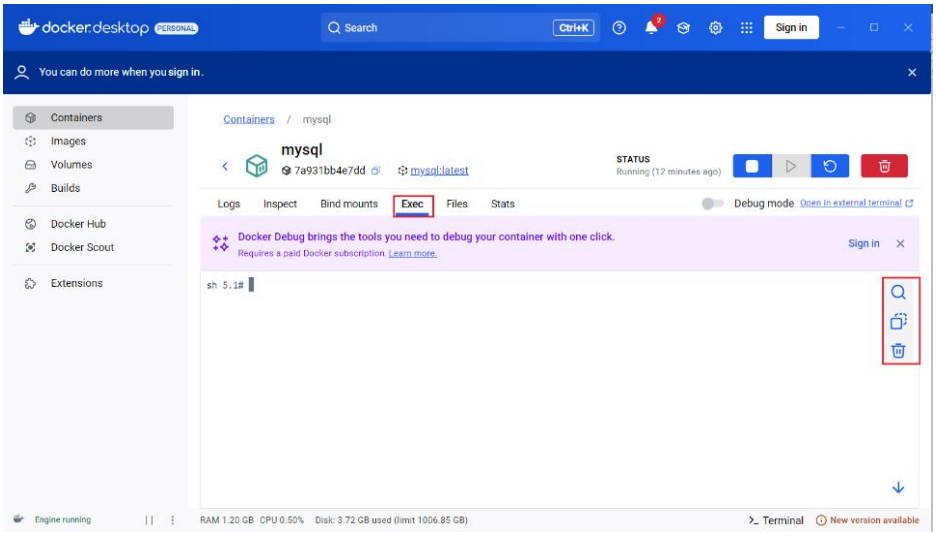
- **Exec** tab allows to quickly run commands within the running container. This tab is the same as running one of the following commands:

  When accessing Linux containers:

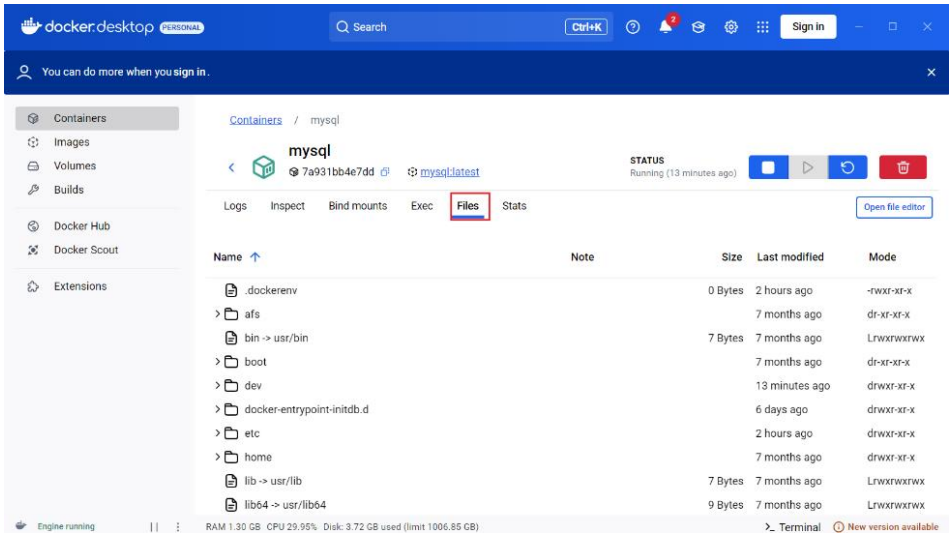
  ```
  docker exec -it <container-id> /bin/sh
  ```

  When accessing Windows containers:

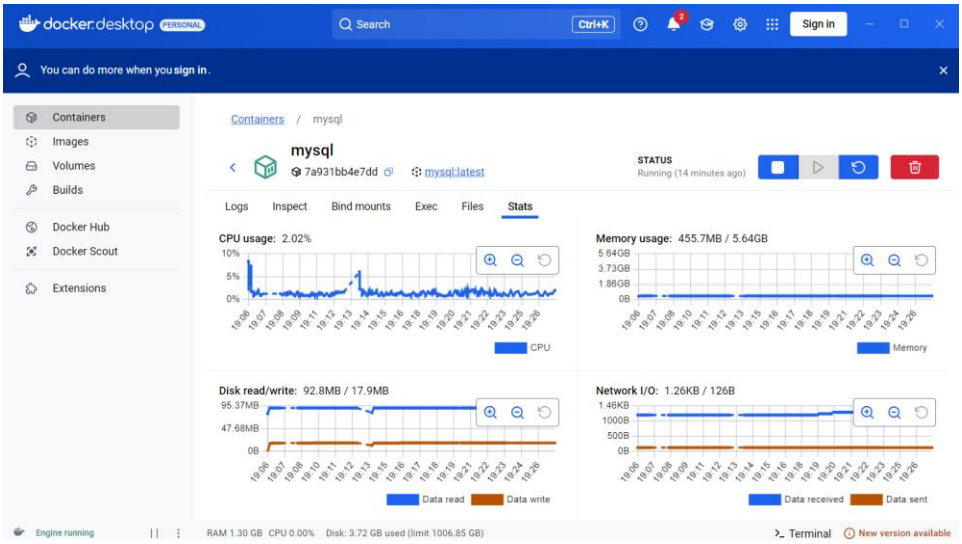
  ```
  docker exec -it <container-id> cmd.exe
  ```



- **Files** tab allows to explore the filesystem of the container. It allows to see files that have been added, modified or deleted recently, to edit and delete a file and to drag and drop files between host and container.



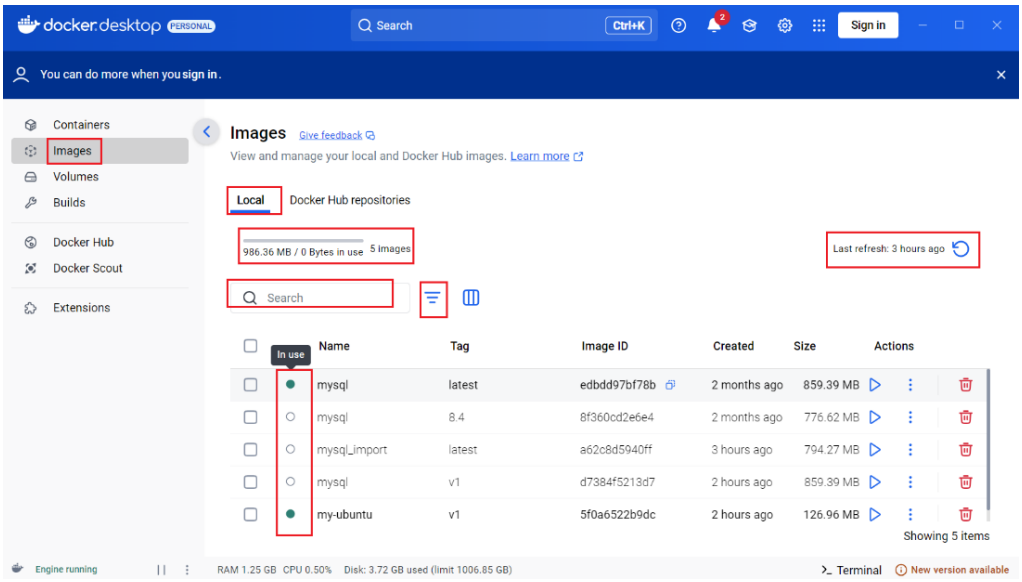
- **Stats** tab allows to display container's resource utilization and monitor container's CPU, memory, disk space and network usage over time in a graphical format.



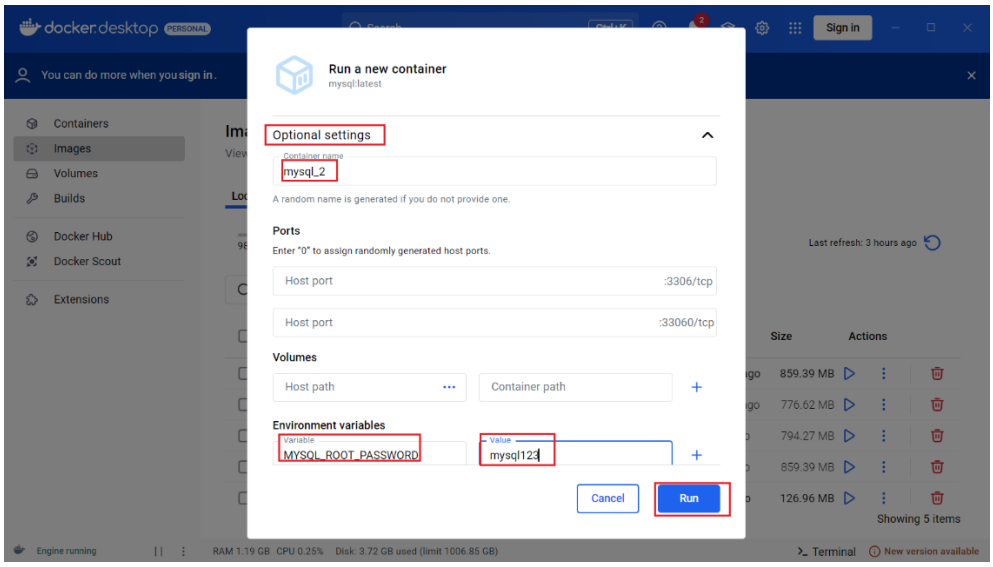
4.3. Images View:

The **Images** view in Docker Desktop provides a graphical interface for managing images. It lists all Docker images available locally and shared on Docker Hub. It allows to run an image as a container, pull the latest version of an image from Docker Hub, push the image to Docker Hub and inspect images.

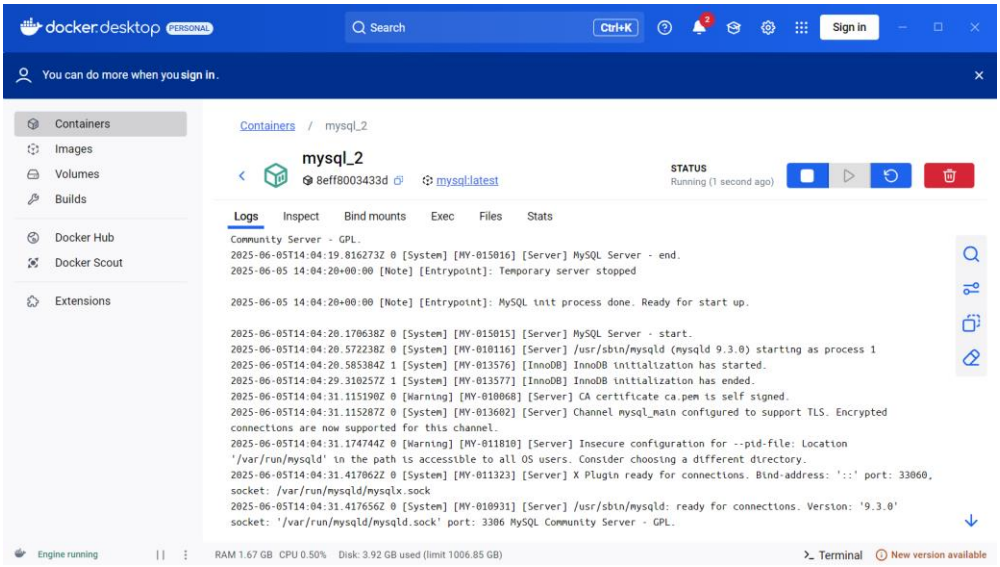
All docker images that are either built locally and pulled from the Docker repository are visible under **Local** tab. The **In Use** option next to the image name helps to identify if the image is in used by the container or unused. The **Images status** bar displays the number of images and total disk space used by the images and when this information was last refreshed. The **Search** field allows to search for a specific image and sort images by *In use, Unused* and *Dangling*.



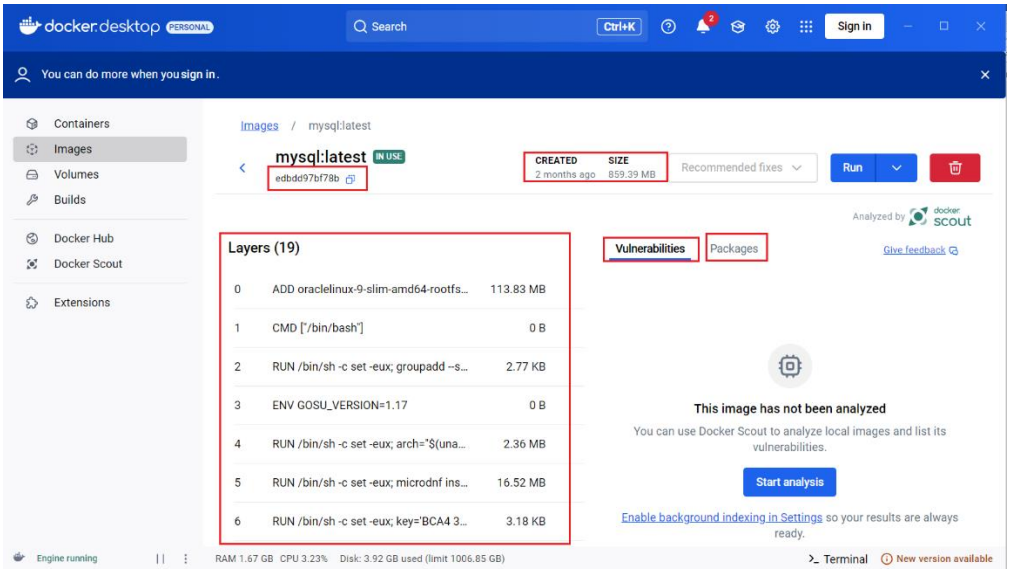
To run an image as a container, select the image, in this example it is `mysql`, and click on **Run** icon under **Actions** menu. When prompted, select the **Optional settings** drop-down and specify the container name as `mysql_2` and add the **Environment variable** as `MYSQL_ROOT_PASSWORD` and provide some password (let's say `mysql123`) and select **Run**. Additionally, we can also specify container ports and volumes if needed.



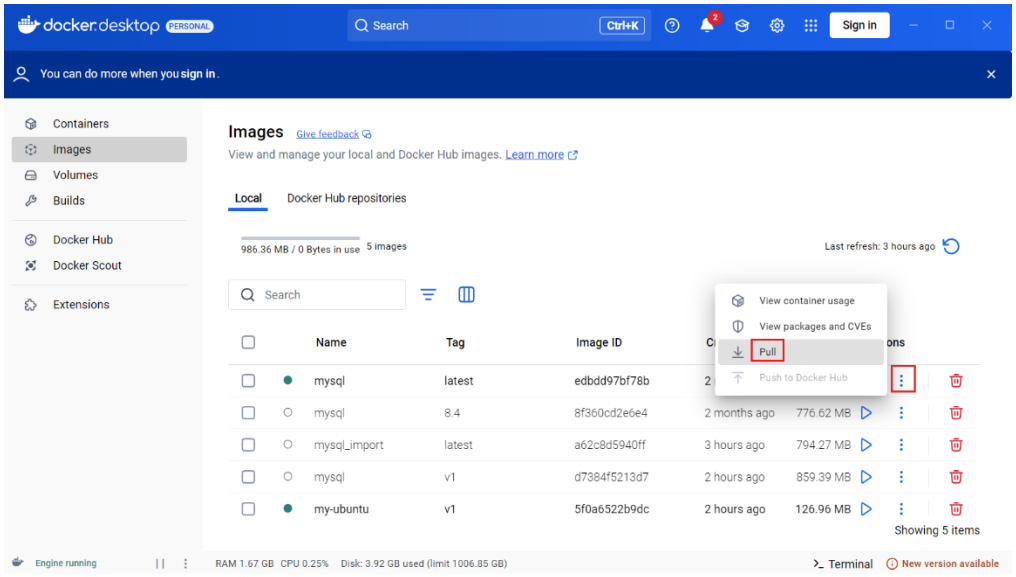
It then creates a new container `mysql_2` and displays log entries



To inspect an image, click on the `mysql` image to display detailed information about the image such as image history, image ID, image created date, image size, layers making up the image, base images used, vulnerabilities found and packages inside the image.

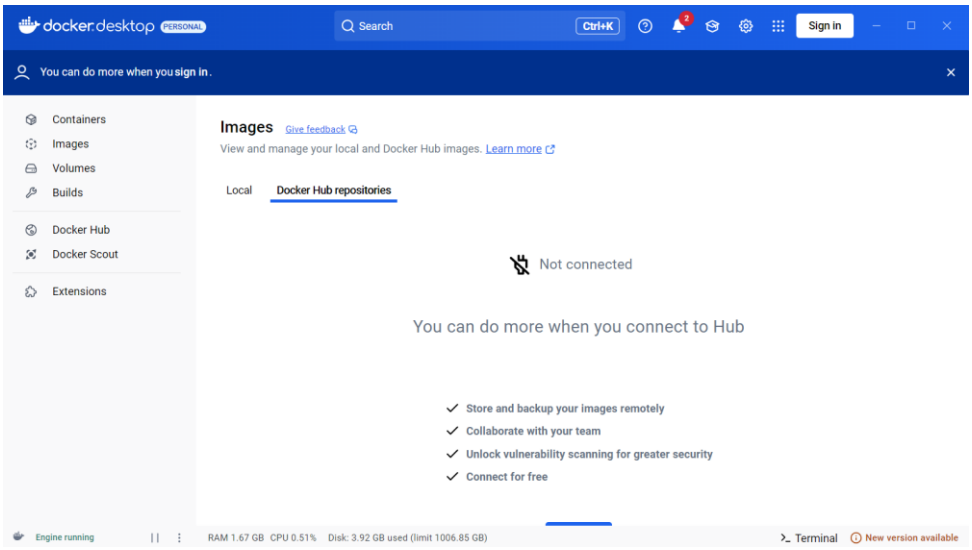


To pull the latest version of the image from Docker Hub, select the image and click on **Show image actions** icon and select **Pull**. Note that the repository must exist on Docker Hub in order to pull the latest version of an image.

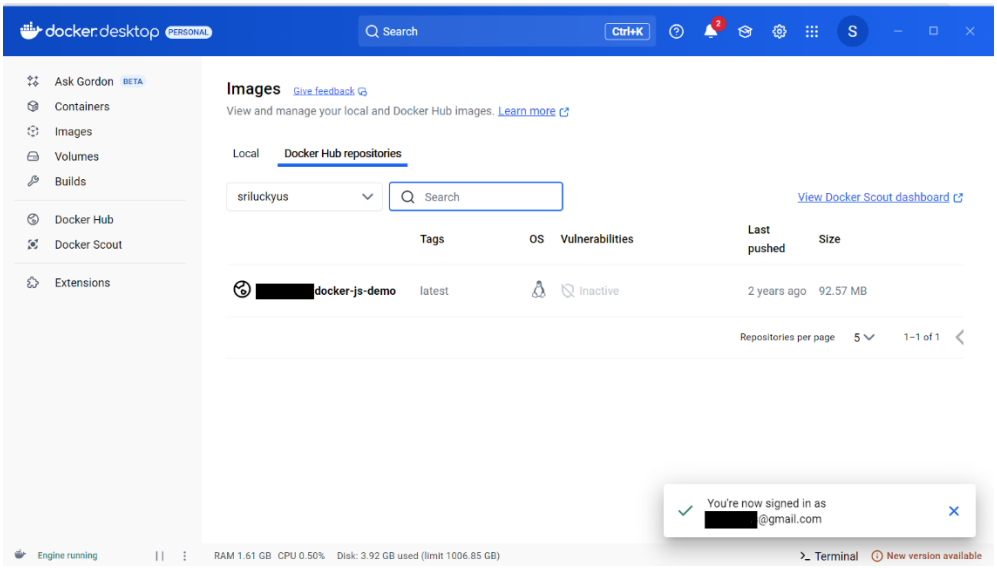


To push the image to Docker Hub, select the image and click on **Show image actions** icon and select **Push to Docker Hub**. Note that the Docker Hub has already been logged in and can push the image if it contains the correct username/organization in its tag.

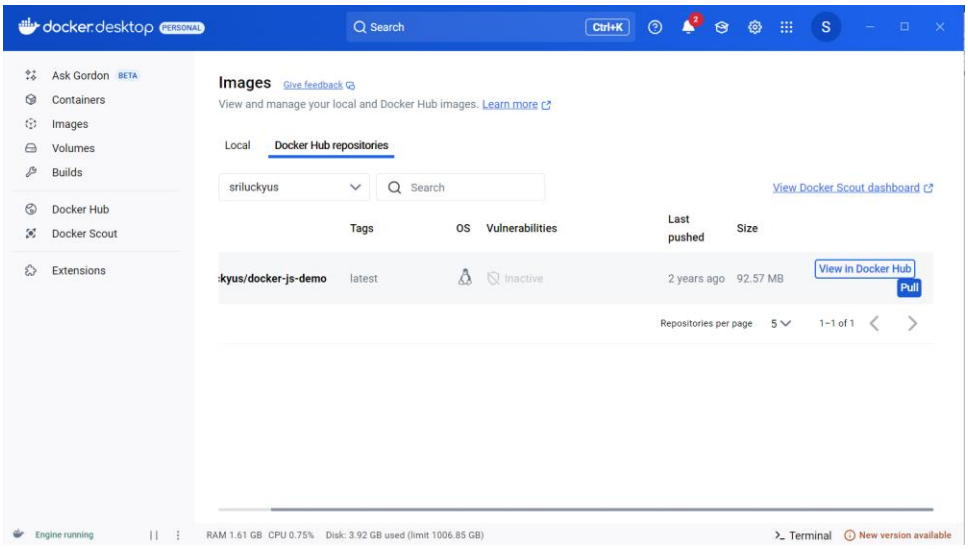
The **Images** view also allows to manage and interact with images in Docker Hub repositories. Click on **Docker Hub repositories** tab which prompts to sign in to Docker Hub account, if not already signed in.



When signed in, it shows a list of images in Docker Hub organizations and repositories that you have access to. Select an organization from the drop-down to view a list of repositories for that organization.



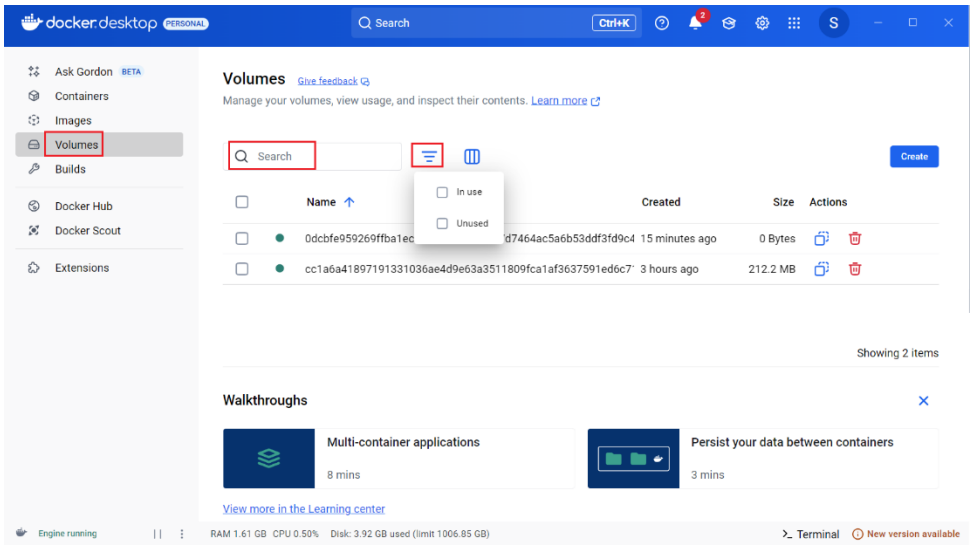
If **Docker Scout** is enabled on the repositories, it displays image analysis results and health scores next to the image tags. Hover over the image tag which displays **Pull** option pull the latest version of the image from Docker Hub and **View in Hub** to open the Docker Hub page and display detailed information about the image.



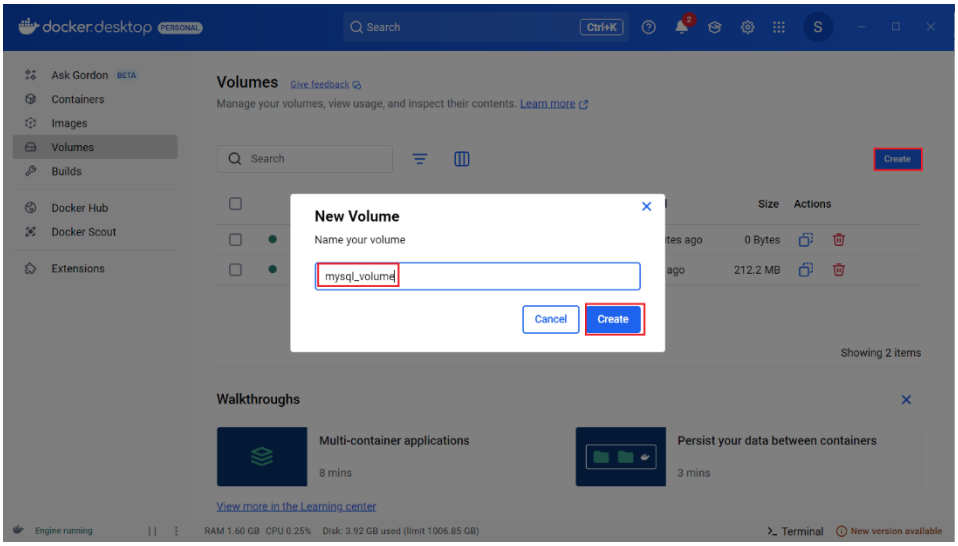
4.4. Volumes View:

The **Volumes** view in Docker Desktop provides a graphical interface for managing Docker volumes which are used to persist data generated and used by Docker containers. It displays list of volumes and allows to easily create, inspect, delete, clone, empty, export, and import Docker volumes, browse files and folders in volumes and see which volumes are being used by containers.

The **Search** field in the Volumes view allows to search for a specific volume and filter by *In use* and *Unused*.

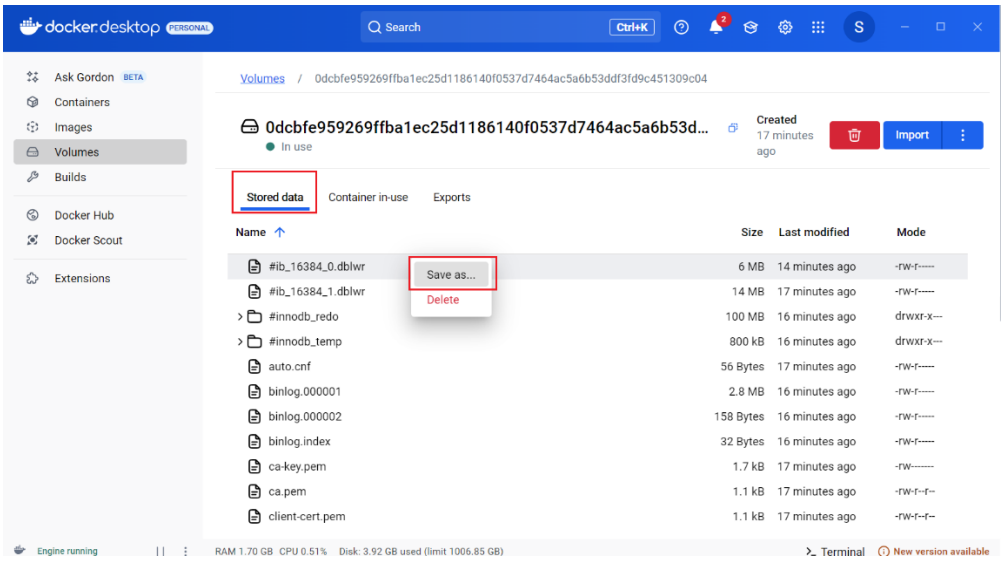


To create a volume, select **Create** button and specify the volume name (let's say `mysql_volume`) and click on **Create** in **New Volume** modal.

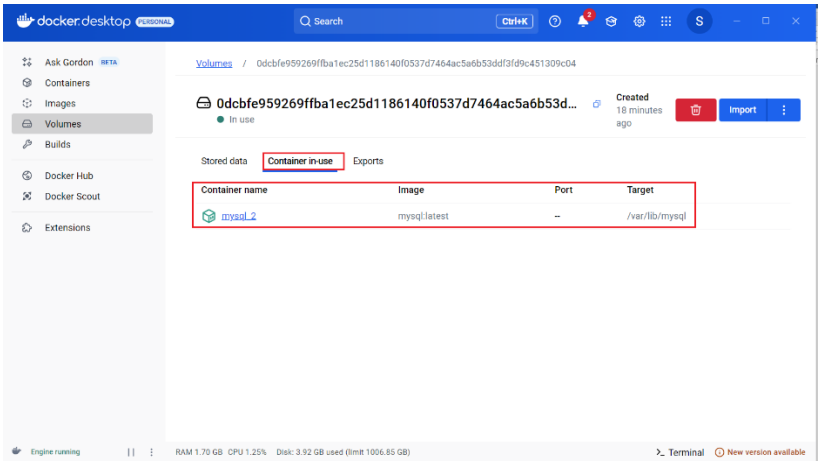


To inspect a volume, click on the volume to display detailed information about the volume such as:

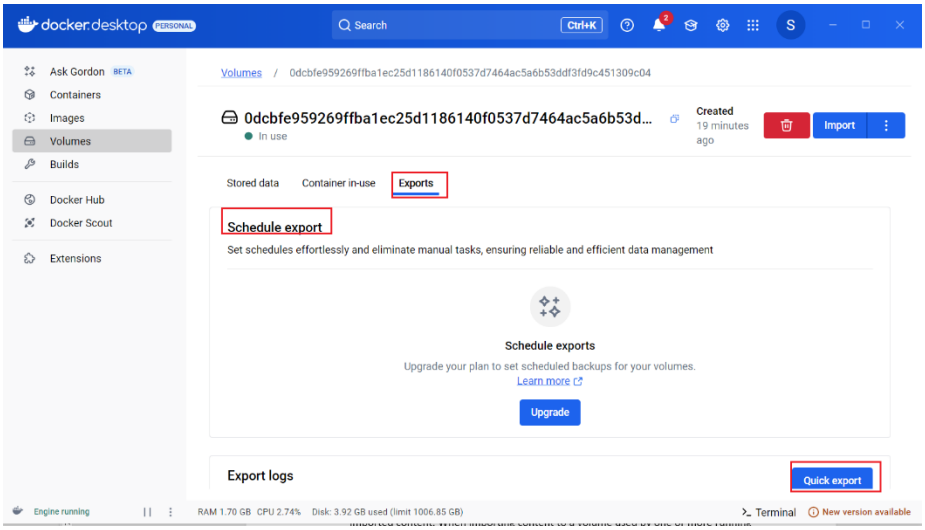
- Stored Data tab which displays the files and folders in the volume and the file size. To save a file or a folder, right-click on the file or folder to display the options menu, select **Save as...**, and then specify a location to download the file.



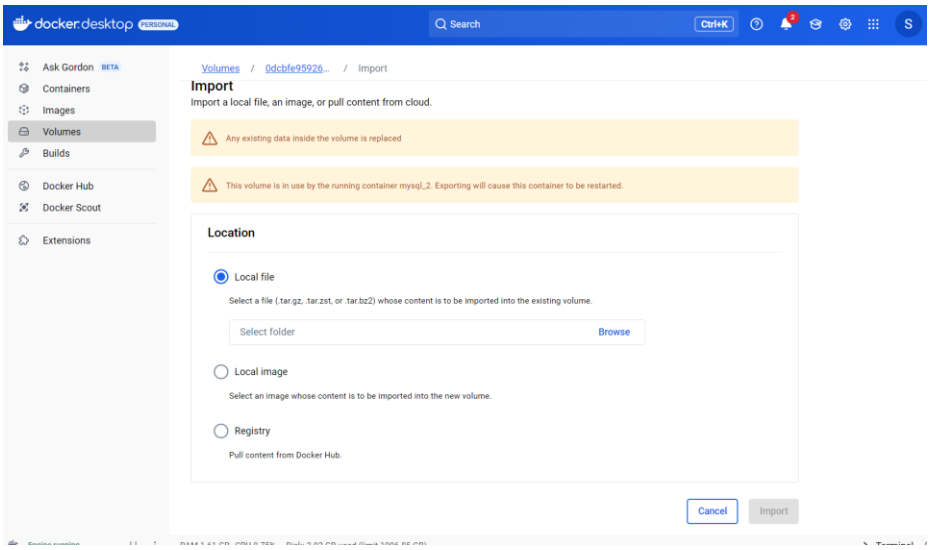
- **Container in-use** tab which displays the name of the container using the volume, the image name, the port number used by the container, and the target. A target is a path inside a container that gives access to the files in the volume.



- **Export** tab which exports the content of a volume to a local file, a local image, and to an image in Docker Hub, or to a supported cloud provider. When exporting content from a volume used by one or more running containers, the containers are temporarily stopped while Docker exports the content, and then restarted when the export process is completed.
  Select **Quick export** button to export a volume now to Docker Hub or choose **Schedule export** option to schedule a recurring export but it requires a paid Docker subscription.



- Click on **Import** button and choose the import location to import a local file, a local image, or content from Docker Hub. Any existing data in the volume is replaced by the imported content. When importing content to a volume used by one or more running containers, the containers are temporarily stopped while Docker imports the content, and then restarted when the import process is completed.

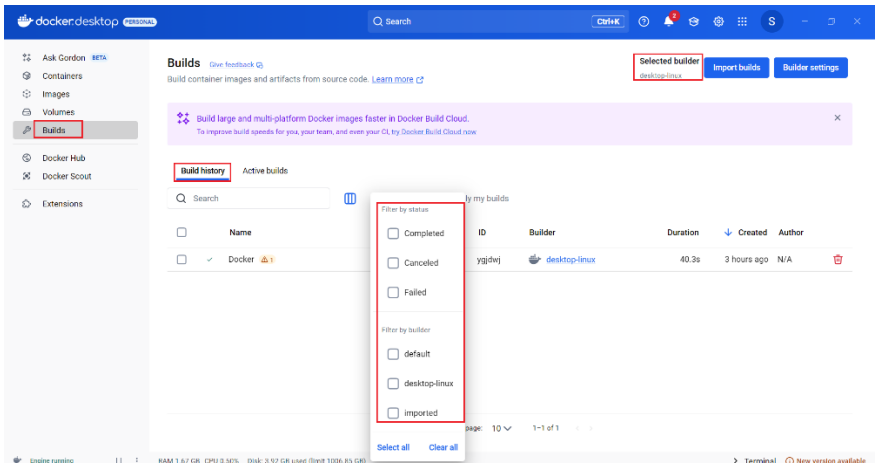


4.5. Builds View:

The **Builds** view in Docker Desktop provides a graphical interface for managing and monitoring image builds from Dockerfiles, offering tools for troubleshooting, and integrating build options to streamline Docker image creation. It displays a list of all ongoing and completed builds and allows to inspect build history, monitor active builds, and manage builders.

By default, the Builds view displays **Build history** to view the history of all completed builds but use the toggle option **Show only my builds** allows to display builds created by the current user with access to logs, dependencies, traces and others. This view also displays any active or completed cloud builds by other team members if connected to a cloud builder through Docker Build Cloud.

The **Search** field allows to search for a specific build and filter by status such as *Completed*, *Cancelled* and *Failed*, and filter by builder such as *default*, *desktop-linux*, and *imported*. The top right corner displays the current selected builder



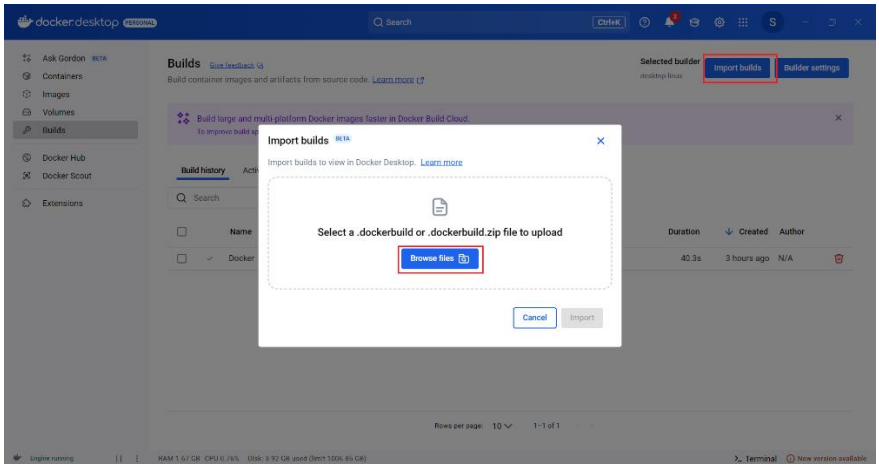
Note that when building Windows container images using the `docker build` command, the legacy builder is used which does not populate the Builds view. Instead, use either of the build commands to use **BuildKit** which populates the Builds view:

```
DOCKER_BUILDKIT=1 docker build .
```

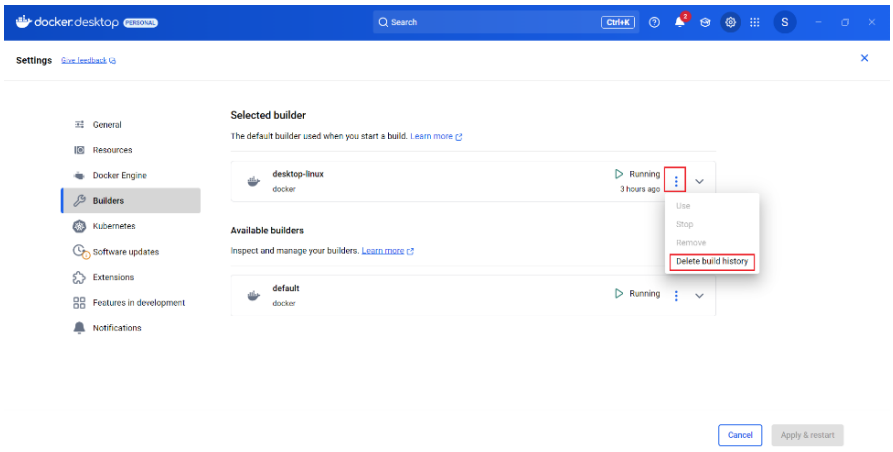
or

```
docker buildx build .
```

Click on **Import builds** button to import build files with `.dockerbuild` or `.dockerbuild.zip` extension for builds by other people. When a build record is imported, it gives full access to the logs, traces, and other data for that build, directly in Docker Desktop.

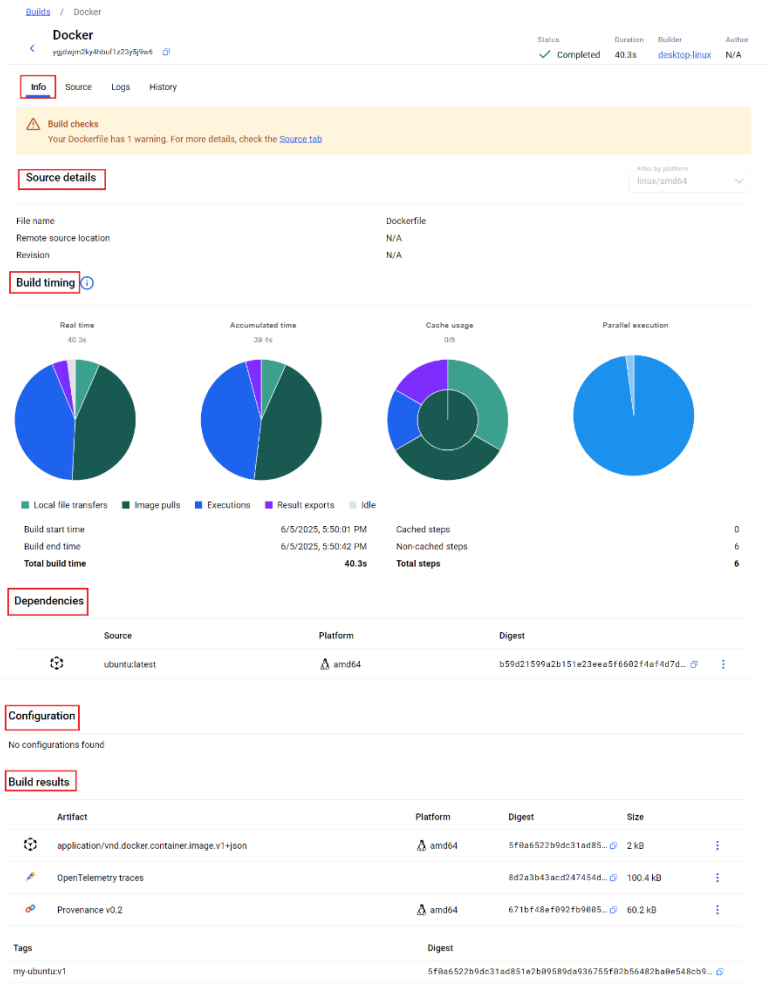


Click on **Builder settings** button to manage builders including inspecting the state and configuration of active builders, starting and stopping a builder, deleting the build history, adding or removing builders, connecting and disconnecting cloud builders directly from the Docker Desktop settings.

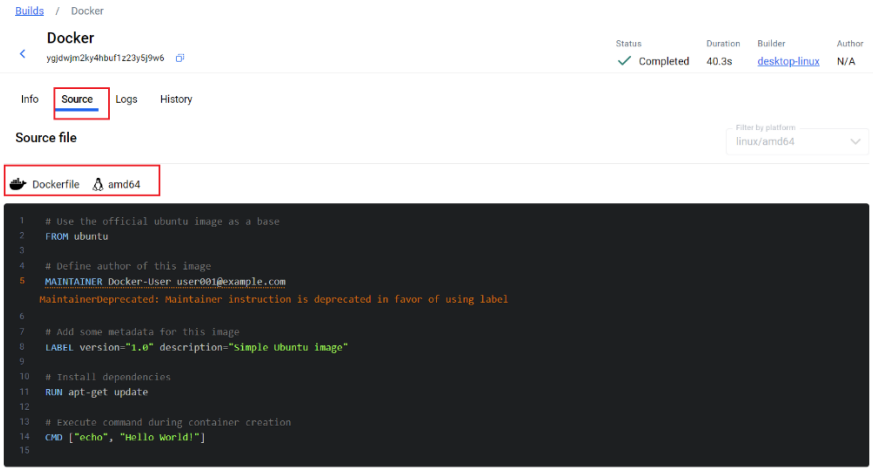


To inspect a build, click on the build name to display detailed information about the build such as:

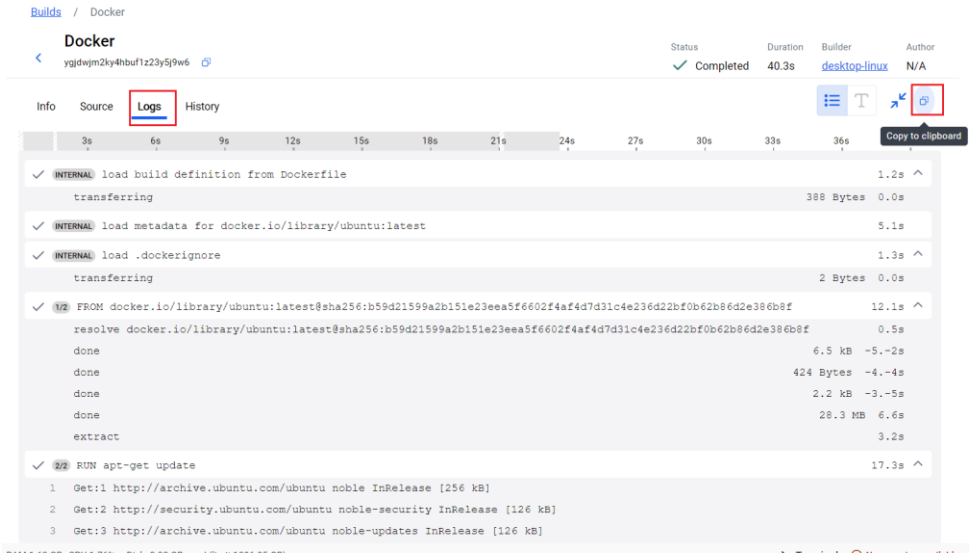
- **Info** tab which displays details about the build with various sections including
  - **Source Details** to display information about the frontend and, if available, the source code repository used for the build.
  - **Build timing** to displays charts showing a breakdown of the build execution from various angles.
  - **Dependencies** to display images and remote resources such as container images used for the build, Git repositories and Remote HTTPS resources included using the ADD Dockerfile instruction.
  - **Configuration** to display parameters such as Build arguments, Secrets, SSH sockets, Labels, Additional contexts, etc.
  - **Build results** to display a summary of the generated build artifacts, including image manifest details, attestations, and build traces.



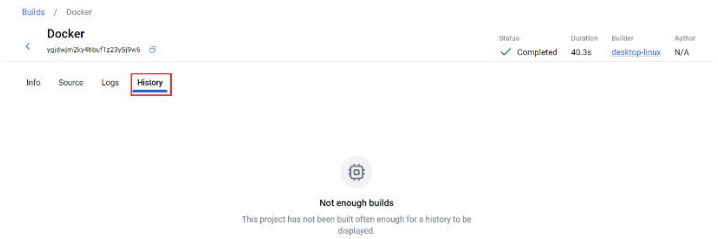
- **Sources** tab which displays the frontend used to create a build. If the build fails, an **Error** tab displays instead of the Source tab and the error message is inlined in the Dockerfile source, indicating where and why the failure happened.



- **Logs** tab which displays the build logs that are updated in real-time for active builds. The **Copy** button allows to copy the plain-text version of the log to the clipboard.

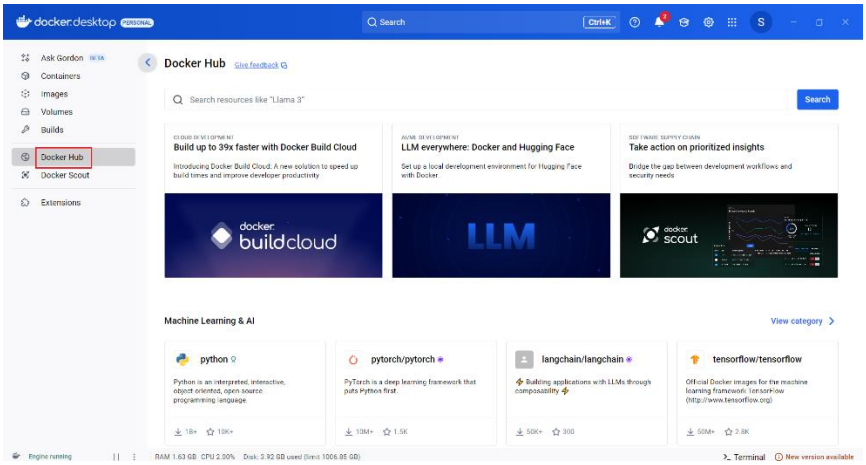


- **History** tab which displays statistics data about completed builds.



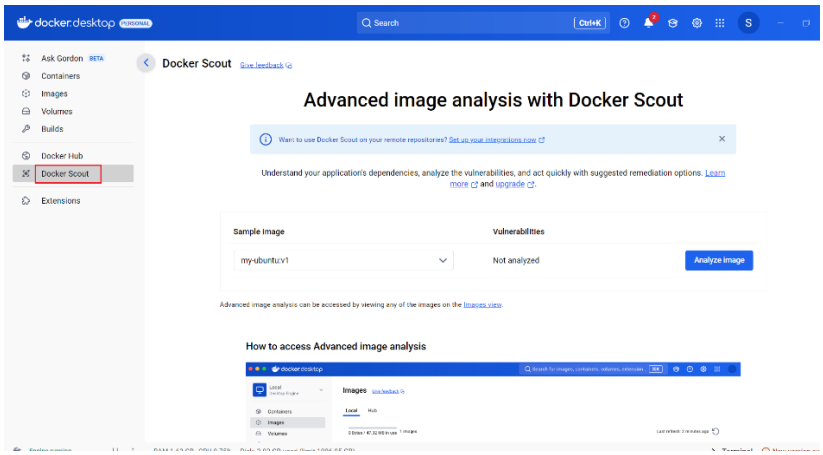
4.6. Docker Hub View:

The **Docker Hub** view allows to interact with Docker Hub when signed in from the Docker Desktop directly.



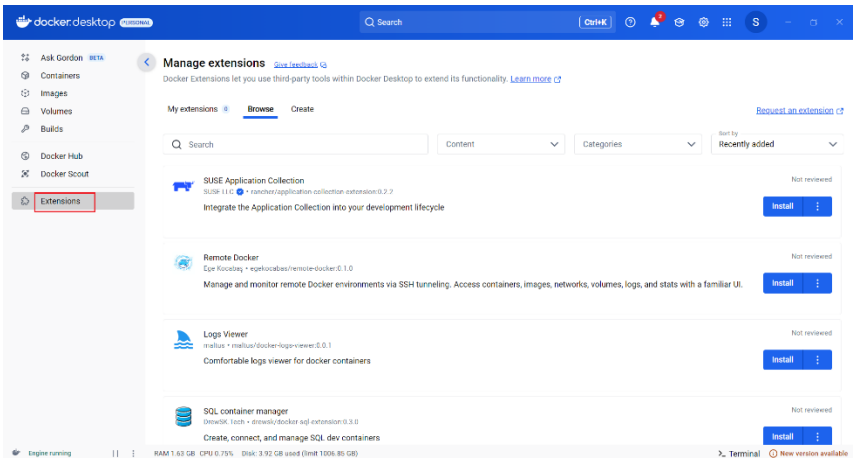
4.7. Docker Scout View:

The **Docker Scout** view in Docker Desktop allows to access the Docker Scout service to analyze images and raise security vulnerabilities. Docker Scout compiles an inventory of components, also known as a Software Bill of Materials (SBOM) which is matched against a continuously updated vulnerability database to pinpoint security weaknesses.



4.8. Docker Extensions View:

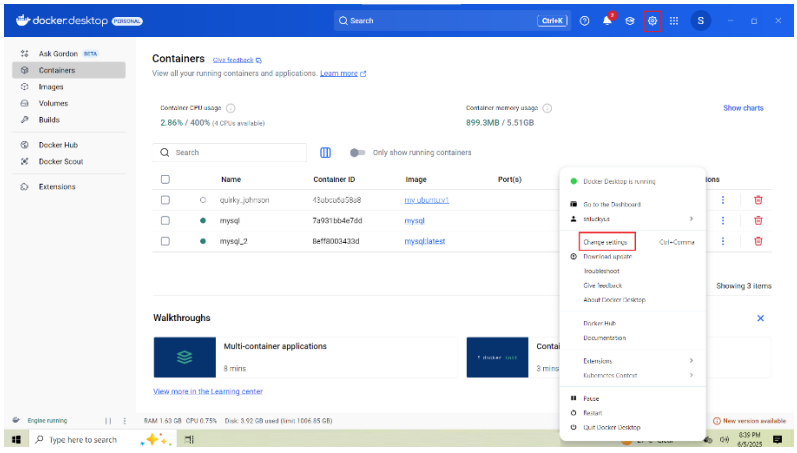
The **Extensions** view in Docker Desktop allows to use integrate third-party tools seamlessly in Docker Desktop for application development and deployment of workflows. It allows to explore the list of available extensions in Docker Hub or in the Extensions Marketplace within Docker Desktop.



4.9. Change Settings:

There are two ways to navigate to Docket Desktop settings:

- Select the **Settings** icon on the top right corner of the Docker Desktop Dashboard.
- Right click on the **Docker** menu on the task bar and select **Change settings** option.



These settings can also be accessed from the `settings-store.json` file located at `C:\Users\[USERNAME]\AppData\Roaming\Docker\settings-store.json` in Windows or at `~/.docker/desktop/settings-store.json` in Linux or at `~/Library/Group\Containers/group.com.docker/settings-store.json` in MacOS.

The Docker Desktop Settings displays:

- **General** tab by default allows to configure when to start Docker Desktop and specify other settings.
- **Resources** tab allows to configure CPU, memory, disk, proxies, network, and other resources.
    - **Advanced** tab under Resources allows to limit resources available to the Docker Linux VM. When WSL 2 backend is used, it allows to configure limits on the memory, CPU, and swap size allocated to WSL 2 in a `.wslconfig` file. It also displays the current Disk image location and enables or disables Resource Saver.
    - **Proxies** tab allows to define the manual proxy settings. Docker Desktop supports the use of HTTP/HTTPS and SOCKS5 proxies and automatically detects and uses the system proxy.
    - **Network** tab allows to define the custom subnet. Docker Desktop uses a private IPv4 network for internal services such as a DNS server and an HTTP proxy.
    - **WSL Integration** tab allows to configure WSL 2 distributions. By default, the integration is enabled on default WSL distribution. To change the default WSL distribution to Ubuntu, run the following command:
      ```
      wsl --set-default ubuntu
      ```

- **Docker Engine** tab allows to configure the Docker daemon used to run containers with Docker Desktop. The Docker daemon can be configured using a JSON configuration file located at `$HOME/.docker/daemon.json` which might look like:

```
{
"builder": {
"gc": {
"defaultKeepStorage": "20GB",
"enabled": true
}
},
"experimental": false
}
```

- **Builders** tab allows to inspect and manage builders. It allows to inspect only active builders.
    - The **Selected builder** section displays the selected builder.
    - To create a builder, use the Docker CLI.
    - Builders that use the `docker-container` driver run the `BuildKit` daemon in a container
    - Builders can be started and stopped only using the `docker-container` driver.

- **Kubernetes** tab allows to enable Kubernetes. Docker Desktop includes a standalone Kubernetes server to test deploying Docker workloads on Kubernetes.
- **Software Updates** tab allows to notify any updates available to Docker Desktop.
- **Extensions** tab allows to enable Docker Extensions and allow only extensions distributed through the Docker Marketplace.
- **Feature control** tab allows to control settings for **Beta features** and **Experimental features**.
- **Notifications** tab allows to turn on or turn off notifications for the events including status updates on tasks and processes, recommendations from Docker, Docker announcements, Docker surveys. By default, all general notifications are turned on.

Select **Apply & Restart** to save your settings and restart Docker Desktop.

5. Create Python Docker Application:

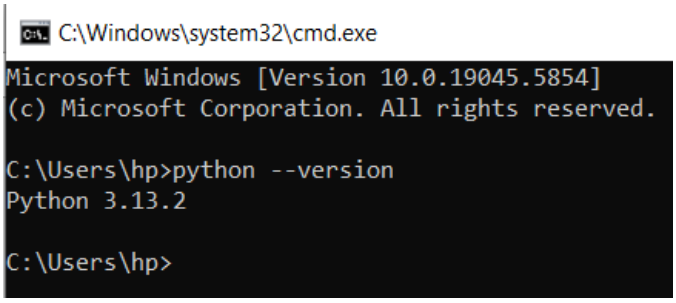
We will create a simple **Python Flask** containerized application to run from Docker. **Flask** (`flask` library) is a Python framework used to create a web application.

5.1. Verify Python Version:

To create a Python application, we should have **Python** software running in the system. If Python is not installed yet in your system, install it from **Anaconda Distribution** which is an open source software built for **Python and R** programming languages. Refer to Official Anaconda Installation Guide on how to install it.

Open a new **Command Prompt** and verify the installed Python version using the below command:

```
python --version
```



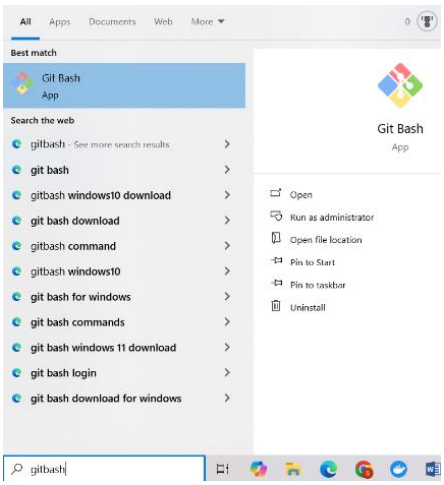
In my system, **Python 3.13.0** version is installed.

5.2. Launch VS Code:

To create this Python based application, we will use **Microsoft VS Code** software to write the Python script and execute Docker commands for easy management *(it is not necessary to have VS Code but it is good to use)*. Follow these steps to install VS Code software if you do not have it already.

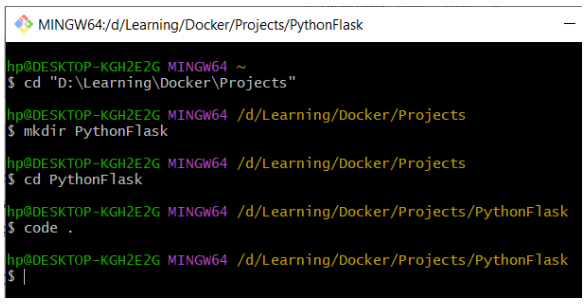
Though VS Code can be opened directly, we will use **Git** software to launch VS Code. Install **Git** software on Windows from the Git website if you do not have it.

In the Windows search bar, start typing “gitbash” and select the first match which opens up **Git Bash** application.

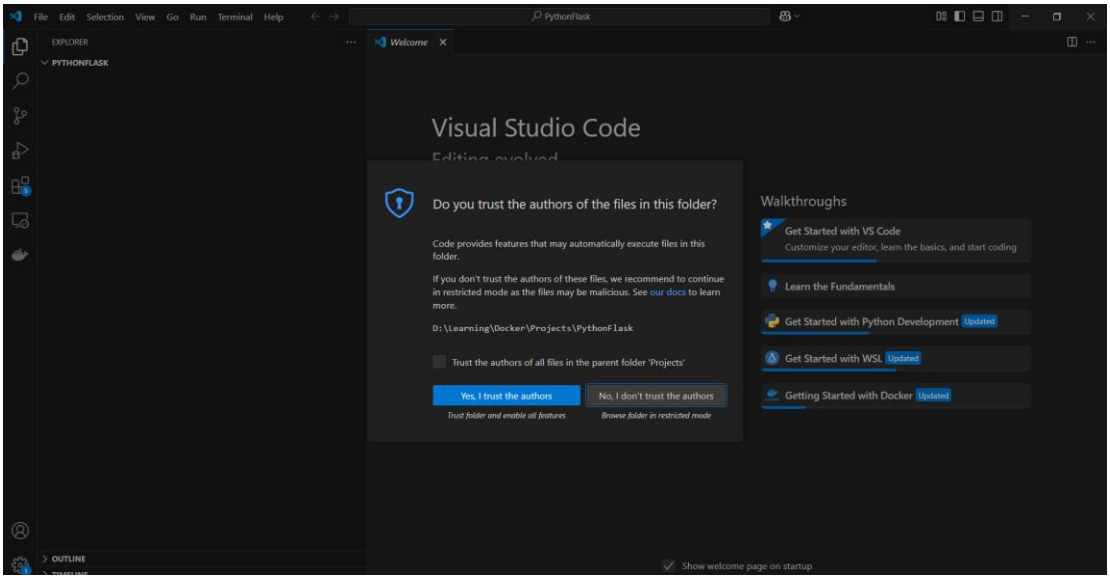


In the **Git Bash** command prompt, run the following commands to create a new directory for our Dockerized Python application at `D:\Learning\Docker\Projects` location and open **VS Code**. You can choose any location to create a new directory.

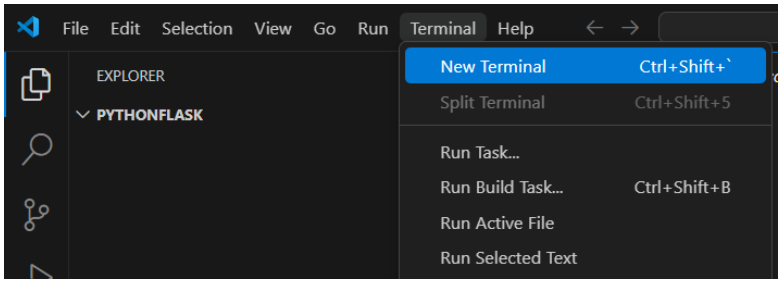
```
cd "D:\Learning\Docker\Projects"
mkdir PythonFlask
cd PythonFlask
code .
```



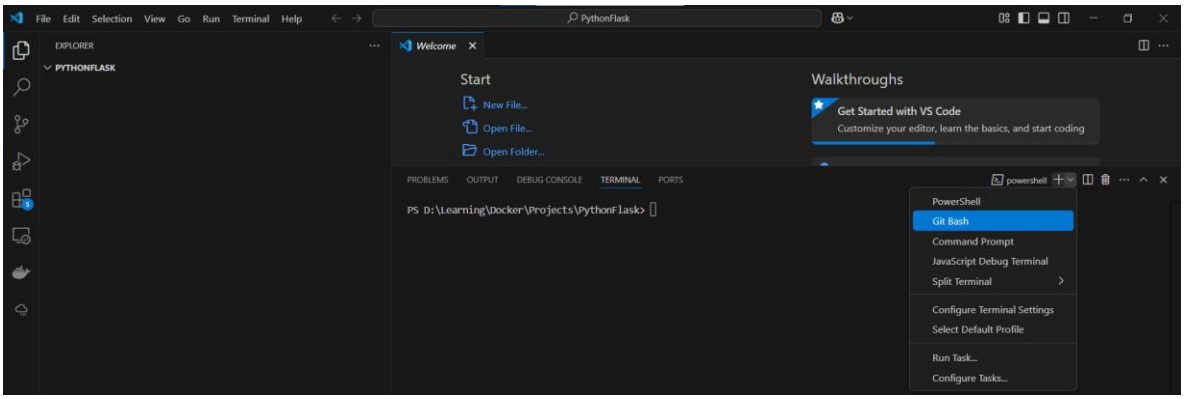
It opens **VS Code** application which might prompt you to confirm if you trust the authors of file in which case, click on **Yes, I trust the authors** button.



In VS Code, go to **Terminal** menu and select **New Terminal**.



By default, it uses **PowerShell** terminal, but let us choose **Git Bash** terminal from the dropdown in the **Terminal** tab below.



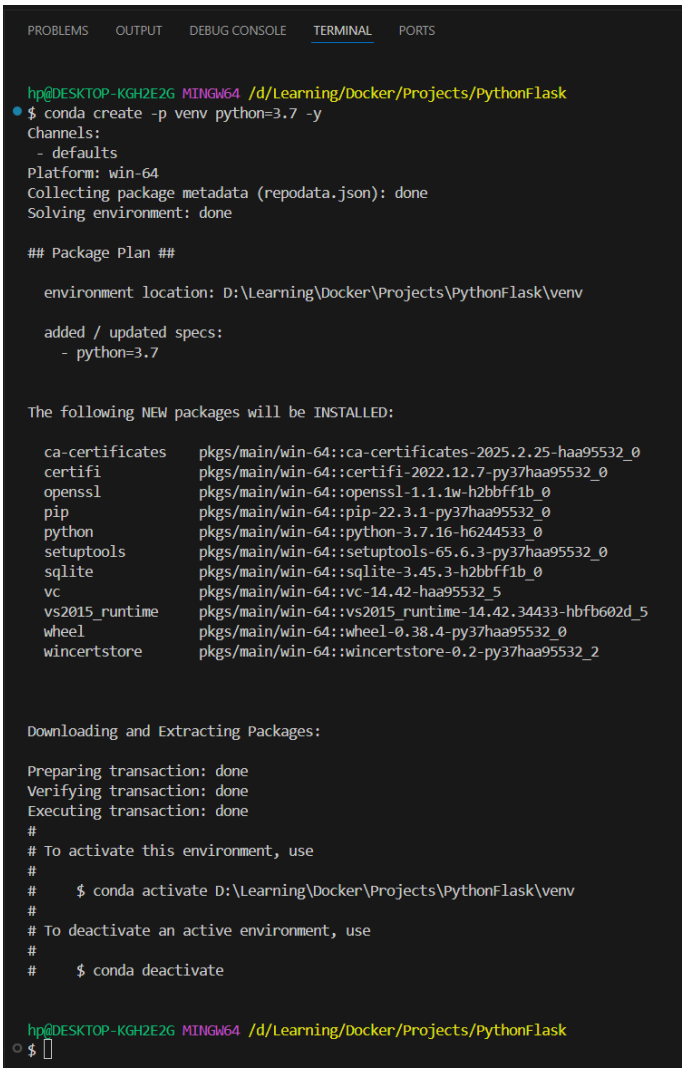
5.3. Create Virtual Environment:

Let us create a new virtual Python environment to have an isolated workspace for our application and install relevant packages without affecting the actual Python environment available in the system.

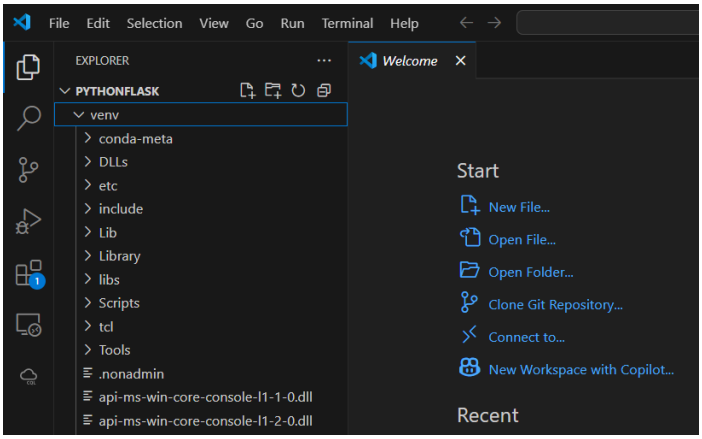
In the **Terminal** window, run the following command to create virtual environment with Python 3.7 version:

```
conda create -p venv python=3.7 -y
```

Note: Before executing this command, make sure that you have installed Anaconda distribution and set `Scripts` location of Anaconda installation in your `PATH` environment variable, otherwise you might encounter "*bash: conda: command not found*" error.



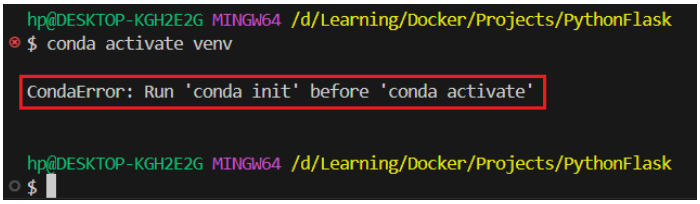
Once the Python virtual environment got created successfully, we can see that `venv` directory which contains Python related libraries and files got created under `PYTHONFLASK` project.



Now, run the following command to activate the new virtual environment in **Terminal** window:

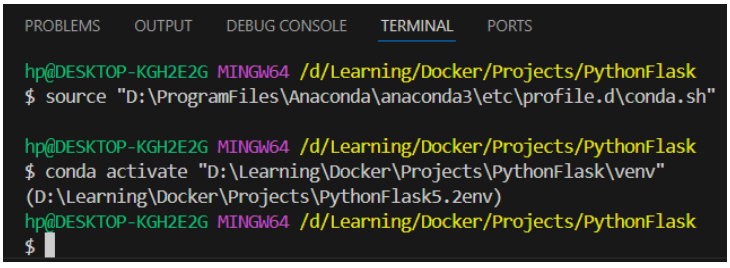
```
conda activate venv
```

When this command is executed, it might throw error CondaError: Run 'conda init' before 'conda activate' as below:



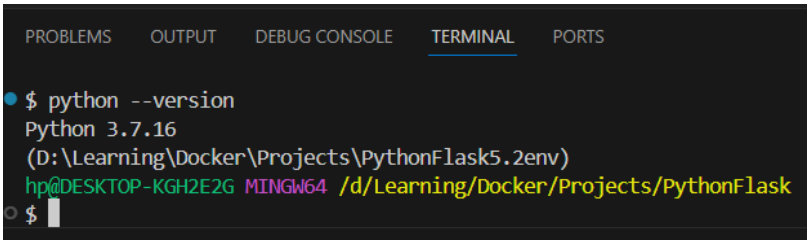
To fix this issue, source the location of `conda.sh` file from your Anaconda installation path and then activate the virtual environment using the following commands. In my case Anaconda was installed at `D:\ProgramFiles\Anaconda\anaconda3` and if your Anaconda installation path is different, replace this with your path:

```
source "D:\ProgramFiles\Anaconda\anaconda3\etc\profile.d\conda.sh"
conda activate "D:\Learning\Docker\Projects\PythonFlask\venv"
```



To verify if the virtual environment was activated successfully, check the version of Python:

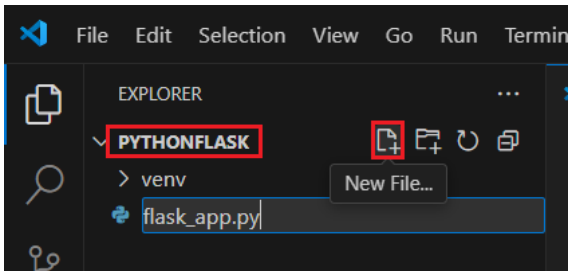
```
python --version
```



As you can see, it displayed **Python 3.7.16** version that was installed in virtual environment.

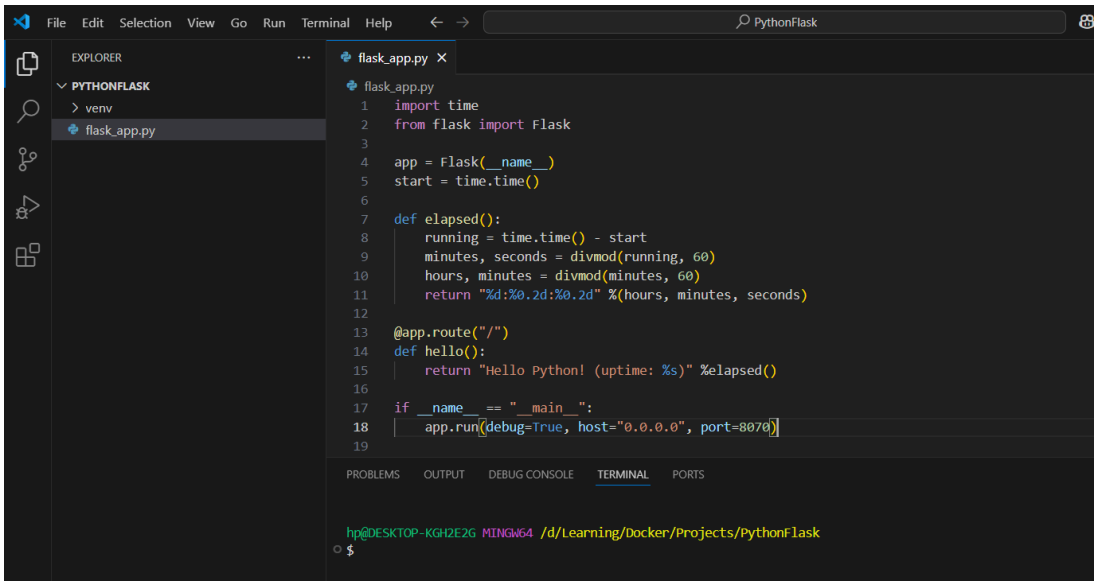
5.4. Create Python File:

In **VS Code** application, click on **New File** icon next to `PYTHONFLASK` project under **EXPLORER** and name the file as `flask_app.py` *(you can use any file name).*



Enter the following code in `flask_app.py` file and save it.

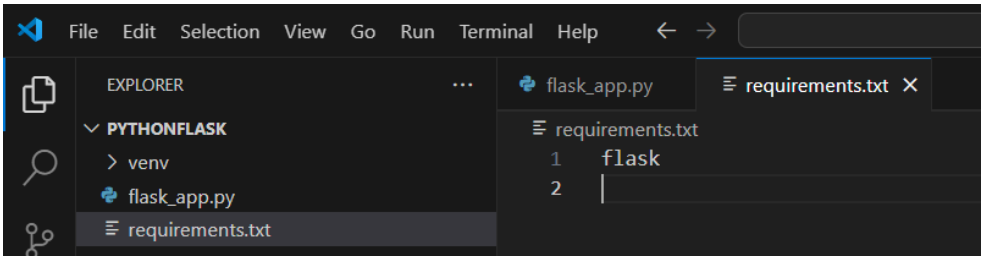
```
import time
from flask import Flask

app = Flask(__name__)
start = time.time()

def elapsed():
    running = time.time() - start
    minutes, seconds = divmod(running, 60)
    hours, minutes = divmod(minutes, 60)
    return "%d:%0.2d:%0.2d" %(hours, minutes, seconds)

@app.route("/")
def hello():
    return "Hello Python! (uptime: %s)" %elapsed()

if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=8070)
```



To make this Python code working successfully, the `flask` Python library should be installed. For this, create a new file under `PYTHONFLASK` project and name it as `requirements.txt` *(you can use any file name but this is the standard convention)*. This file is generally used to install multiple python libraries at once.

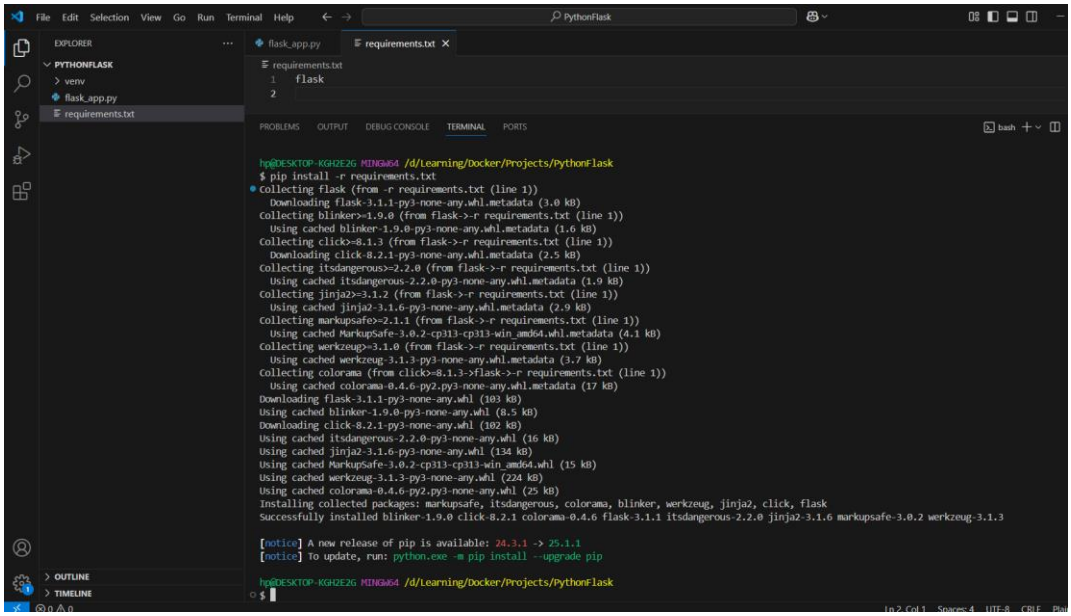
Enter the following line in `requirements.txt` file and save it.

```
flask
```



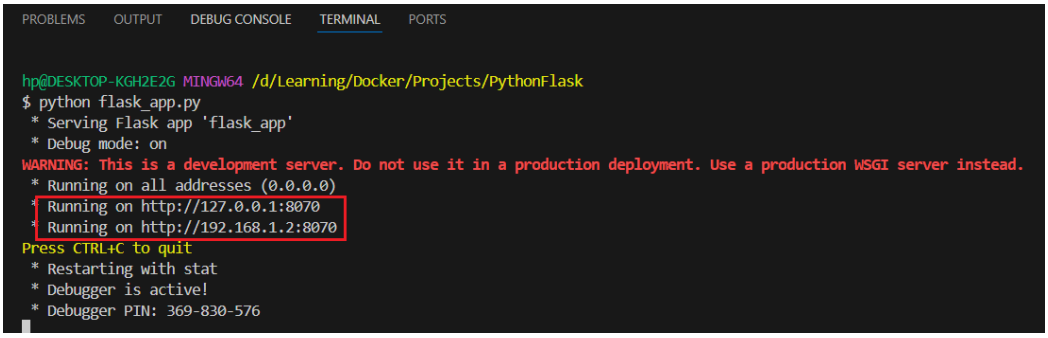
Next, run the following command in the VS Code Terminal window to install Python library requirements:

```
pip install -r requirements.txt
```

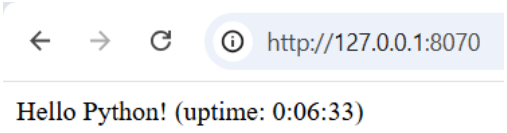


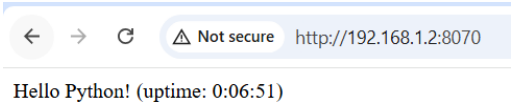
Now, execute the `flask_app.py` file using the following command:

```
python flask_app.py
```



When it runs successfully, it provides two localhost web URLs http://127.0.0.1:8070 and http://<machine_ip>:8070 where our Flask application is running. Open any URL to see the output of our python application:





It displays the uptime which gets update upon every refresh of the browser URL.

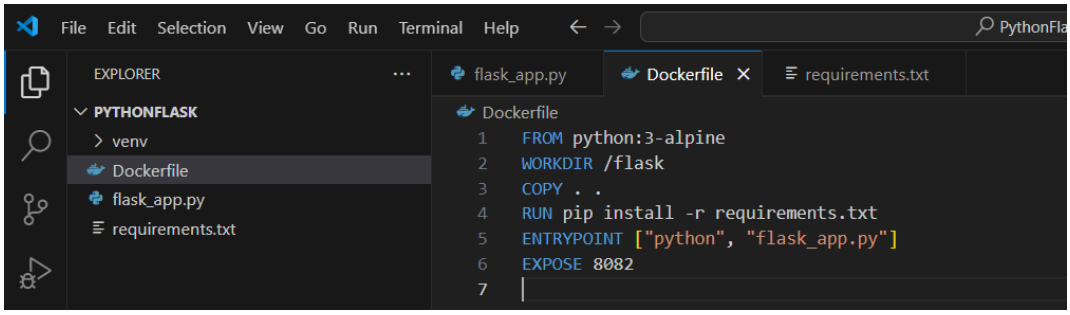
Press **Ctrl+C** on the VS Code terminal to exit out of Flask application.

5.5. Create Docker File:

Now, create a Docker file under `PYTHONFLASK` project and name it as `Dockerfile` without any extension *(Docker looks for this file name while building an image, however a different file name can also be used and must be specified during Docker Build)*.

Enter the following instructions in `Dockerfile` file and save it. This the basic Docker code for a simple Python application.

```
FROM python:3-alpine
WORKDIR /flask
COPY . .
RUN pip install -r requirements.txt
ENTRYPOINT ["python", "flask_app.py"]
EXPOSE 8082
```

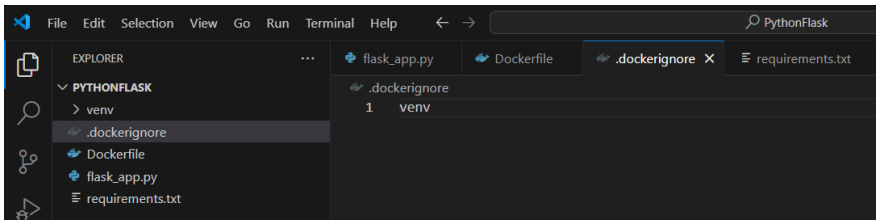


The above Docker instructions perform the following actions:

- `FROM python:3-alpine` instruction pulls the base application image of `python` which is tagged as `3-alpine` version. Here, `alpine` version is used since it uses light weight Linux distribution that is smaller in size which makes `alpine` version image smaller compared to non-alpine image. You can check the compressed size of python:3-alpine which is ~15MB vs python:3 image which is ~365MB in Docker Hub.
- `WORKDIR /flask` instruction creates a directory named `flask` and set it as current working directory in Docker container image.
- `COPY . ./` instruction copies all files from the present working directory in local host to the current directory which is `/flask` in Docker container image.
- `RUN pip install -r requirements.txt` instruction installs python libraries specified in `requirements.txt` file.
- `ENTRYPOINT ["python3", "flask_app.py"]` instruction runs the command when Docker container is started, here the command to execute is `python3 flask_app.py`
- `EXPOSE 8082` instruction exposes the Docker application port to outside of container.

Note that there is a `venv` sub-folder in PYTHONFLASK project and it is not required to copy this sub-folder to the Docker container. To ignore this folder, create a file named `.dockerignore` and write the following line and save it so that Docker ignores all files/folders listed in `.dockerignore` file while building image.

```
venv
```



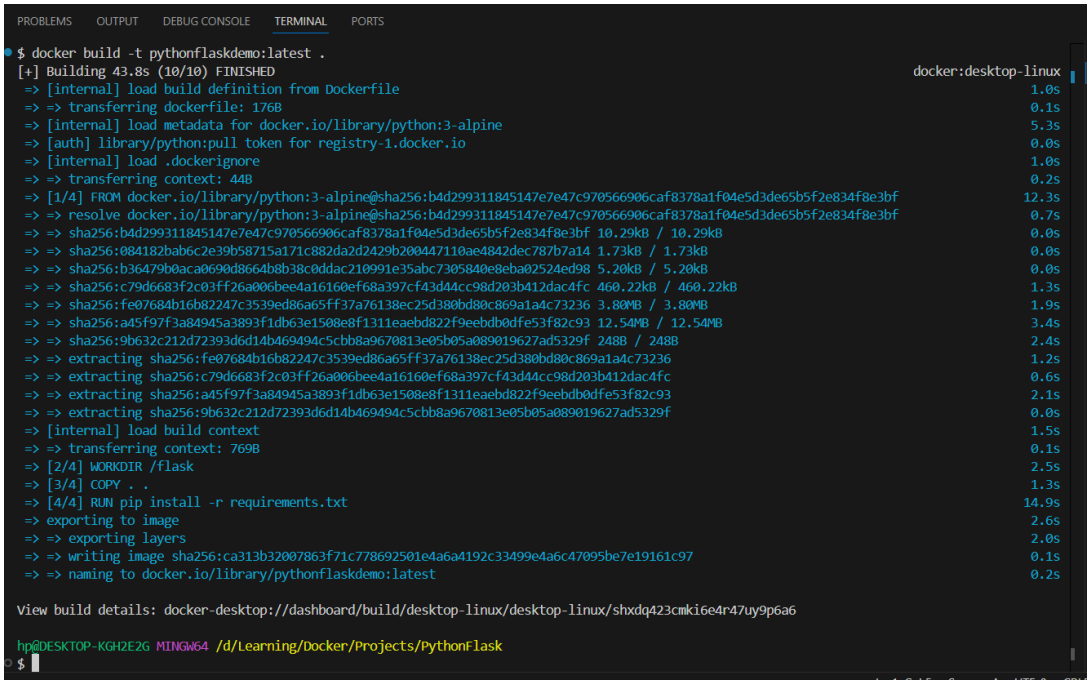
5.6. Build Docker Image:

Now, open a new terminal in VS Code and run the following command to create a docker image. In this command, the `.` indicates the current working directory where `Dockerfile` is present.

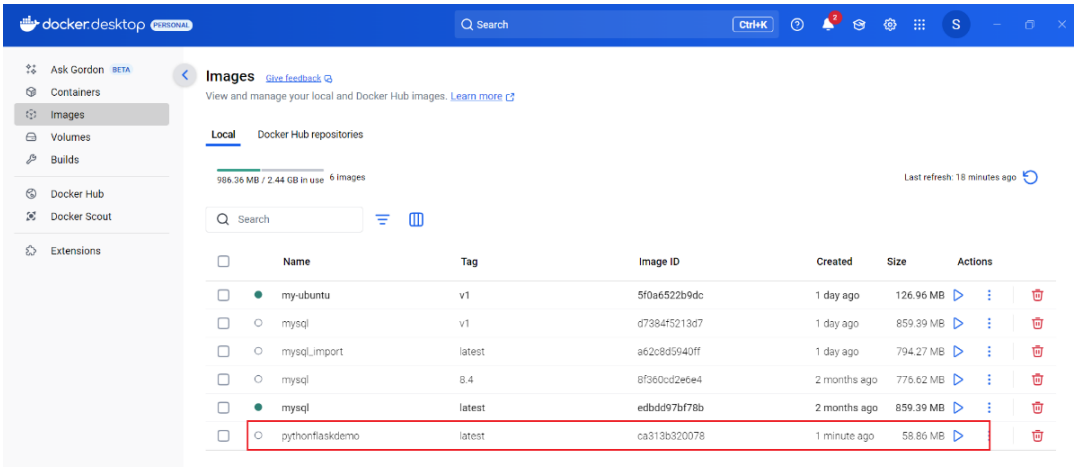
```
docker build -t pythonflaskdemo:latest .
```

If you have used a different Dockerfile name, then specify it in the below command:

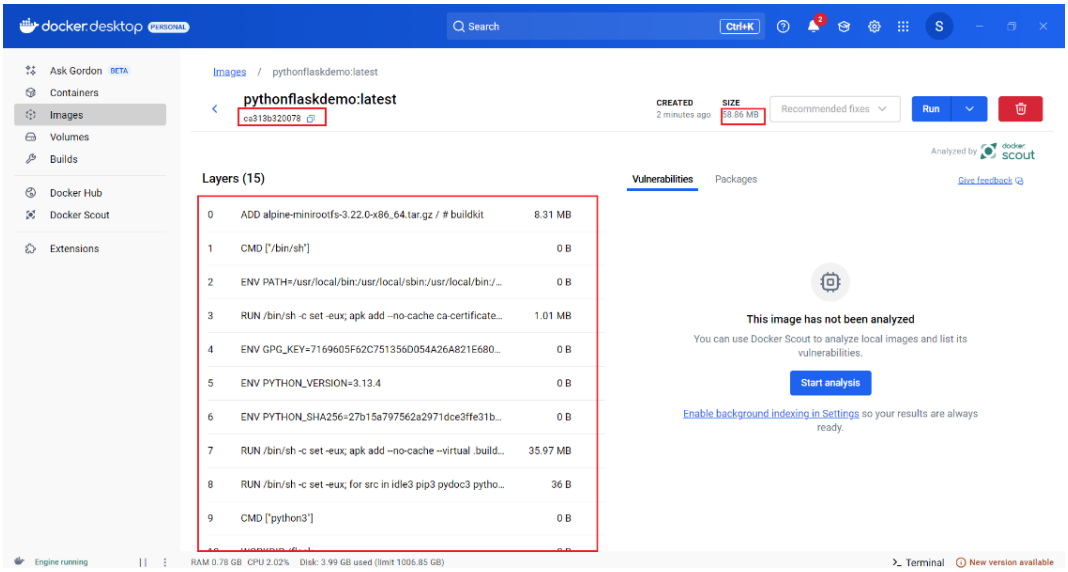
```
docker build -t pythonflaskdemo:latest . -f <docker_file>
```



Go to **Docker Desktop** and see that `pythonfalskdemo` image has been created with `latest` tag name in **Images** view.



Click on `pythonfalskdemo` image and it displays image ID, image size and layers created to makeup this image.



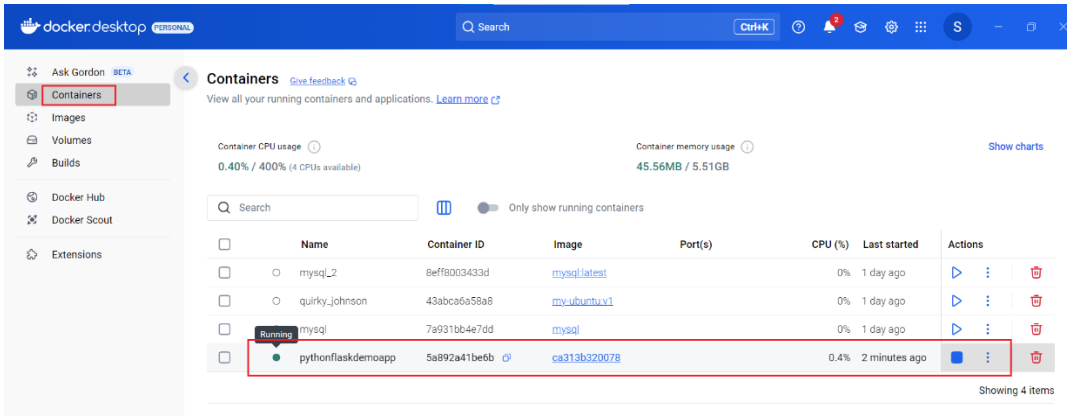
5.7. Run Docker Image:

Now, run the Docker image using the following command in VS Code terminal. In my case, the image Id is `ca313b320078` as displayed in my Docker Desktop.

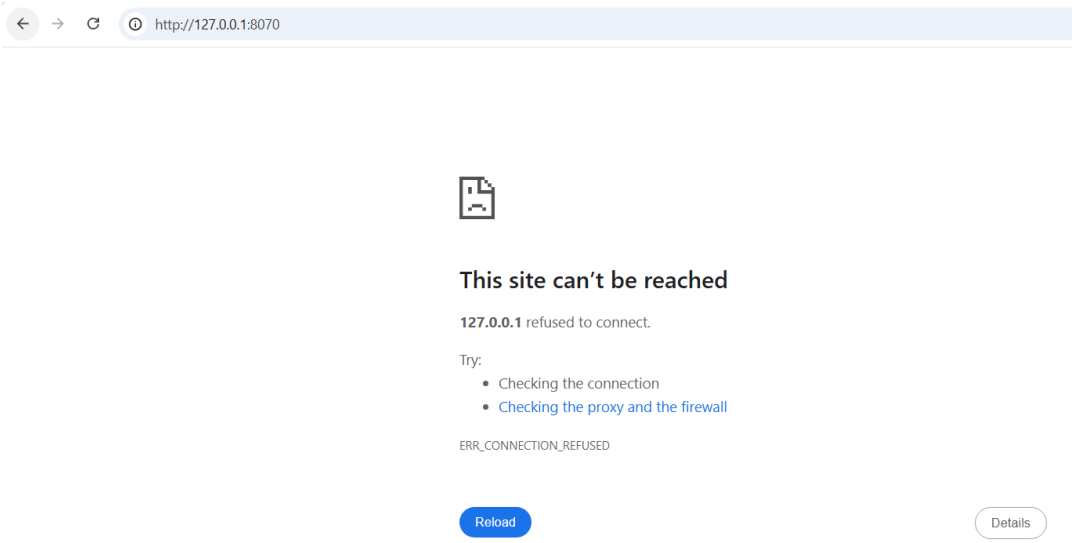
```
docker run --name pythonflaskdemoapp ca313b320078
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/PythonFlask
$ docker run --name pythonflaskdemoapp ca313b320078
 * Serving Flask app 'flask_app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:8070
 * Running on http://172.17.0.2:8070
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 115-701-637
```

Go to **Docker Desktop** and see that a new container called `pythonfalskdemoapp` has been created and running in **Containers** view.

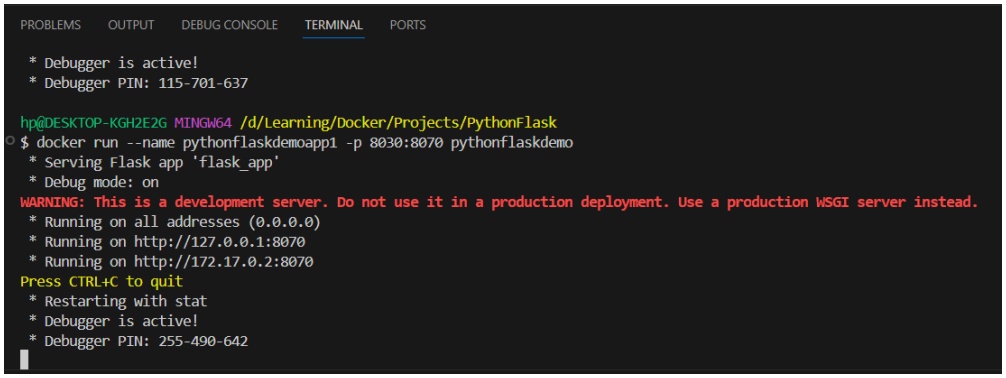


Now, hit the URL http://127.0.0.1:8070/ produced by the docker image upon running but it displays site cannot be reached error because the flask application port is not mapped to run outside the Docker container.

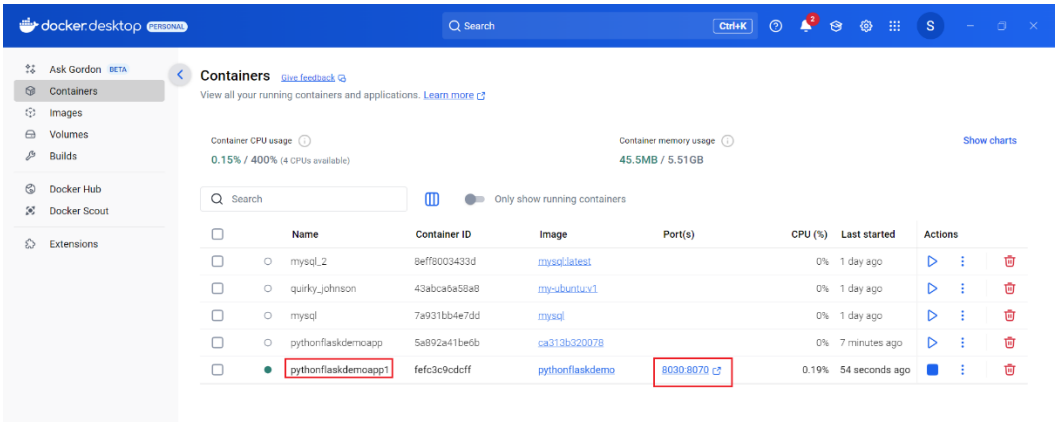


To overcome this problem, run the Docker container using the following command which maps the flask application port `8070` to port `8030` in local system. Press **Ctrl + C** to exit out of the previous flask execution before executing this command:

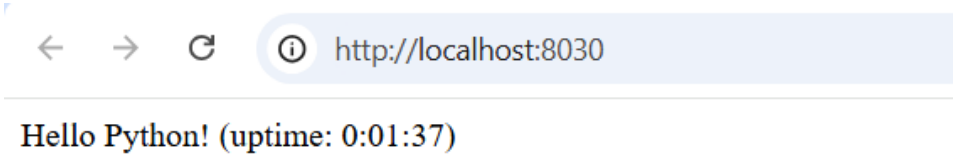
```
docker run --name pythonflaskdemoapp1 -p 8030:8070 pythonflaskdemo
```



Go to Docker Desktop which shows a new container called `pythonflaskdemoapp1` has been created and running on 8030 port locally



Now, hit the URL http://localhost:8030/ produced by the docker container which displays the uptime and gets changed upon every refresh of the browser URL.



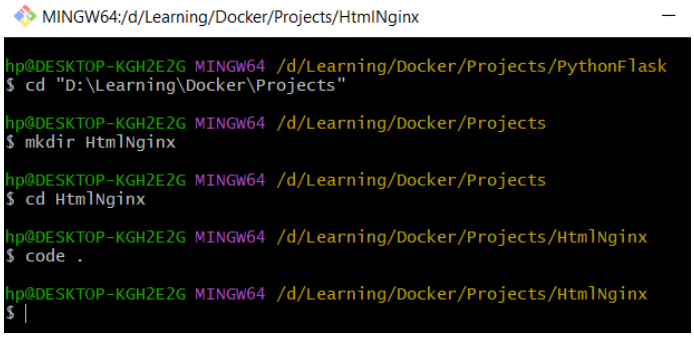
6. Create HTML Docker Application:

We will create a simple **HTML** containerized application using **Nginx** web server to run from Docker. **Nginx** is a web server software that also acts as a reverse proxy, load balancer, mail proxy and HTTP cache.

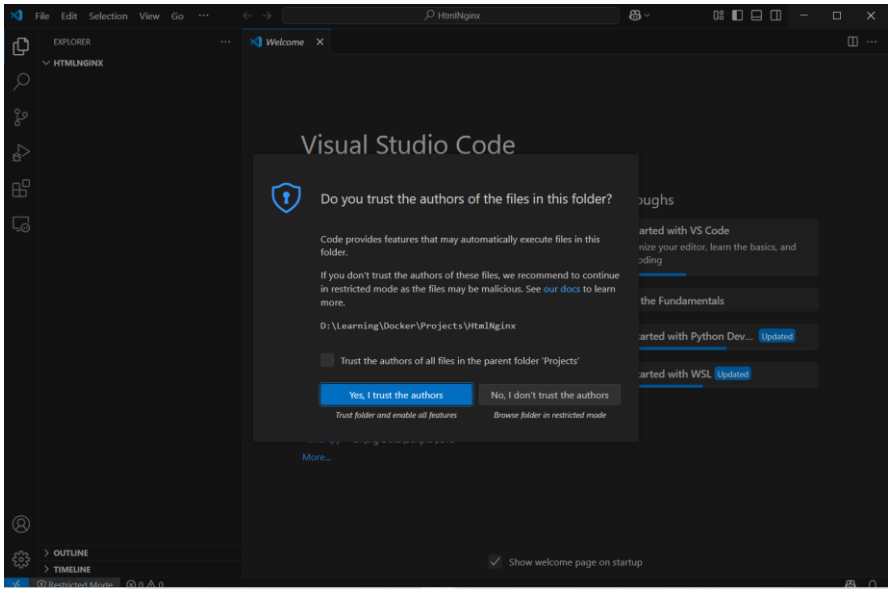
6.1. Launch VS Code:

In the **Git Bash** command prompt, run the following commands to create a new directory at `D:\Learning\Docker\Projects` location and open **VS Code**. You can choose any location to create a new directory.

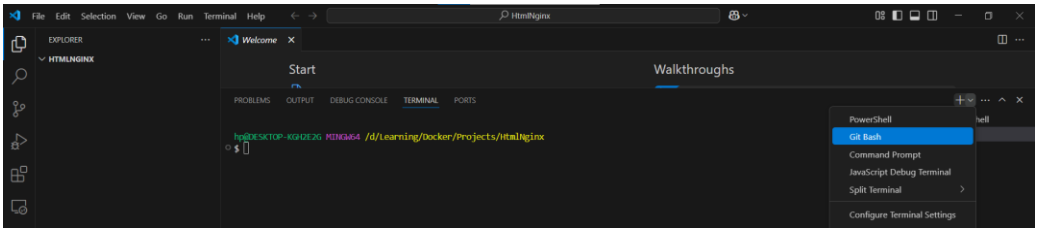
```
cd "D:\Learning\Docker\Projects"
mkdir HtmlNginx
cd HtmlNginx
code .
```



It opens **VS Code** application which might prompt you to confirm if you trust the authors of file in which case, click on **Yes, I trust the authors** button.

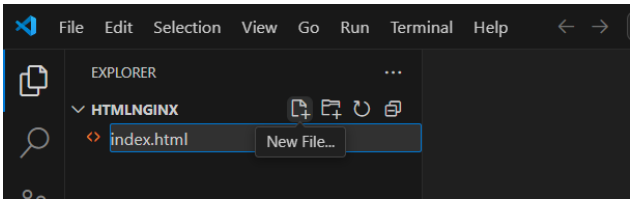


In VS Code, go to **Terminal** menu and select **New Terminal** and choose **Git Bash** terminal from the dropdown in the **Terminal** tab below.



6.2. Create HTML File:

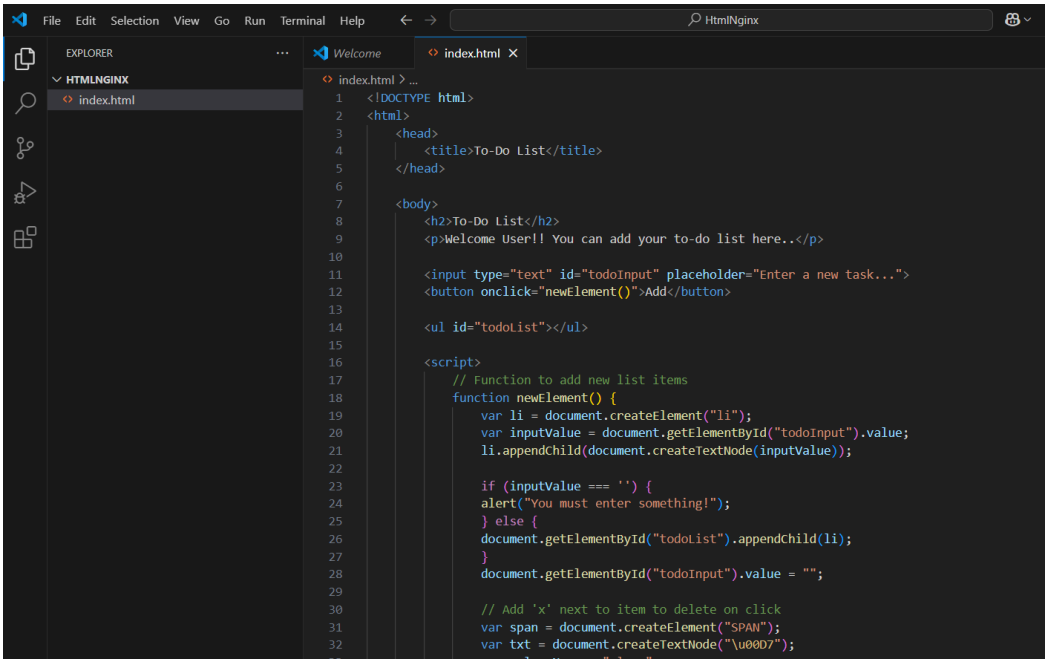
In **VS Code** application, click on **New File** icon next to `HTMLNGINX` project under **EXPLORER** and name the file as `index.html`



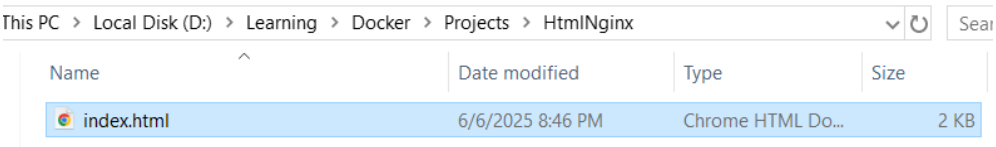
Enter the following code in `index.html` file and save it.

```
<!DOCTYPE html>
<html>
	<head>
	  <title>To-Do List</title>
	</head>

	<body>
	  <h2>To-Do List</h2>
	  <p>Welcome User!! You can add your to-do list here..</p>

	  <input type="text" id="todoInput" placeholder="Enter a new task...">
	  <button onclick="newElement()">Add</button>

	  <ul id="todoList"></ul>

	  <script>
		// Function to add new list items
		function newElement() {
			var li = document.createElement("li");
			var inputValue =
document.getElementById("todoInput").value;
			li.appendChild(document.createTextNode(inputValue));

			if (inputValue === '') {
			alert("You must enter something!");
			} else {
```

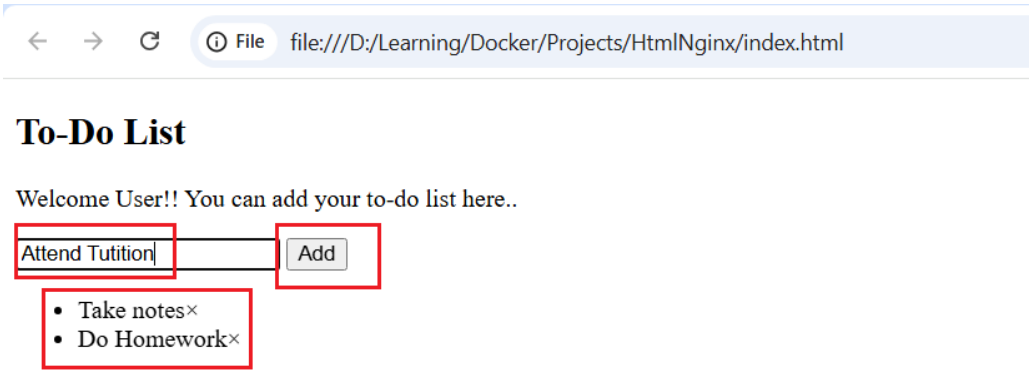
```
                        document.getElementById("todoList").appendChild(li);
                        }
                        document.getElementById("todoInput").value = "";

                        // Add 'x' next to item to delete on click
                        var span = document.createElement("SPAN");
                        var txt = document.createTextNode("\u00D7");
                        span.className = "close";
                        span.appendChild(txt);
                        li.appendChild(span);

                        span.onclick = function() {
                                var div = this.parentElement;
                                div.remove();
                        }
                }
        </script>
      </body>
</html>
```



To check if the above code is working, open `index.html` file in `D:\Learning\Docker\Projects\HtmlNginx` location.



It opens a webpage where we can enter tasks and click on **Add** button. It then displays the entered tasks below. To remove the added task, click on **x** next to the task.

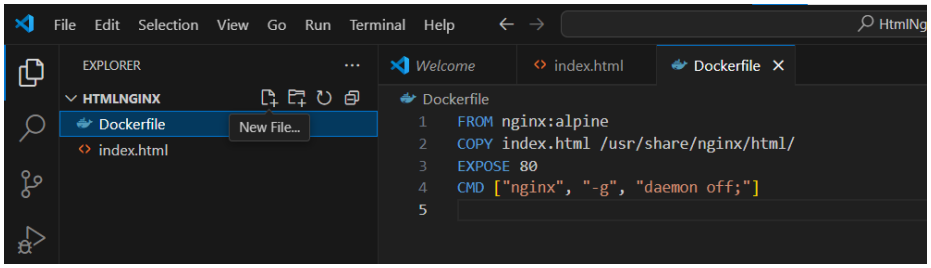


6.3. Create Docker File:

Now, create a Docker file under `HTMLNGINX` project and name it as `Dockerfile` without any extension *(Docker looks for this file name while building an image but if you wish to use a different name, make sure to specify it to Docker Build)*.

Enter the following instructions in `Dockerfile` file and save it. This the basic Docker code for a simple HTML application.

```
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```



The above Docker instructions perform the following actions:

- `FROM nginx:alpine` instruction pulls the base application image of `nginx` tagged as `alpine` version. Here, `alpine` version is used since it uses light weight Linux distribution that is smaller in size which makes `alpine` version image smaller compared to non-alpine image.
- `COPY index.html /usr/share/nginx/html/` instruction copies `index.html` file from the present working directory in local host to `/usr/share/nginx/html`

directory which is a default location for serving HTML file by Nginx Docker container. Note that Nginx by default refers to `/etc/nginx/conf.d/default.conf` file in Nginx container.

- `EXPOSE 80` instruction exposes the Nginx port to outside of container.
- `CMD ["nginx", "-g", "daemon off;"]` instruction runs the command when Docker container is started and this command ensures to run nginx server foreground.

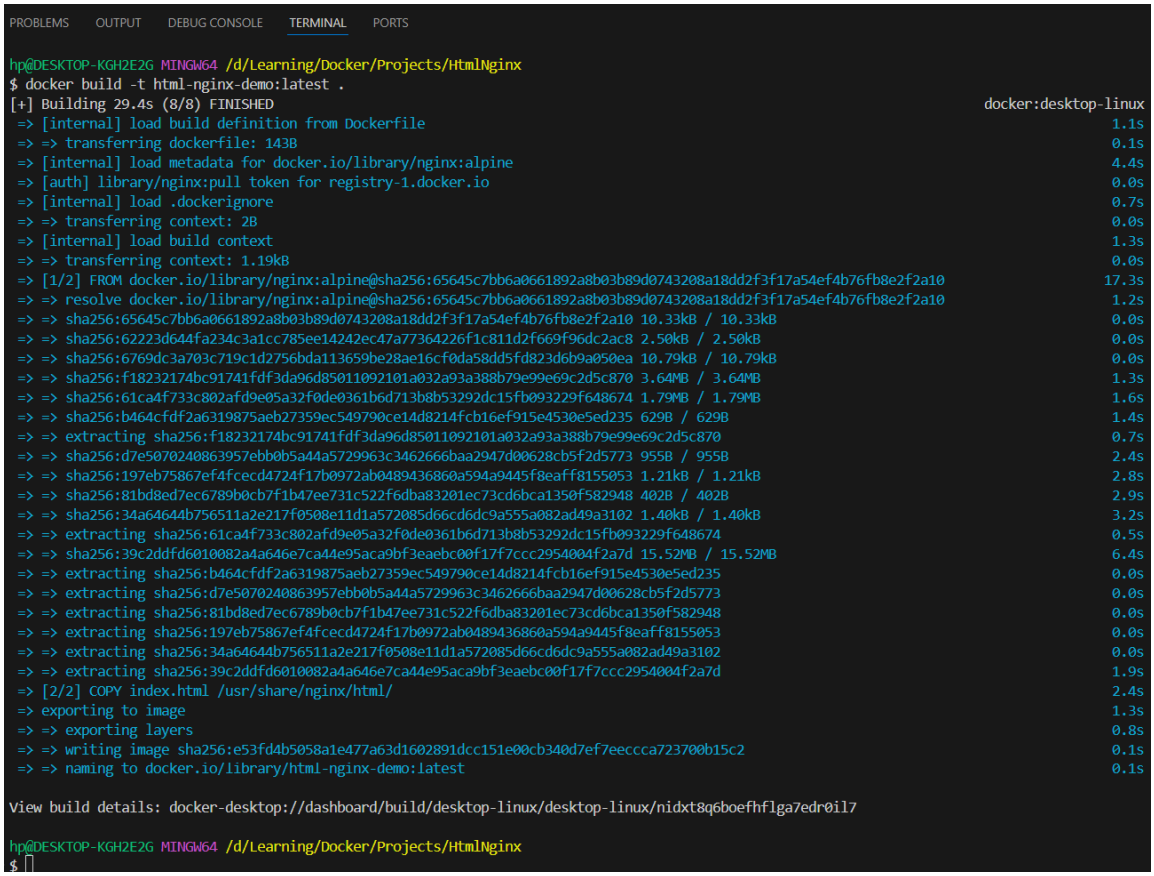
6.4. Build Docker Image:

Now, open a new terminal in VS Code and run the following command to create a docker image. In this command, the `.` indicates the current working directory where `Dockerfile` is present.

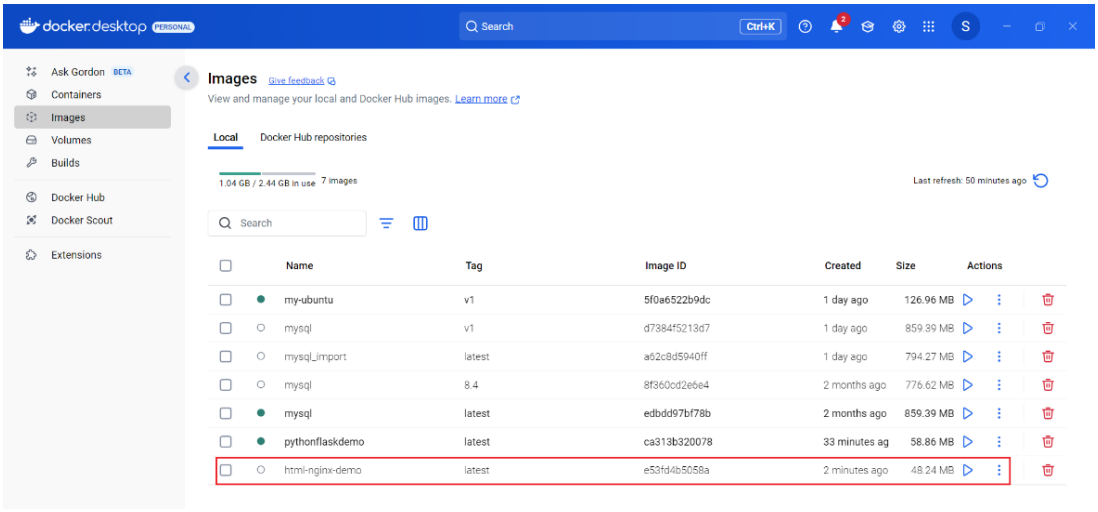
```
docker build -t html-nginx-demo:latest .
```

If you have used a different Dockerfile name, then specify it in the below command:

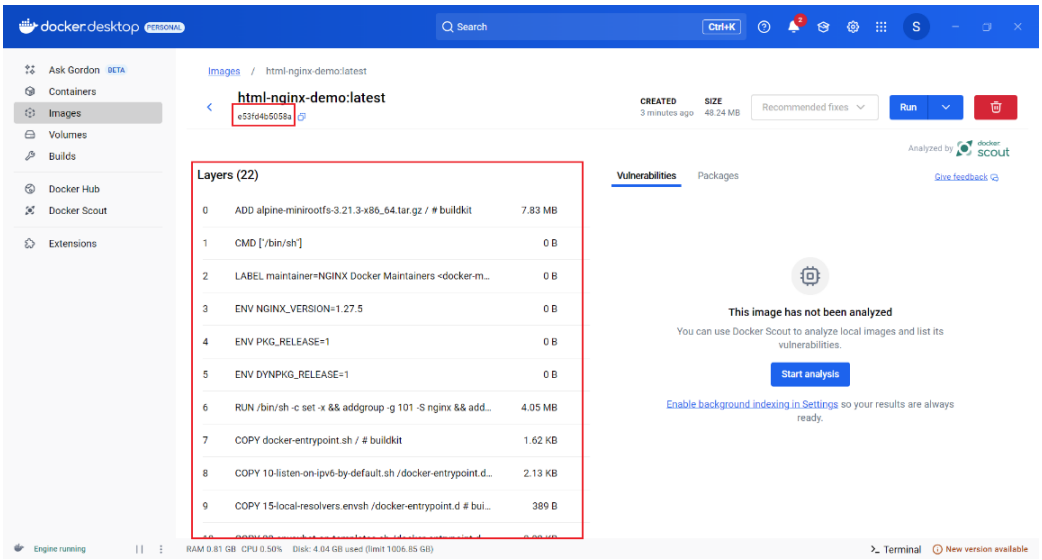
```
docker build -t html-nginx-demo:latest . -f <docker_file>
```



Go to **Docker Desktop** and see that `html-nginx-demo` image has been created with `latest` tag name in **Images** view.



Click on `html-nginx-demo` image and it displays image ID, image size and layers created to makeup this image.



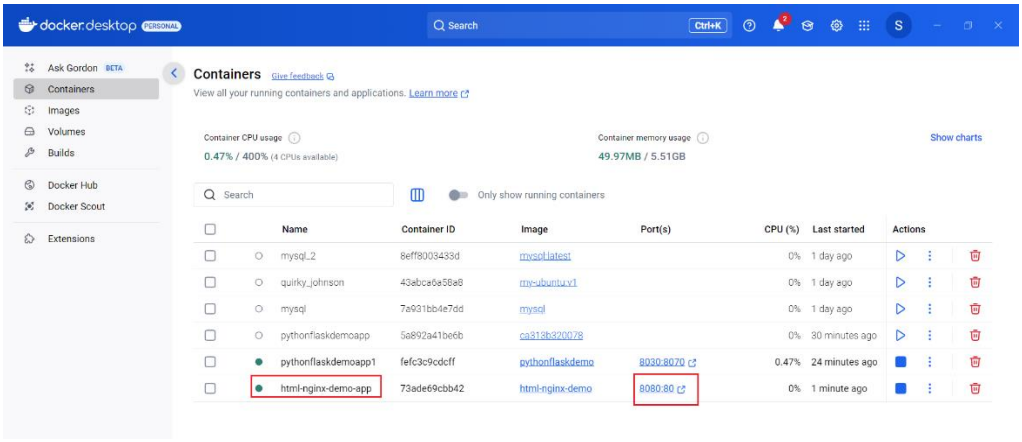
6.5. Run Docker Image:

Now, execute the following commands to run the docker image which creates a docker container.

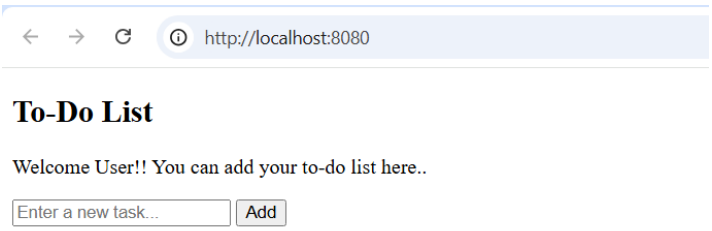
```
docker run --name html-nginx-demo-app -p 8080:80 html-nginx-demo
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/HtmlNginx
$ docker run --name html-nginx-demo-app -p 8080:80 html-nginx-demo
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2025/06/06 15:32:17 [notice] 1#1: using the "epoll" event method
2025/06/06 15:32:17 [notice] 1#1: nginx/1.27.5
2025/06/06 15:32:17 [notice] 1#1: built by gcc 14.2.0 (Alpine 14.2.0)
2025/06/06 15:32:17 [notice] 1#1: OS: Linux 6.6.87.1-microsoft-standard-WSL2
2025/06/06 15:32:17 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2025/06/06 15:32:17 [notice] 1#1: start worker processes
2025/06/06 15:32:17 [notice] 1#1: start worker process 30
2025/06/06 15:32:17 [notice] 1#1: start worker process 31
2025/06/06 15:32:17 [notice] 1#1: start worker process 32
2025/06/06 15:32:17 [notice] 1#1: start worker process 33
```

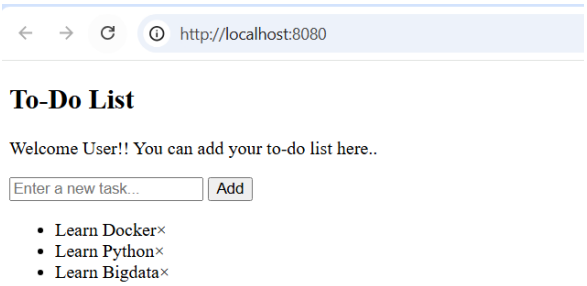
Go to **Docker Desktop** and see that a new container called `html-nginx-demo-app` has been created and running on port `8080` in **Containers** view.



Now, hit the URL http://localhost:8080/ produced by the docker container which launches the web page.



In this webpage, we should be able to create a To-do list by adding or deleting items as designed.



6.6. Modify HTML Code:

Let us change the webpage little bit by adding “Launched on: 5/24/2025” line
In the VS Code, open `index.html` file.

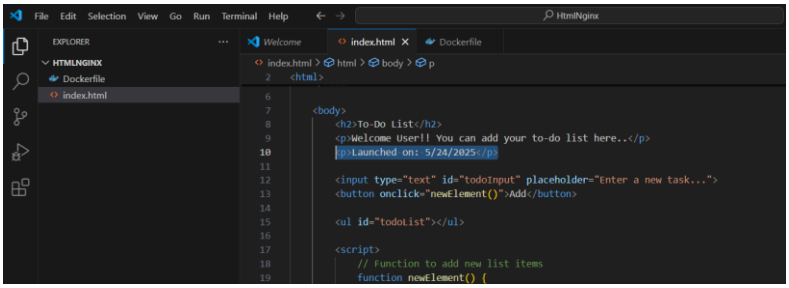
Change lines from:

```
        <body>
                <h2>To-Do List</h2>
                <p>Welcome User!! You can add your to-do list here..</p>

                <input type="text" id="todoInput" placeholder="Enter a new
task...">
                <button onclick="newElement()">Add</button>
```

To:

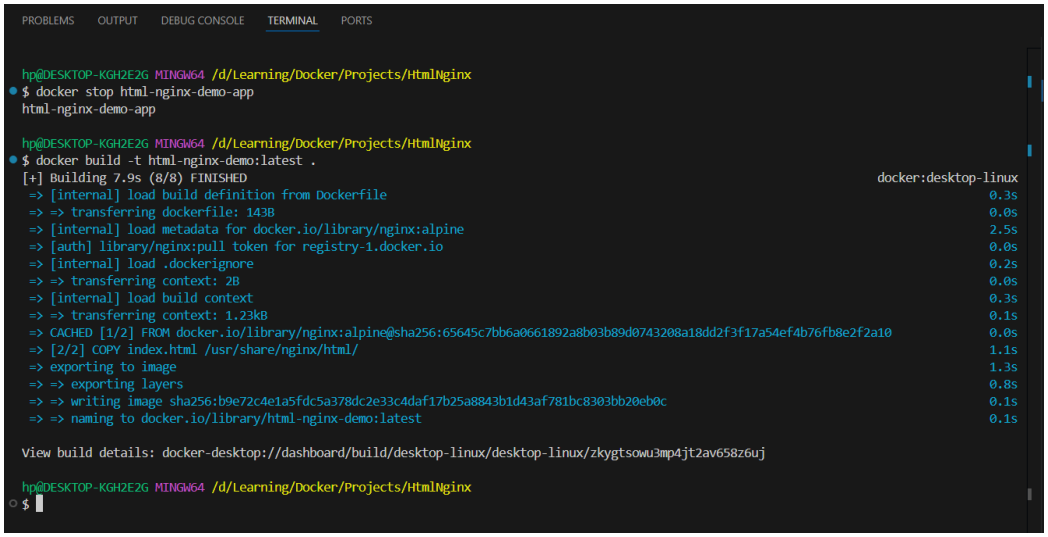
```
        <body>
                <h2>To-Do List</h2>
                <p>Welcome User!! You can add your to-do list here..</p>
        <p>Launched on: 5/24/2025</p>

                <input type="text" id="todoInput" placeholder="Enter a new
task...">
                <button onclick="newElement()">Add</button>
```

Save changes in `index.html` file.

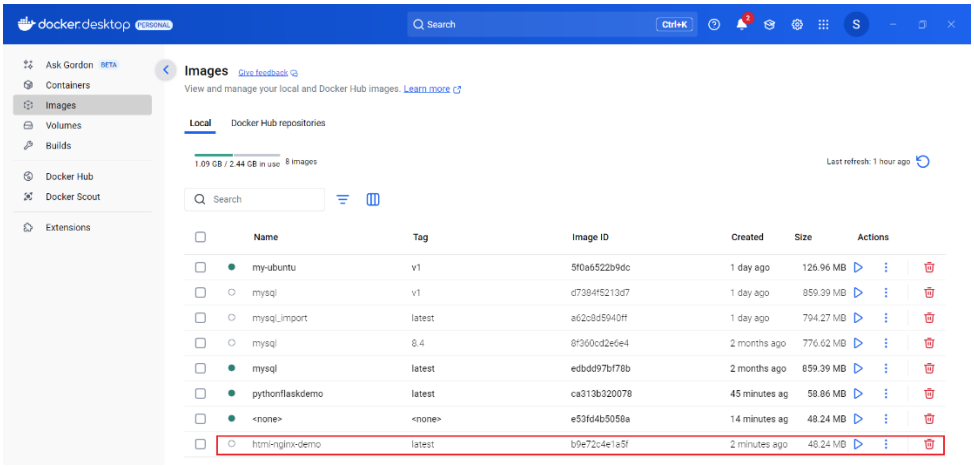


Now, open another terminal and run the following commands to stop the running container and rebuild the image file

```
docker stop html-nginx-demo-app
docker build -t html-nginx-demo:latest .
```



In Docker Desktop, you can see that the image was rebuilt.

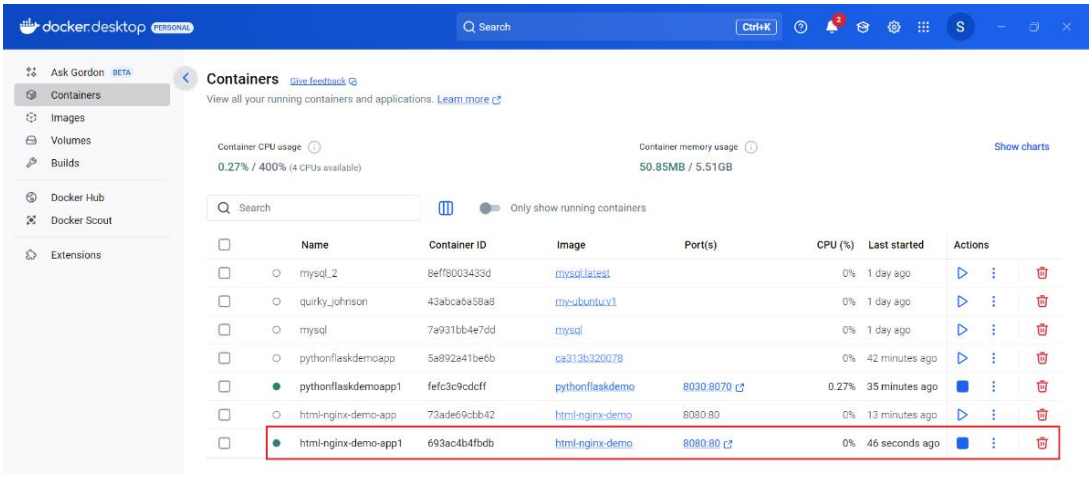


Now, let us run the container in the detached mode (to run container at background) with the new image by executing the following command

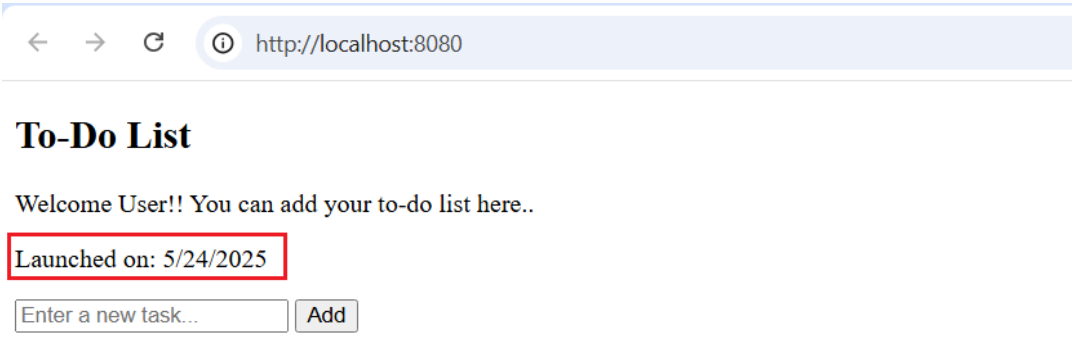
```
docker run --name html-nginx-demo-app1 -dp 8080:80 html-nginx-demo
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

$ docker run --name html-nginx-demo-app1 -dp 8080:80 html-nginx-demo
693ac4b4fbdb145cbb9ae9f349779fe9017b8e2cd47524321ae6ae4a16cebfc3

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/HtmlNginx
$
```

In Docker Desktop, a new container called `html-nginx-demo-app1` has been created and running on port `8080` in the Docker desktop.



Now, hit the URL http://localhost:8080/ produced by the docker container and it launches the web application with new changes.

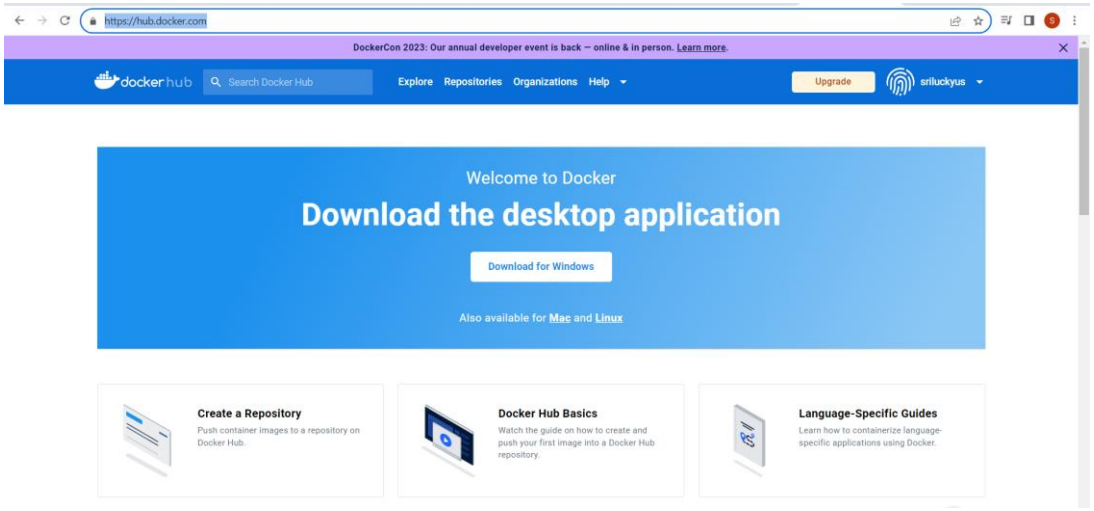


7. Publish Docker Image to Docker Hub:

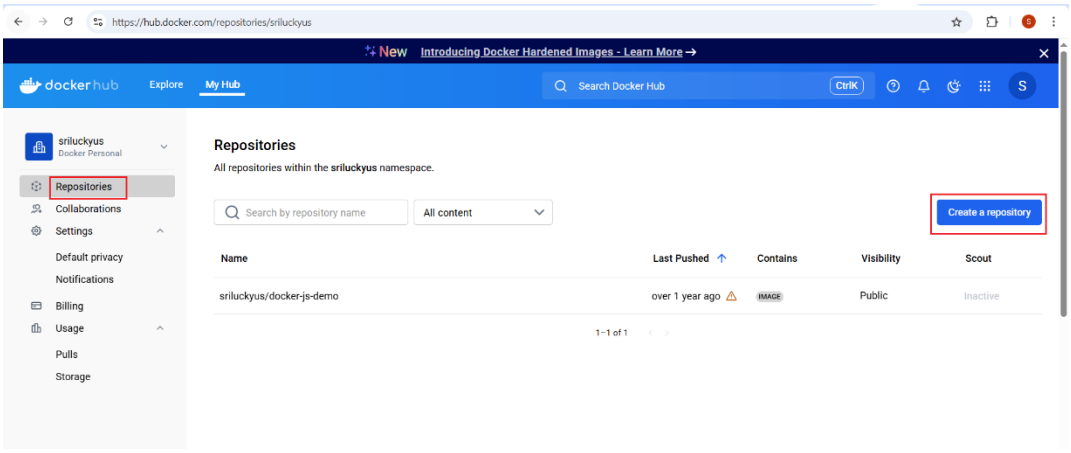
In order the above **HTML Nginx** Docker image available for everyone, the image should be published to the Docker Hub. This Docker image can be shared in other cloud services as well such as AWS, ECRS, Azure Docker container.

7.1. Create Repository:

First, create a repository in Docker Hub.
Login to https://hub.docker.com/



Select **Repositories** tab and click on **Create repository.**

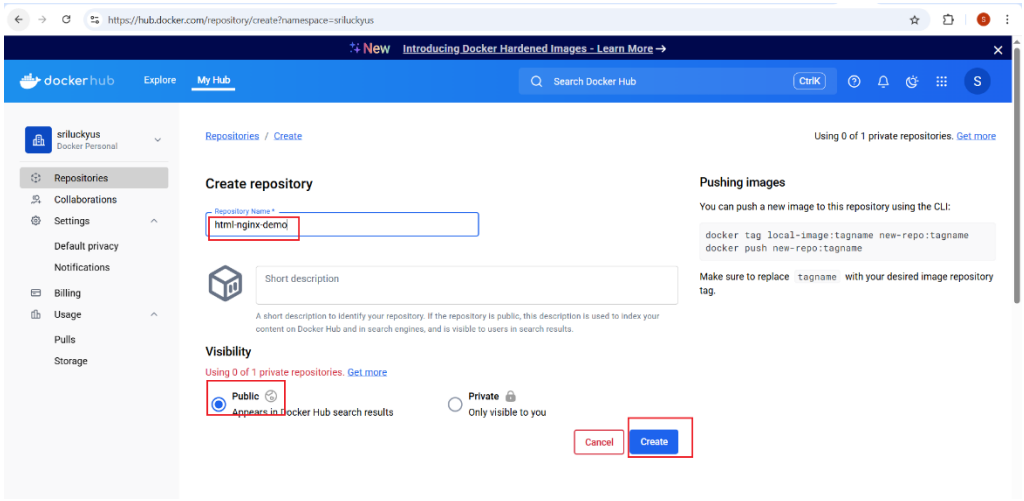


Under **Create repository** page:

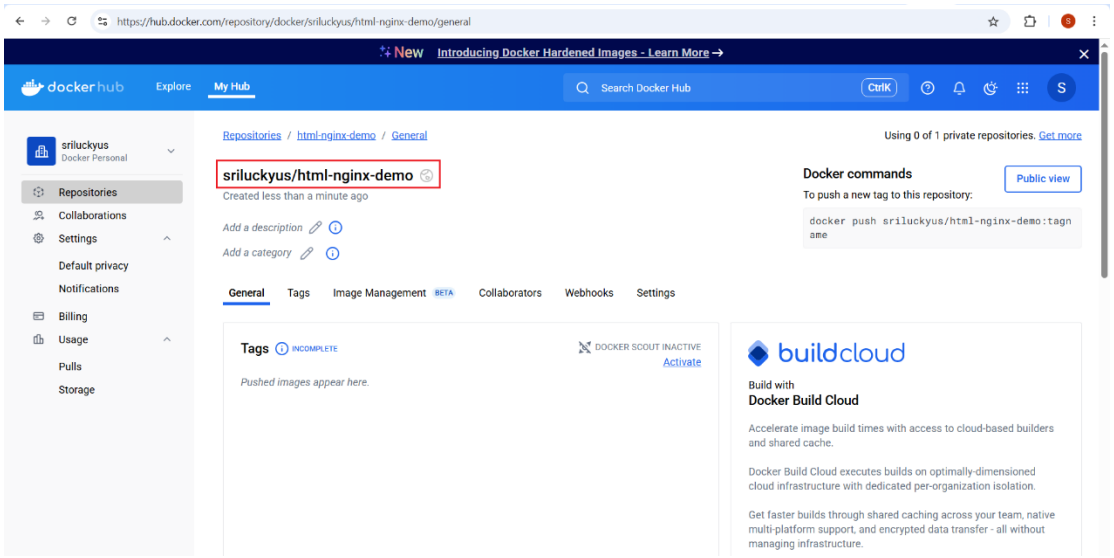
- Provide the **Repository Name** *(it is always good to keep the repository name same as the folder name)*. Here, we are trying to publish HTML Docker application which has the

Docker image name as **`html-nginx-demo`**, provide the same name as the Docker Hub repository name.

- Select the **Visibility** as Public so that it is available for anyone who has access to Docker Hub and then click on **Create** button.



A Docker repository has been created.

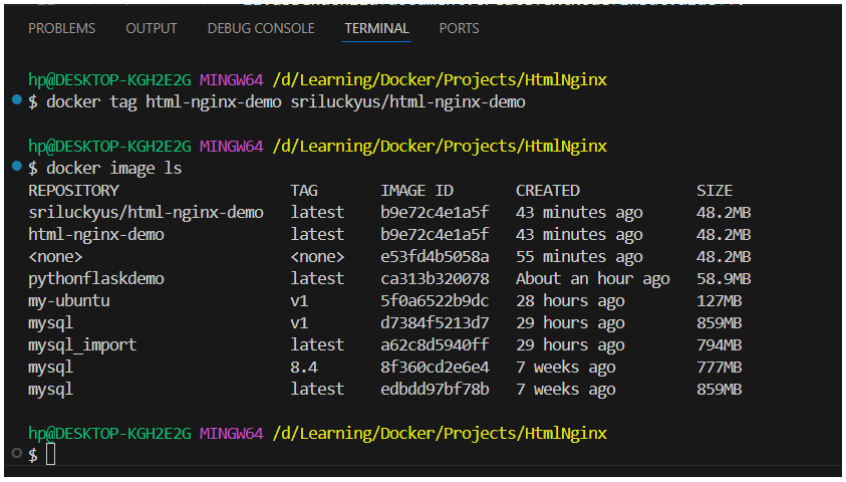


7.2. Tag Image to User Repository:

When the image needs to be pushed to Docker Hub repository, it should have a specific naming convention such as `docker_user_name/docker_repo_name`.

Go to VS Code and run the below command to tag the existing image name to the new image name. Here, I am pushing the image to my Docker Repo and so tagging as **sriluckyus**/html-nginx-demo.

```
docker tag html-nginx-demo sriluckyus/html-nginx-demo
docker image ls
```



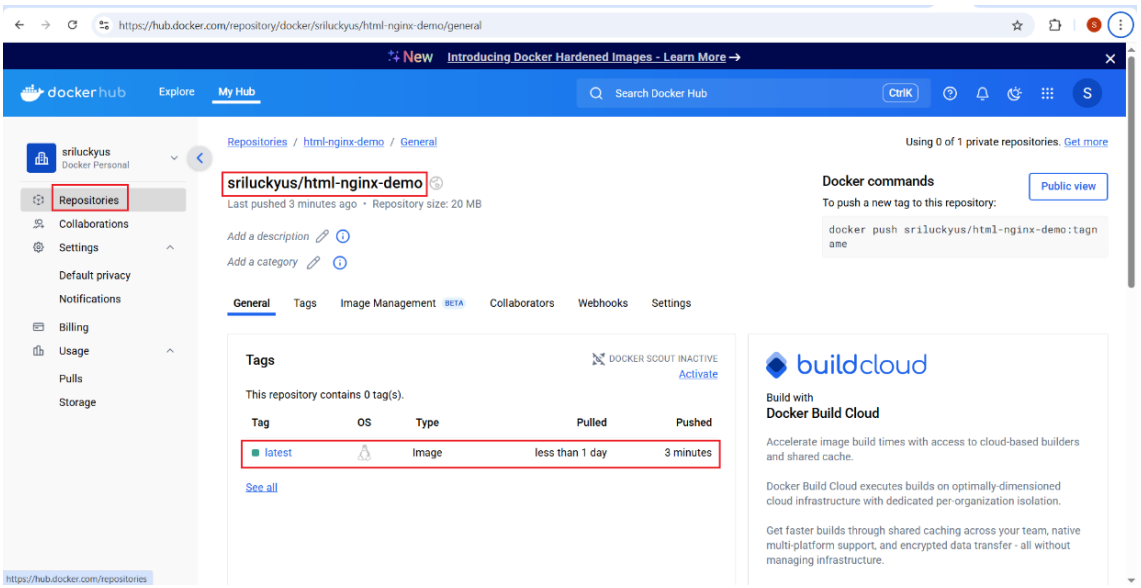
7.3. Login & Push User Repo Docker Image:

Now, run the below commands to login to Docker Hub and push the image. In these commands, you need to add your respective Docker Hub **username**:

```
docker login -u "username" docker.io
docker push username/html-nginx-demo
```

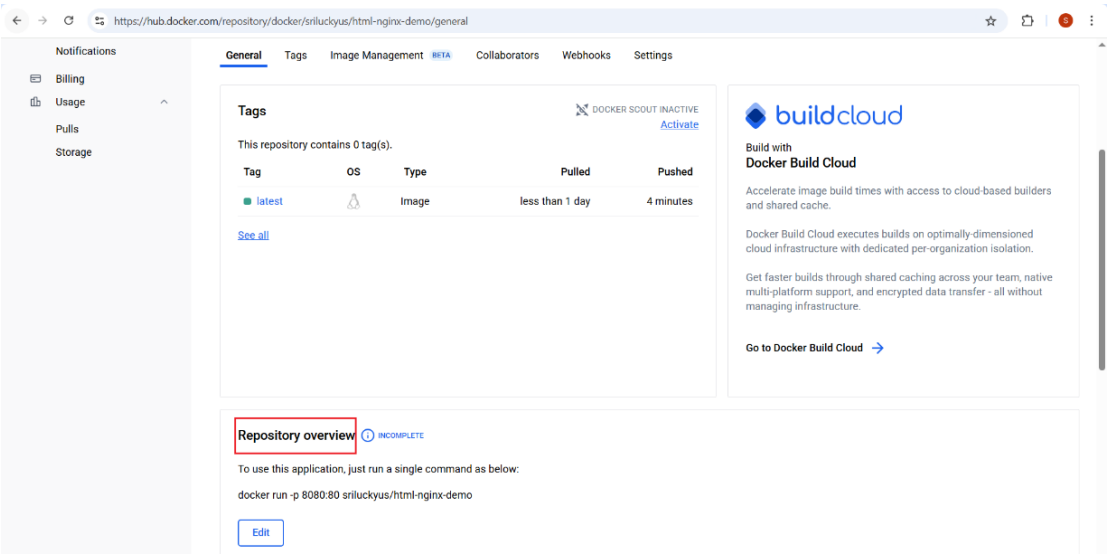
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/HtmlNginx
$ docker login -u "sriluckyus" docker.io

i Info → A Personal Access Token (PAT) can be used instead.
         To create a PAT, visit https://app.docker.com/settings


Password:
Login Succeeded

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/HtmlNginx
$

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/HtmlNginx
$ docker push sriluckyus/html-nginx-demo
Using default tag: latest
The push refers to repository [docker.io/sriluckyus/html-nginx-demo]
9c70ff36425e: Pushed
0d853d50b128: Mounted from library/nginx
947e805a4ac7: Mounted from library/nginx
811a4dbbf4a5: Mounted from library/nginx
b8d7d1d22634: Mounted from library/nginx
e244aa659f61: Mounted from library/nginx
c56f134d3805: Mounted from library/nginx
d71eae0084c1: Mounted from library/nginx
08000c18d16d: Mounted from library/nginx
latest: digest: sha256:ae41b23f7d5933c8a0e8cc51db18c9820ec5ce1bbba69b63f004e11e51ce4de7 size: 2196

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/HtmlNginx
$
```

In Docker Hub, the image is visible under the new repository created



In the **Repository Overview** section, click on **Add Overview** and write the information about this repository something like below:

```
To use this application, just run a single command as below:

docker run -p 8080:80 sriluckyus/html-nginx-demo
```

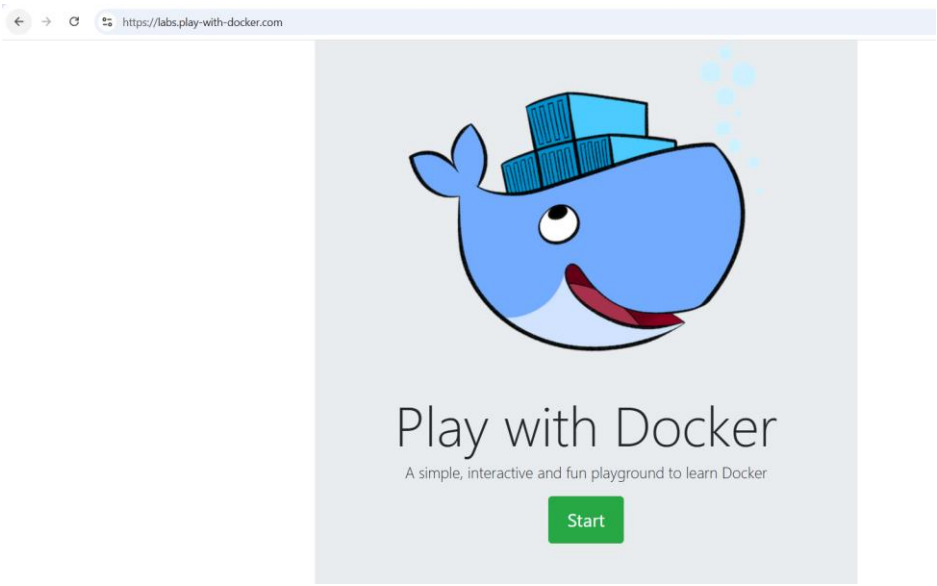


Now, anyone who has access to the above repository can get the docker container running locally.

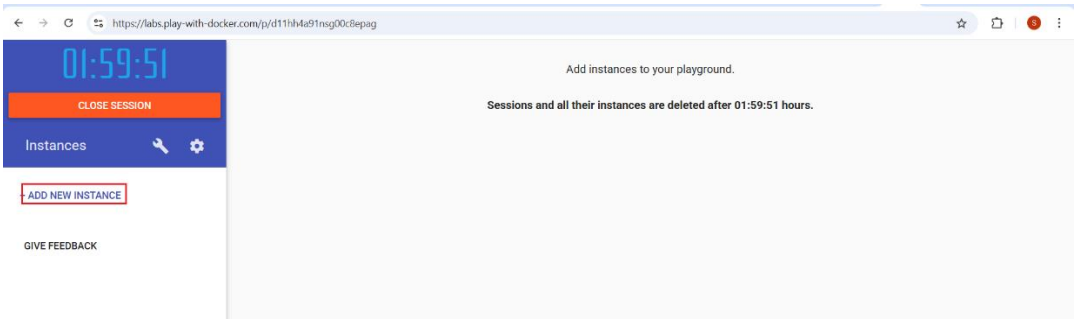
7.4. Run User Repo Docker Image:

Let us try to run the above Docker image in **Docker Playground** cloud platform.

Login to Docker Playground https://labs.play-with-docker.com/ and click on **Start** button.

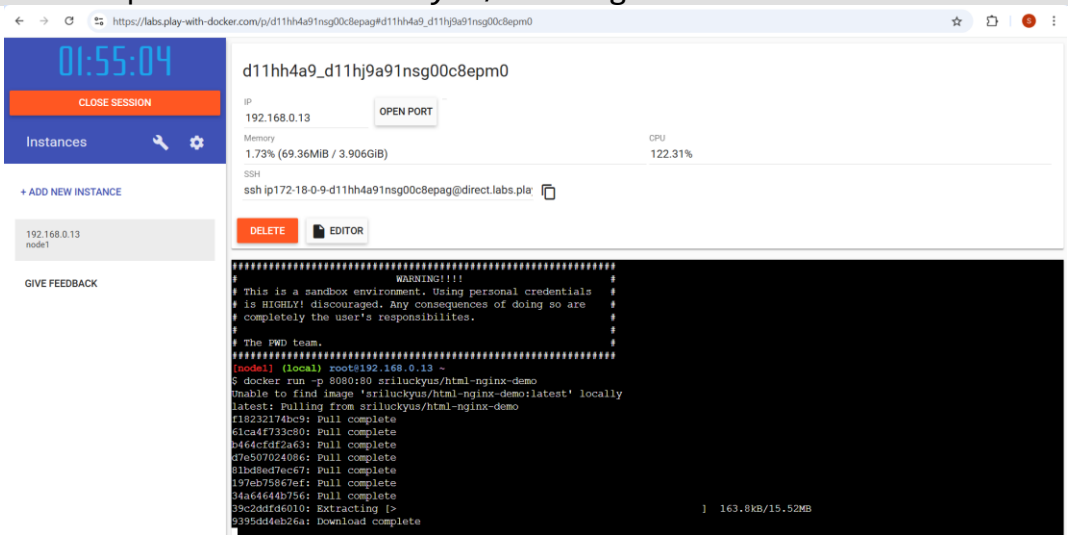


Click on **ADD NEW INSTANCE** to create a new Docker instance

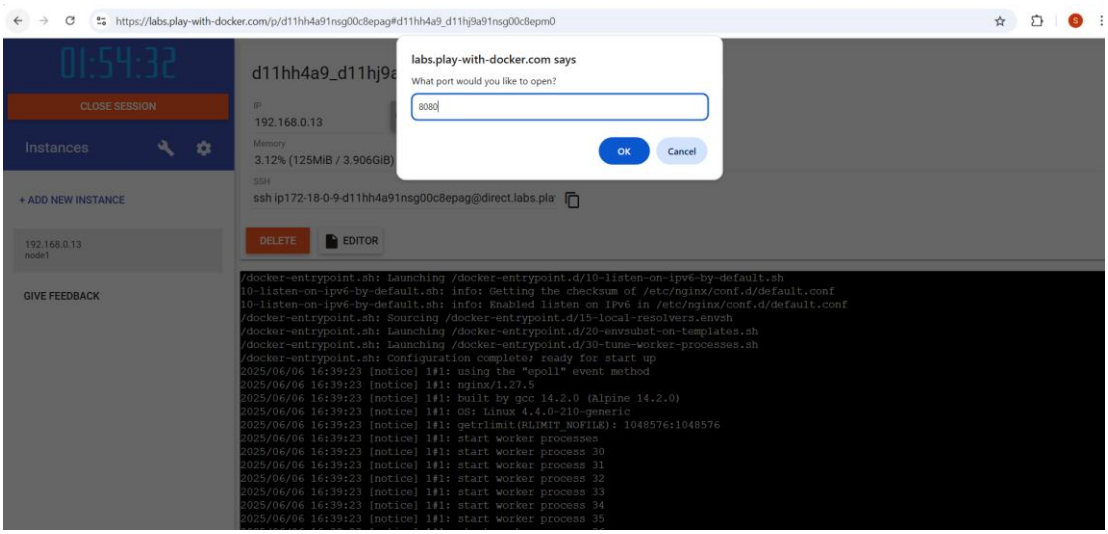


Once the Docker instance is created, run the `html-nginx-demo` image available in `sriluckyus` Docker Hub user by executing the below command in the terminal:

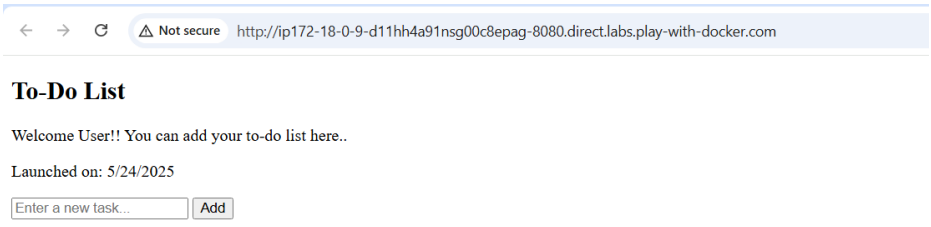
```
docker run -p 8080:80 sriluckyus/html-nginx-demo
```



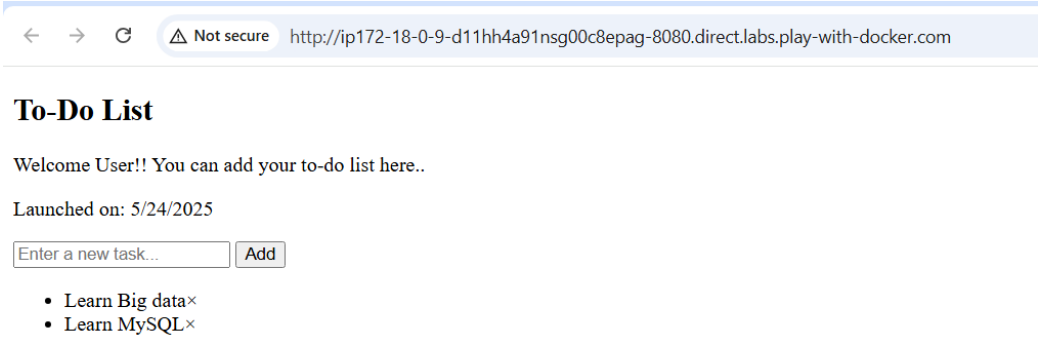
Since our web application is listening on port `8080`, click on **OPEN PORT** button and enter `8080`.



This navigates us to the web application that we built as shown below



This web application is working as expected where we can add or delete tasks.



8. Docker Compose:

Docker Compose is a tool developed by Docker to manage multi-container Docker applications running on the single host. It simplifies the deployment of complex applications that are composed of multiple interconnected containers.

For example, to insert data using a simple Python web application, 3 containers might be needed – first to run Nginx server, second to run Python application with flask framework and third to run the backend database such as MySQL. Though these containers can be initiated individually, setting up internal network for containers to connect with each other is difficult but become easy with Docker Compose tool which helps to bring up all containers at once. When a container goes down, Docker Compose only recreates containers that have changed.

With Docker compose, all configurations such as services, networks and volumes that are needed to setup and run the application are stored in a single configuration file. Then, all services with inter-connected containers are created and started with a single command from the configuration file.

The Docker Compose configuration file has naming convention `compose.yaml` (preferred) or `compose.yml` or `docker-compose.yaml` or `docker-compose.yml` which uses YAML scripting language. It is important to understand the Docker Compose file structure as referred here and Docker Compose commands referred here.

8.1. Create Web Application:

Let us design a simple web application to capture student marks data with the help of Python Flask framework and load into MySQL database.

For this application, three containers are needed which will be managed using Docker Compose:

- A container to run **Nginx** server for serving web **HTML** code and acting as a reverse proxy to forward the HTML request to Python Flask
- A container to run **Python Flask** application
- A container to run **MySQL** database

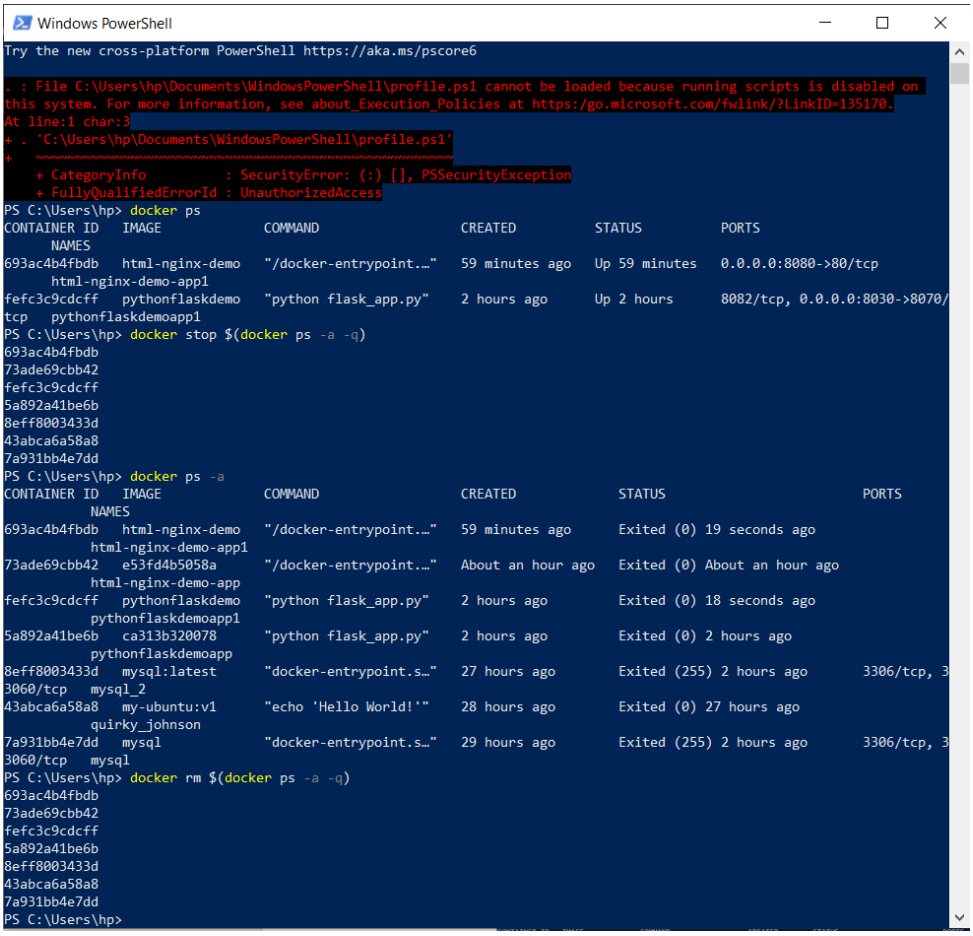
8.2. Stop and Delete Containers:

Before using Docker Compose, first stop all running containers, remove all containers and delete all available images along with any saved volumes just to make sure that everything is started freshly with Docker Compose *(though this is not mandatory)*.

Open a new **Command Prompt** or **Windows PowerShell** and run the following commands:

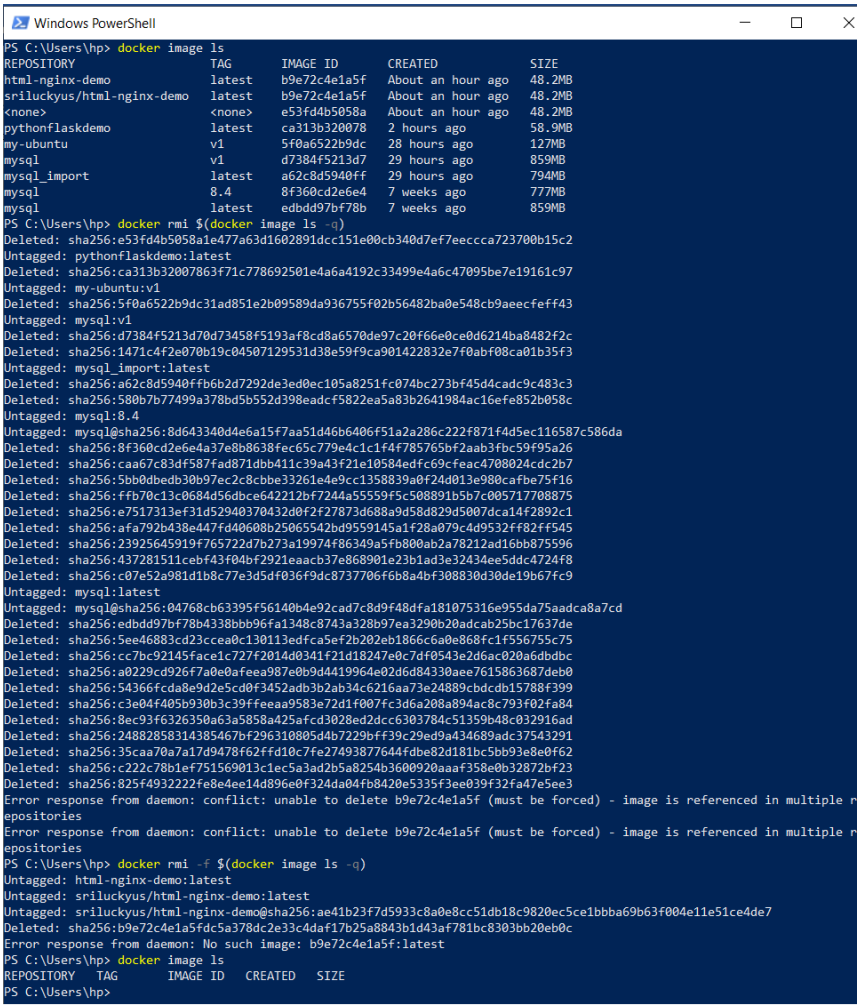
Stop and delete all containers:

```
docker ps
docker stop $(docker ps -q)
docker ps -a
docker rm $(docker ps -a -q)
```



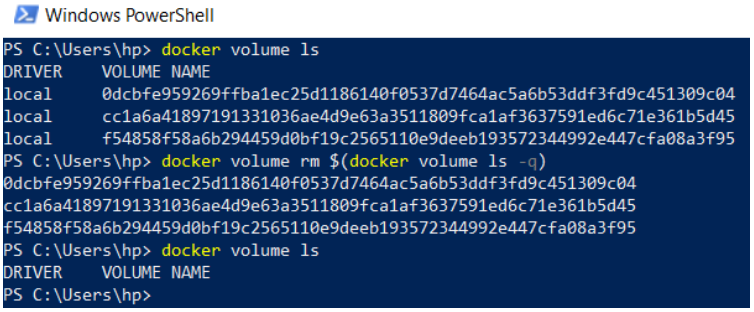
Delete all images:

```
docker image ls
docker rmi $(docker image ls -q)
docker rmi -f $(docker image ls -q)
docker image ls
```



Delete all volumes:

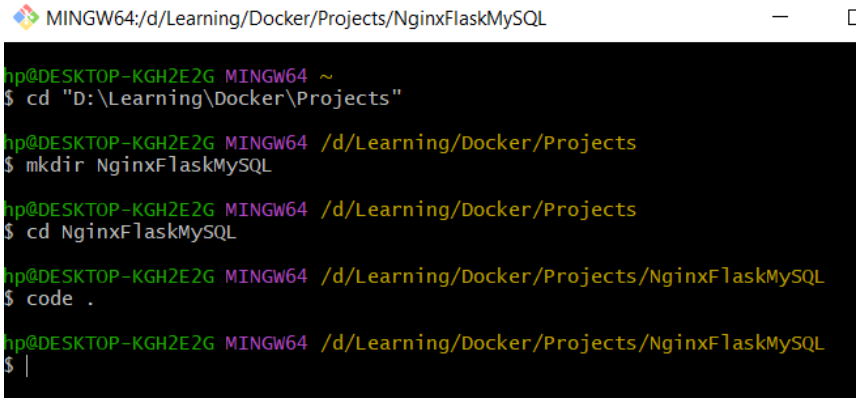
```
docker volume ls
docker volume rm $(docker volume ls -q)
docker volume ls
```



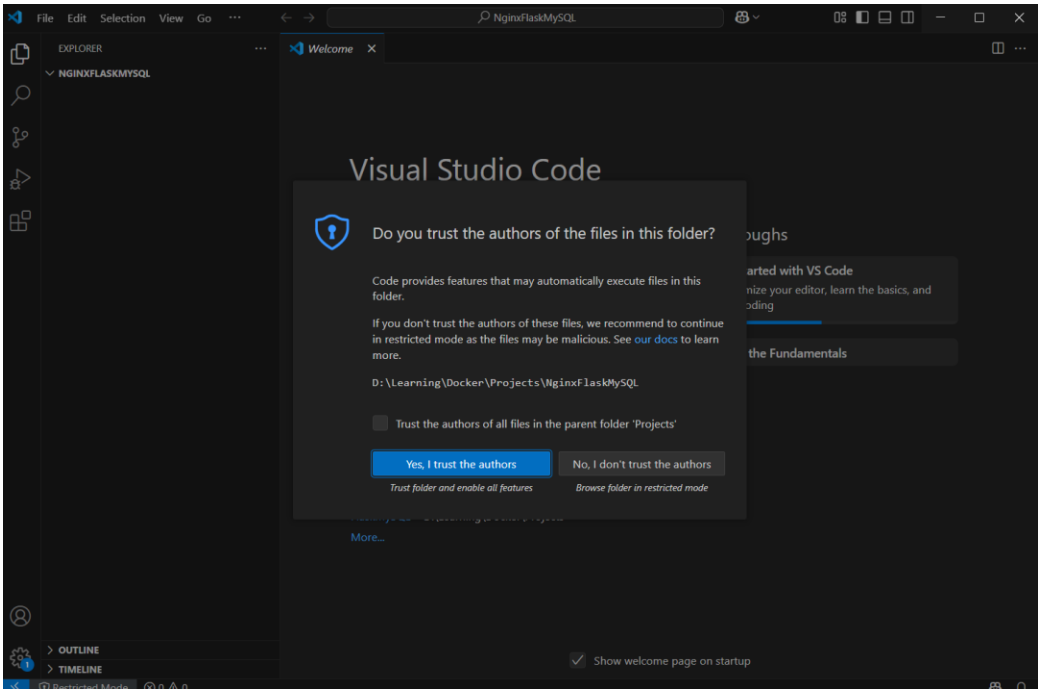
8.3. Launch VS Code:

Open **Git Bash** command prompt, run the following commands to create a new directory at `D:\Learning\Docker\Projects` location and open **VS Code**. You can choose any location to create a new directory.

```
cd "D:\Learning\Docker\Projects"
mkdir NginxFlaskMySQL
cd NginxFlaskMySQL
code .
```



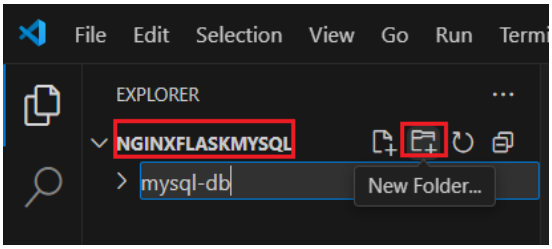
It opens **VS Code** application which might prompt you to confirm if you trust the authors of file in which case, click on **Yes, I trust the authors** button.



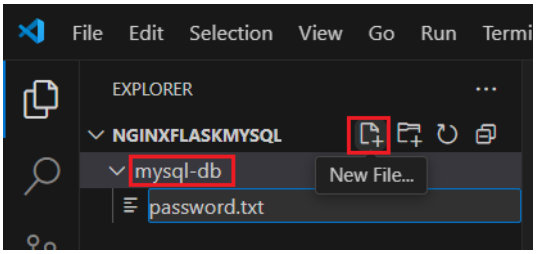
8.4. Configure MySQL Database:

First, we will configure MySQL database which expects a password for `root` user. Here, we are going to store the password in a file which will be passed to containers through Docker `secret` configuration.

In **VS Code** application, click on **New Folder** icon next to `NGINXFLASKMYSQL` project under **EXPLORER** and name the folder as `mysql-db`

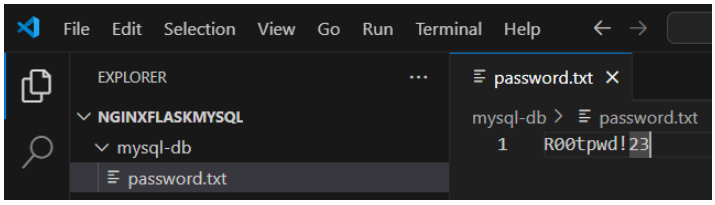


Select the `mysql-db` folder and click on **New File** icon and name it as `password.txt`



Enter the following in `password.txt` file without any extra line and save it *(you can provide any password here but do not enter any new line)*

```
R00tpwd!23
```

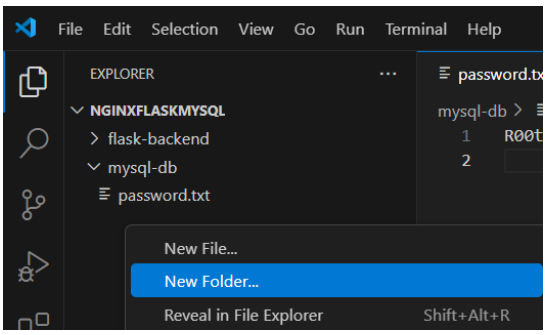


8.5. Configure Python Flask App:

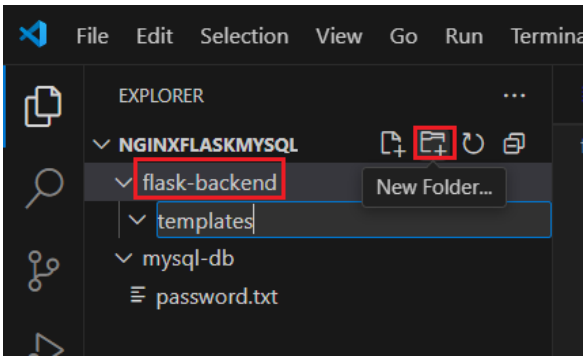
Now, we will create a Python Flask application to render a simple Student form web page and connect to MySQL to insert the submitted data through the web page and display the latest 5 records inserted in the database.

8.5.1. Create HTML File:

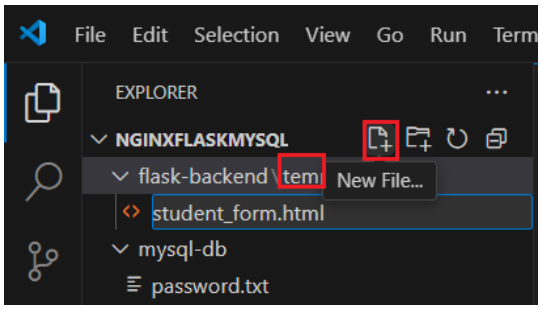
Under `NGINXFLASKMYSQL` project, right click on the empty space and select **New Folder** and name the folder as `flask-backend`



Select the `flask-backend` folder and click on **New Folder** icon and name it as `templates`
(Note that `flask` library expects the HTML file to be available under `templates` folder in order to render the web page)



Select the `templates` folder and click on **New File** icon and name it as `student_form.html`



Enter the following code in `student_form.html` file and save it:

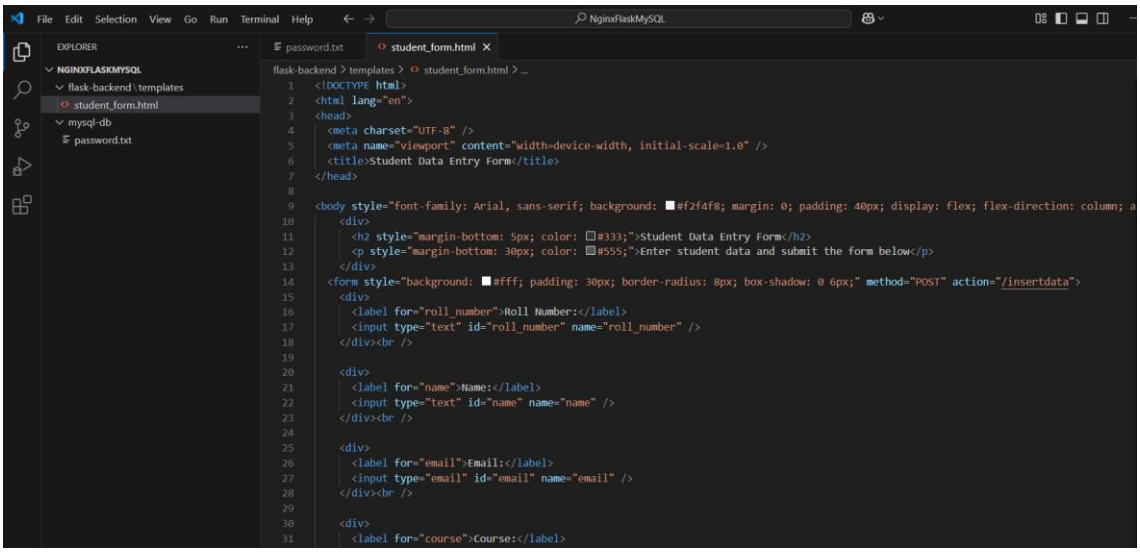
```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Student Data Entry Form</title>
</head>

<body style="font-family: Arial, sans-serif; background: #f2f4f8; margin: 0;
padding: 40px; display: flex; flex-direction: column; align-items: center;">
      <div>
        <h2 style="margin-bottom: 5px; color: #333;">Student Data Entry
Form</h2>
        <p style="margin-bottom: 30px; color: #555;">Enter student data and
submit the form below</p>
      </div>
  <form style="background: #fff; padding: 30px; border-radius: 8px; box-
shadow: 0 6px;" method="POST" action="/insertdata">
    <div>
      <label for="roll_number">Roll Number:</label>
      <input type="text" id="roll_number" name="roll_number" />
    </div><br />

    <div>
      <label for="name">Name:</label>
      <input type="text" id="name" name="name" />
    </div><br />

    <div>
      <label for="email">Email:</label>
      <input type="email" id="email" name="email" />
    </div><br />

    <div>
      <label for="course">Course:</label>
      <select id="course" name="course">
        <option value="btech">B.Tech</option>
        <option value="bca">B.C.A</option>
        <option value="mba">M.B.A</option>
        <option value="bsc">B.Sc</option>
      </select>
    </div><br />

    <div>
      <label for="marks">Marks:</label>
      <input type="text" id="marks" name="marks" />
    </div><br />

    <input type="submit" value="Submit" />
  </form>
</body>
</html>
```

The above code performs the following:

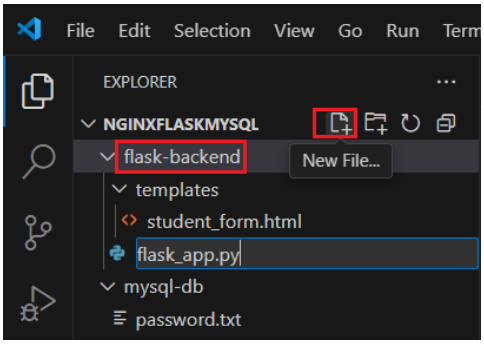
- Build a webpage with a title **Student Data Entry Form.**
- Display a form with 5 fields named **Roll Number**, **Name**, **Email**, **Course**, **Marks** and a **Submit** button.

- Configure the form as `POST` method and call `/insertdata` page up on clicking **Submit** button.



8.5.2. Create Python Files:

Now, select `flask-backend` folder under `NGINXFLASKMYSQL` project and click on **New File** icon and name it as `flask-app.py`



Enter the following code in `flask_app.py` file and save it:

```
from flask import Flask, render_template, request, redirect
from db_operations import DBManager

app=Flask(__name__)

@app.route("/studentform")
def loadForm():
    return render_template('student_form.html')

@app.route("/insertdata", methods=['GET','POST'])
def insertData():
    if request.method == 'POST':
```

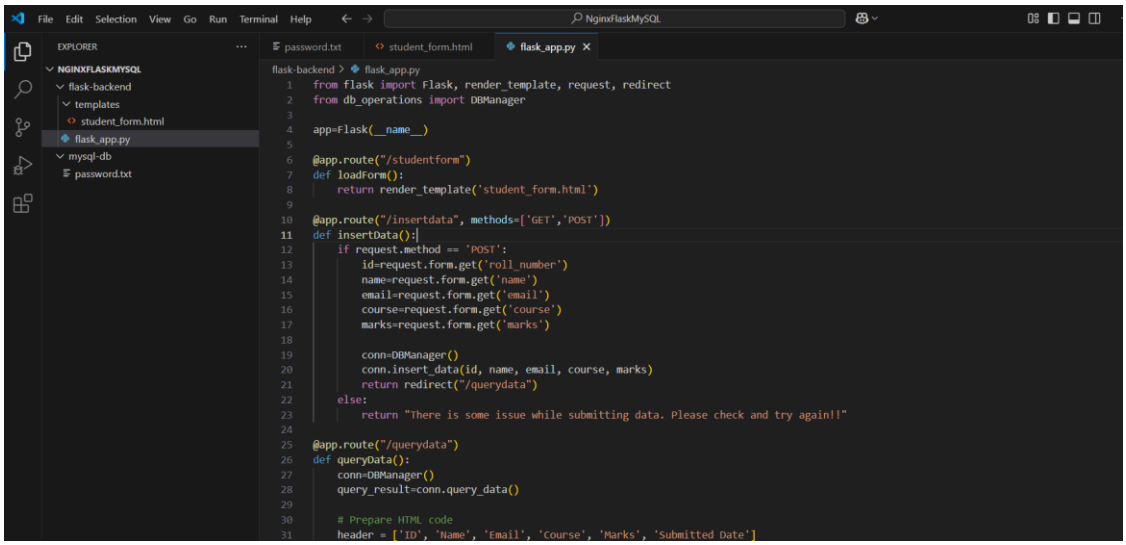
```
        id=request.form.get('roll_number')
        name=request.form.get('name')
        email=request.form.get('email')
        course=request.form.get('course')
        marks=request.form.get('marks')

        conn=DBManager()
        conn.insert_data(id, name, email, course, marks)
        return redirect("/querydata")
    else:
        return "There is some issue while submitting data. Please check and
try again!!"

@app.route("/querydata")
def queryData():
    conn=DBManager()
    query_result=conn.query_data()

    # Prepare HTML code
    header = ['ID', 'Name', 'Email', 'Course', 'Marks', 'Submitted Date']
    table_header = '<tr>' + ''.join(f'<th style="border: 1px solid #ccc;
background: #ddd;">{item}</th>' for item in header) + '</tr>'
    table_rows=''
    for row in query_result:
        table_rows = table_rows + '<tr>' + ''.join(f'<td style="border: 1px
solid #ccc;">{item}</td>' for item in row) + '</tr>'

    response_html = f'''
        <body style="font-family: Arial, sans-serif; text-align: center;">
            <h3 style="color: green;">Data has been submitted
successfully!!</h3>
            <h4>Click <a href="./studentform" style="color: blue">here</a>
to submit a new record</h4>
            <h4>For your reference, 5 most recently submitted records are
listed below:</h4>
            <table style="margin: auto; max-width: 500px;">
                {table_header}
                {table_rows}
            </table>
        </body>
    '''
    return response_html

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=8070)
```

The above code performs the following:

- Import `Flask`, `render_template`, `request`, `redirect` methods from `flask` library.
- Import `DBManager` class from `db_operations` module *(we need to create this module later)*
- Create three flask application routes:

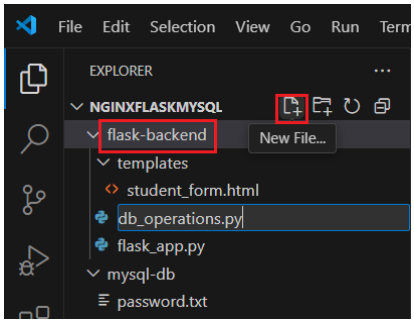
- `/studentform` route with a Python function that renders `student_form.html` file *(available in `templates` folder)*.
- `/insertdata` route with a Python function that verifies the method. If the method is `POST`, it reads values stored in form ids named `roll_number`, `name`, `email`, `course` and `marks` and creates `DBManager` object to call `insert_data` method with values read from the form and then redirects to `/querydata` page.
- `/querydata` route with a Python function that creates `DBManager` object to call `query_data` method. Then, it prepares a HTML string to display a message along with a link to submit new record and display a table with last 5 records submitted and finally returns the HTML string.

- Run the overall flask application on all IP addresses (`0.0.0.0`) and port `8070` with debug enabled.



Now, let us create `db_operations.py` file which is being referenced in our `flask_app.py` file.

Select the `flask-backend` folder and click on **New File** icon and name it as `db_operations.py`



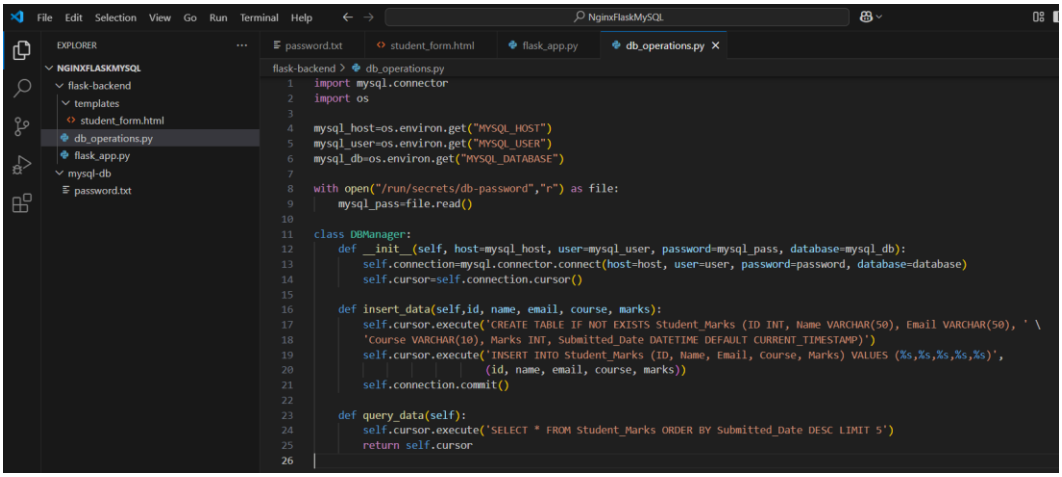
Enter the following code in `db_operations.py` file and save it:

```
import mysql.connector
import os

mysql_host=os.environ.get("MYSQL_HOST")
mysql_user=os.environ.get("MYSQL_USER")
mysql_db=os.environ.get("MYSQL_DATABASE")

with open("/run/secrets/db-password","r") as file:
    mysql_pass=file.read()

class DBManager:
    def __init__(self, host=mysql_host, user=mysql_user,
password=mysql_pass, database=mysql_db):
        self.connection=mysql.connector.connect(host=host, user=user,
password=password, database=database)
        self.cursor=self.connection.cursor()

    def insert_data(self,id, name, email, course, marks):
        self.cursor.execute('CREATE TABLE IF NOT EXISTS Student_Marks (ID
INT, Name VARCHAR(50), Email VARCHAR(50), ' \
        'Course VARCHAR(10), Marks INT, Submitted_Date DATETIME DEFAULT
CURRENT_TIMESTAMP)')
        self.cursor.execute('INSERT INTO Student_Marks (ID, Name, Email,
Course, Marks) VALUES (%s,%s,%s,%s,%s)',
                            (id, name, email, course, marks))
        self.connection.commit()

    def query_data(self):
        self.cursor.execute('SELECT * FROM Student_Marks ORDER BY
Submitted_Date DESC LIMIT 5')
        return self.cursor
```

The above code performs the following:

- Import `mysql.connector` library.
- Get MySQL host, user and database details from environment variables.
- Get database password from `/run/secrets/db-password` file *(this file will be created inside the container when it is built)*.
- Define `DBManager` class:

- Create initialization method to call `connect` method of `mysql.connector` library by passing `host`, `user`, `password` and `database` variables.
- Create `insert_data` method to execute `CREATE TABLE` query that creates `Student_Marks` table with `ID`, `Email`, `Course`, `Marks`, `Submitted_Date` (defaults to current time stamp) columns.
- Create `query_data` method to execute `SELECT` query that fetches last 5 records from `Student_Marks` table ordered by `Submitted_Date` column in descending order.



8.5.3. Create Requirement File:

Again, select the `flask-backend` folder under `NGINXFLASKMYSQL` project and click on **New File** icon and name it as `requirements.txt` in which write the following lines and save the file. This file is used to install Python libraries specified in the file.

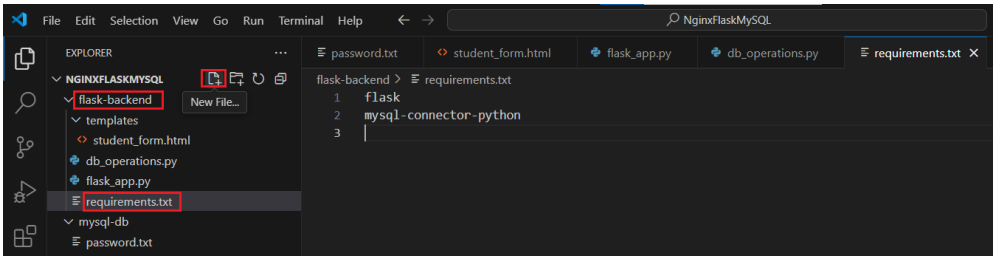
```
flask
mysql-connector-python
```

The above code performs the following:

- Use `flask` library
- Use `mysql-connector-python` library which is mandatary for Flask to be able to connect to MySQL database.

  Note: There are other libraries such as `mysql-connector`, `pymysql` also available to connect to MySQL database from Python but these libraries might result an error **Authentication plugin 'caching_sha2_password' is not supported** when the latest version of MySQL software is used. Hence `mysql-connector-python` library *(developed by MySQL Developer Zone and now maintained by **Oracle**)* is recommended which is compatible with the latest version of MySQL which uses

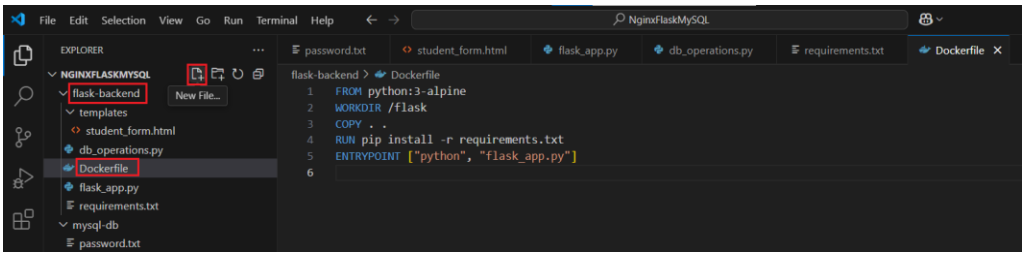
caching_sha2_password plugin from MySQL 8.0 version. If `mysql-connector`, `pymysql` libraries are needed, use either **MySQL 8.0** version image or set `auth_plugin` to `mysql_native_password` in the latest version of MySQL image. For more details, refer [this blog].



8.5.4. Create Docker File:

Now, let us create a Docker file for Python Flask application. Select `flask-backend` folder under `NGINXFLASKMYSQL` project and click on **New File** icon and name it as `Dockerfile`. Then, enter the following instructions in `Dockerfile` file and save it.

```
FROM python:3-alpine
WORKDIR /flask
COPY . .
RUN pip install -r requirements.txt
ENTRYPOINT ["python", "flask_app.py"]
```



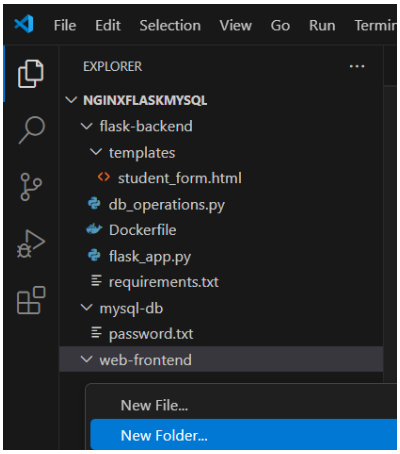
The above code performs the following:

- Pull `python:3-alpine` image if not available in local host.
- Create and set the working directory as `/flask` in container image.
- Copy all files from local host to image working directory.
- Run command to install libraries mentioned in `requirements.txt` file within image.
- Set the entry point command to run Python `flask_app.py` program when container is started.

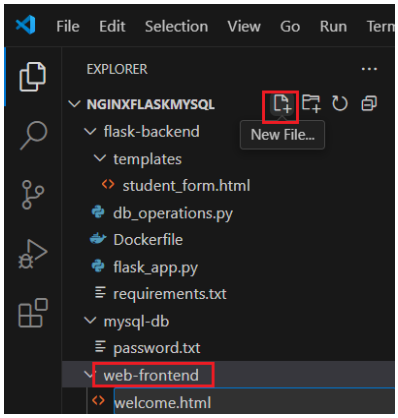
8.6. Configure Web App:

Now, we will create a simple `welcome` web page that calls `student_form` page that was created earlier.

Under `NGINXFLASKMYSQL` project, right click on the empty space and select **New Folder** and name the folder as `web-frontend`



Select the `web-frontend` folder and click on **New File** icon and name it as `welcome.html`



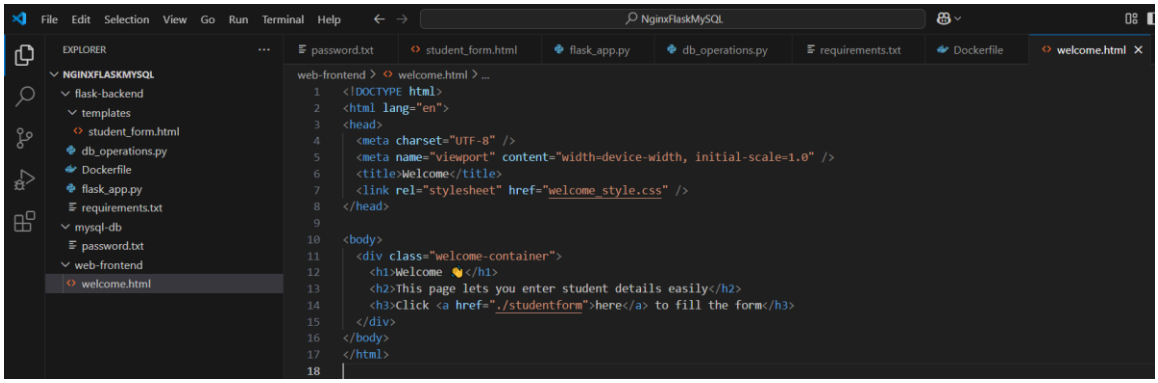
Enter the following code in `welcome.html` file and save it.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Welcome</title>
  <link rel="stylesheet" href="welcome_style.css" />
</head>

<body>
```

```
  <div class="welcome-container">
    <h1>Welcome 👏</h1>
    <h2>This page lets you enter student details easily</h2>
    <h3>Click <a href="./studentform">here</a> to fill the form</h3>
  </div>
</body>
</html>
```

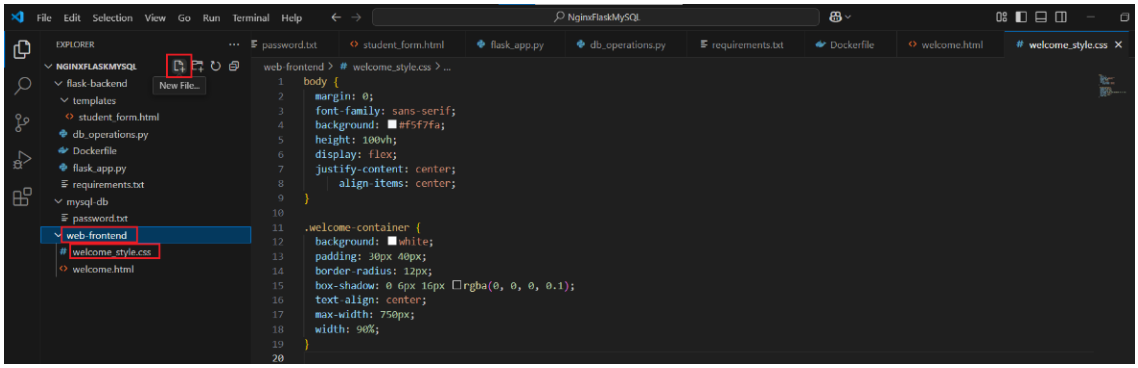
The above code performs the following:

- Build a webpage with a title **Welcome**.
- Display a message to enter student details.
- Display a web link that calls `/studentform` page at background.



Now, let us create a styling sheet which has been referenced in the above HTML code. Select the `web-frontend` folder and click on **New File** icon and name it as `welcome_style.css` in which enter the following code and save it.

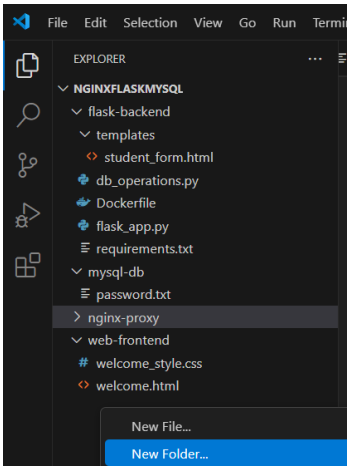
```
body {
  margin: 0;
  font-family: sans-serif;
  background: #f5f7fa;
  height: 100vh;
  display: flex;
  justify-content: center;
        align-items: center;
}

.welcome-container {
  background: white;
  padding: 30px 40px;
  border-radius: 12px;
  box-shadow: 0 6px 16px rgba(0, 0, 0, 0.1);
  text-align: center;
  max-width: 750px;
  width: 90%;
}
```



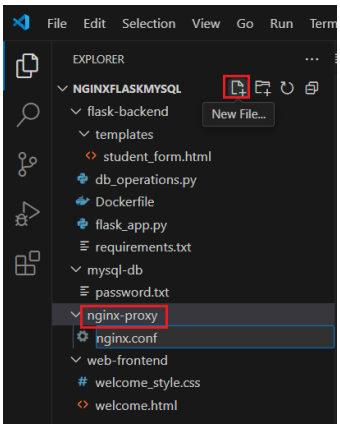
8.7. Configure Nginx Server:

Now, we will use **Nginx** server to act as a reverse proxy to forward the web app request to Python Flask app.

Under `NGINXFLASKMYSQL` project, right click on the empty space and select **New Folder** and name the folder as `nginx-proxy`



Select the `nginx-proxy` folder and click on **New File** icon and name it as `nginx.conf`

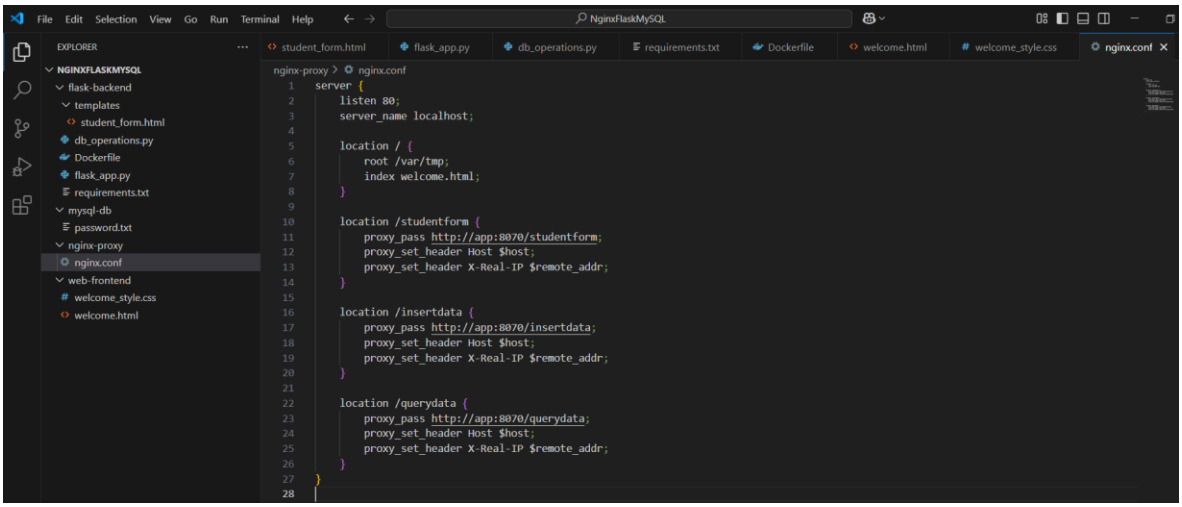


Enter the following code in `nginx.conf` file and save it.

```
server {
    listen 80;
    server_name localhost;

    location / {
        root /var/tmp;
        index welcome.html;
    }

    location /studentform {
        proxy_pass http://app:8070/studentform;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    location /insertdata {
        proxy_pass http://app:8070/insertdata;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    location /querydata {
        proxy_pass http://app:8070/querydata;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

The above code performs the following:

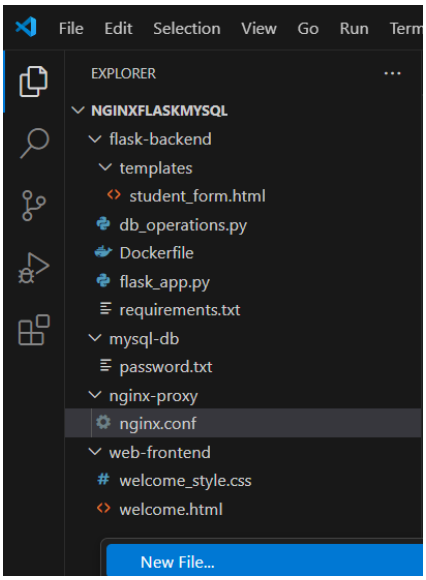
- Set the `server` to listen on port `80`.
- Set the `server_name` as `localhost`.
- Create 4 web locations:
    - `/` location that sets `root` directory as `/var/tmp` and calls `index` page `welcome.html`.
    - `/studentform` location that forwards request to http://app:8070/studentform webpage *(here `app` is the flask container name that will be created later and `8070` is the port where our flask application is configured to run)*
    - `/insertdata` location that forwards request to http://app:8070/insertdata webpage.
    - `/querydata` location that forwards request to http://app:8070/querydata webpage.



8.8. Create Docker Compose File:

Now that all applications are configured, we will create a Docker Compose file to create respective services for each application container and connect them through custom network.

Under `NGINXFLASKMYSQL` project, right click on empty space and select **New File** icon and name it as `docker-compose.yml`



In the `docker-compose.yml` file, write the following lines of code and save it.

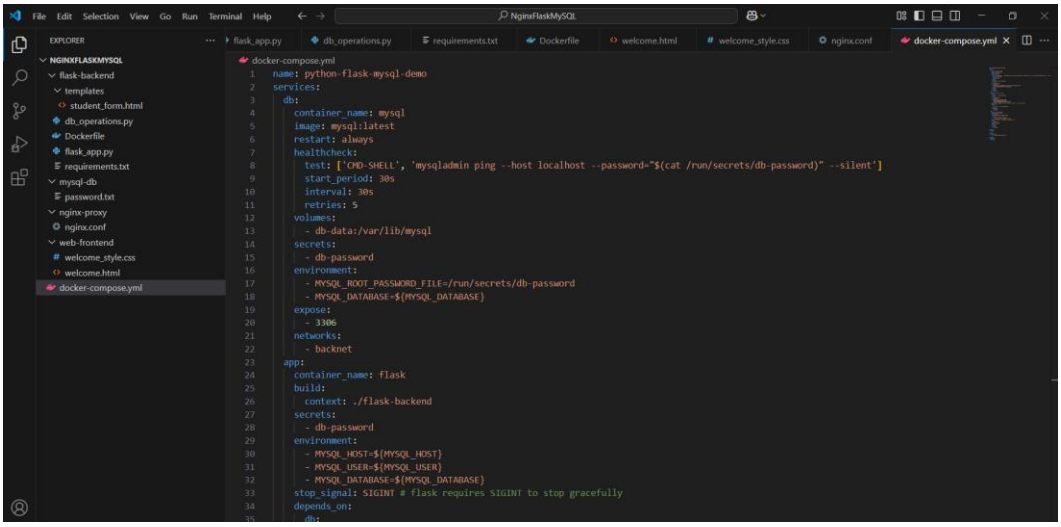
```
name: python-flask-mysql-demo
services:
  db:
    container_name: mysql
    image: mysql:latest
    restart: always
```

```
    healthcheck:
      test: ['CMD-SHELL', 'mysqladmin ping --host localhost --
password="$(cat /run/secrets/db-password)" --silent']
      start_period: 30s
      interval: 30s
      retries: 5
    volumes:
      - db-data:/var/lib/mysql
    secrets:
      - db-password
    environment:
      - MYSQL_ROOT_PASSWORD_FILE=/run/secrets/db-password
      - MYSQL_DATABASE=${MYSQL_DATABASE}
    expose:
      - 3306
    networks:
      - backnet
  app:
    container_name: flask
    build:
      context: ./flask-backend
    secrets:
      - db-password
    environment:
      - MYSQL_HOST=${MYSQL_HOST}
      - MYSQL_USER=${MYSQL_USER}
      - MYSQL_DATABASE=${MYSQL_DATABASE}
    stop_signal: SIGINT # flask requires SIGINT to stop gracefully
    depends_on:
      db:
        condition: service_healthy
    networks:
      - backnet
      - frontnet
  web:
    container_name: nginx
    image: nginx:alpine
    environment:
      - FLASK_SERVER_ADDR=app:8070
    volumes:
      - ./web-frontend:/var/tmp
      - ./nginx-proxy:/etc/nginx/conf.d
    command: /bin/sh -c "nginx -g 'daemon off;'"
    ports:
      - 8080:80
    restart: always
    depends_on:
      - app
    networks:
      - frontnet

volumes:
  db-data:

secrets:
  db-password:
    file: mysql-db/password.txt
```

```
networks:
  backnet:
  frontnet:
```

The above code performs the following:

- Set the Compose stack name as `python-flask-mysql-demo`.
- Create 3 services:
    - `db` service that sets container name as `mysql`, pulls `mysql:latest` image, restarts `always` in the event of failure, performs health check of container with the given `test` command after `30` seconds of container start time and at a regular internal of `30` seconds for a maximum of `5` times, maps `db-data` volume to `/var/lib/mysql` directory inside container, maps the container to `db-password` secret, sets environment variables for MySQL root password file and database, exposes `3306` port and maps the container to `backnet` network.
    - `app` service that sets container name as `flask`, builds image from `Dockerfile` available in `flask-backend` folder in local host, maps the container to `db-password` secret, sets environment variables for MySQL host, user and database, sets `stop_signal` to `SIGINT` for flask application to stop gracefully, sets dependency on `db` service to look for `service_healthy` status and maps the container to `backnet` and `frontnet` networks.
    - `web` service that sets container name as `nginx`, pulls `nginx:alpine` image, sets environment variables for flask server address, maps `./web-frontend` and `./nginx-proxy` directories in local host to `/var/tmp` and `/etc/nginx/conf.d` directories inside container, executes command to run Nginx server foreground, maps `8080` local host port to `80` container port, restarts `always` in the event of failure, sets dependency on `app` service and maps the container to `frontnet` network.
- Create `db-data` volume.
- Create `db-password` secret using `password.txt` file available in `mysql-db` folder in local host.
- Create `backnet` and `frontnet` networks.

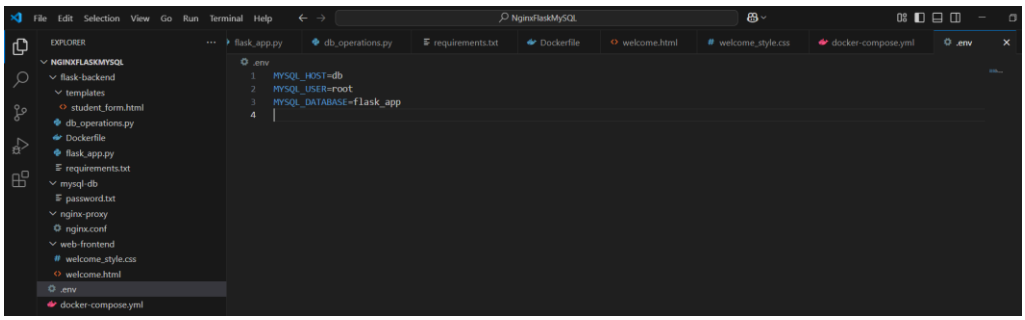


8.9. Create Environment File:

Now, we will create an environment file to set MySQL host, user and database which are being referenced in our `docker-compose.yml` file for Flask application to connect with MySQL.

Under `NGINXFLASKMYSQL` project, right click on empty space and select **New File** icon and name it as `.env` *(Note that there is a* `.` *before* `env`*)* and enter the following lines in `.env` file.

```
MYSQL_HOST=db
MYSQL_USER=root
MYSQL_DATABASE=flask_app
```



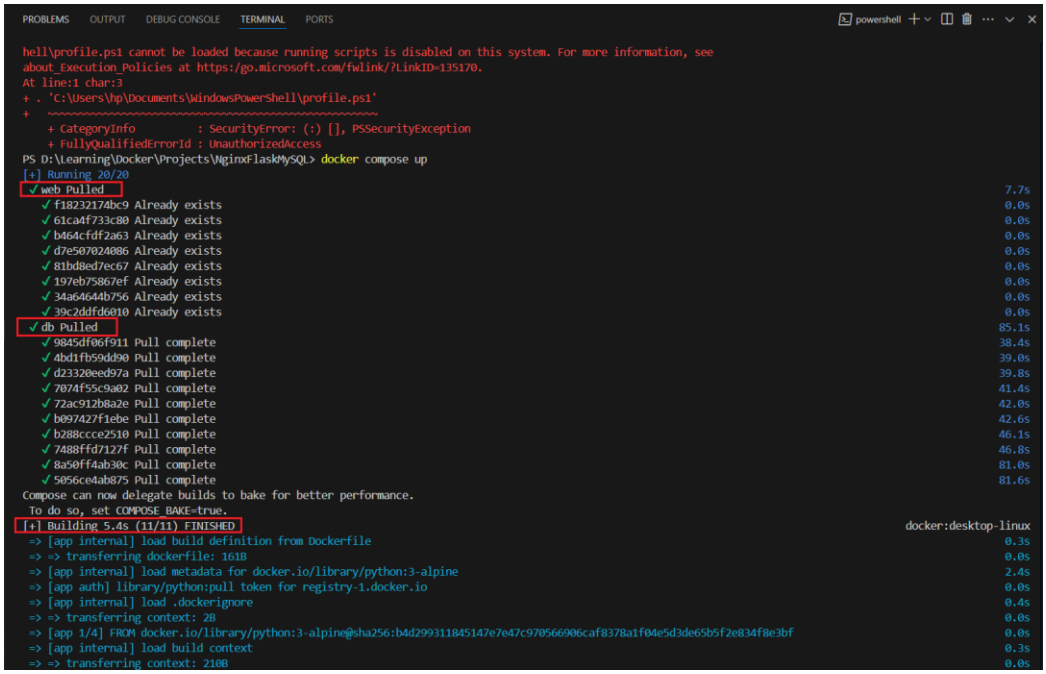
8.10. Start Docker Compose:

Now, let's start the Docker compose to bring up respective Docker containers for our web application.

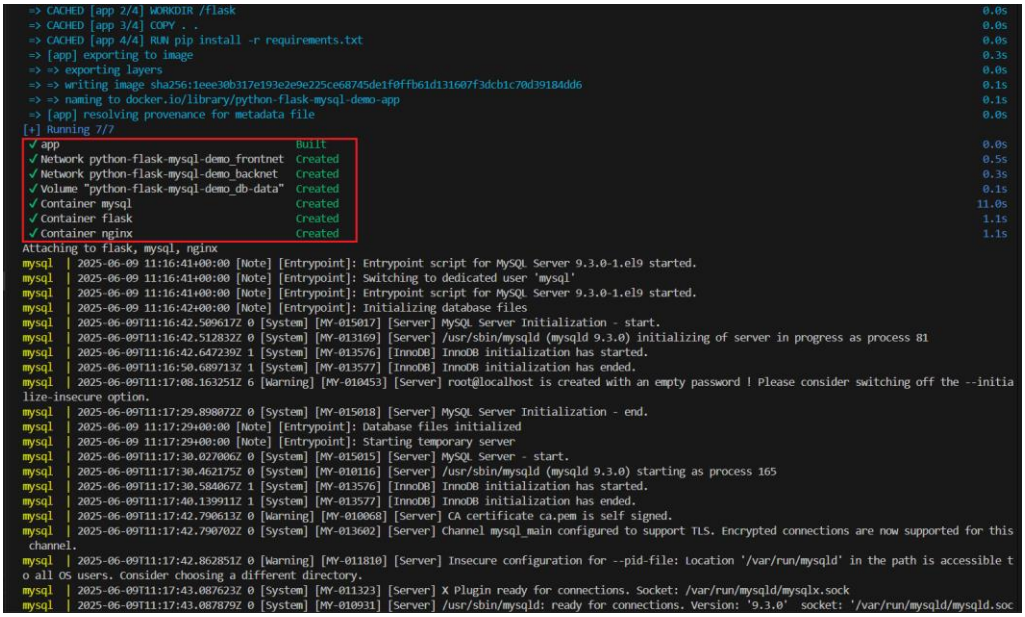
In VS Code, go to **Terminal** menu and select **New Terminal** and run the following command to start the Docker Compose and bring up all containers:

```
docker compose up
```

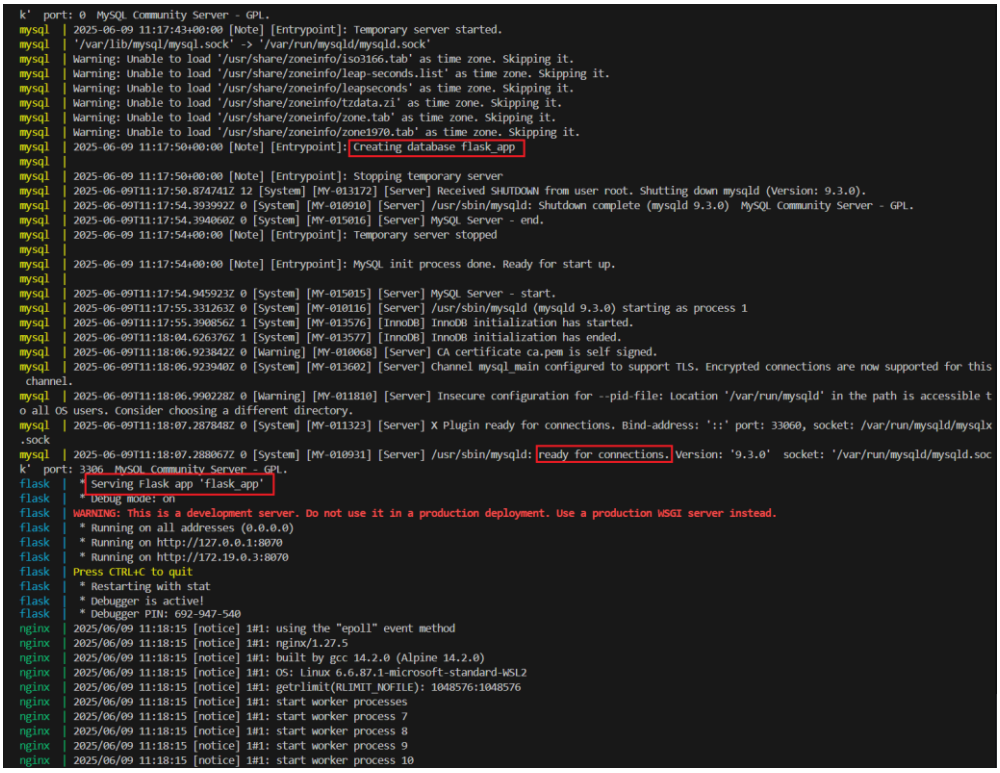
In the output, we can see that docker compose is pulling images for `web` and `db` services while building the image for `flask` service:



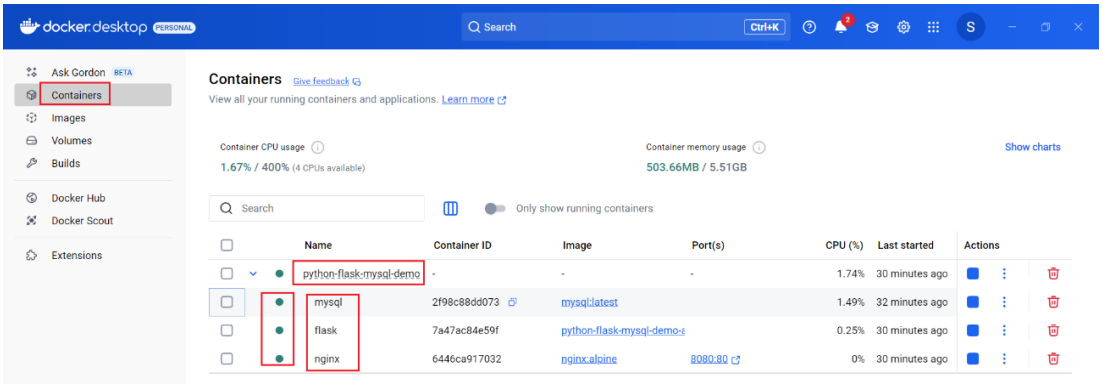
Then, it created 2 networks named `front-net` and `back-net`, volume named `db-data` and containers named `mysql`, `flask` and `nginx`.



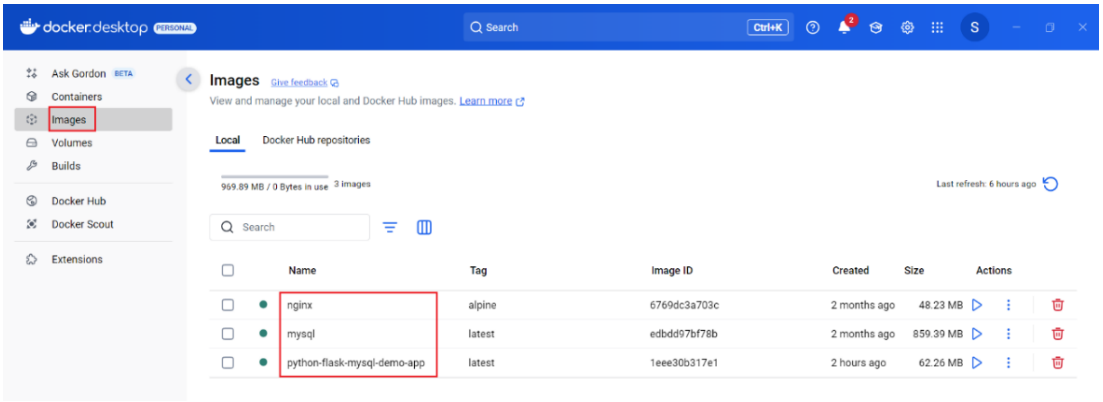
At bottom, we can see that `mysql` container is **creating database flask_app** and making the database **ready for connections**. Then, `flask` container is **serving flask_app** with **running on all addresses** and then brought up the `nginx` container.



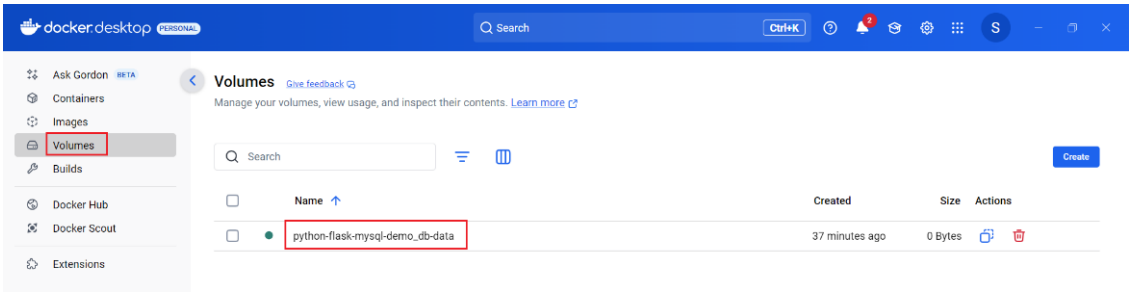
Open **Docker Desktop** and go to **Containers** tab where you can see that `python-flask-mysql-demo` compose stack has been created with three containers – `mysql`, `flask`, `nyginx` which are in **Running** state.



Under **Images** tab, you can see that `nginx` and `mysql` images have been pulled and `python-flask-mysql-demo-app` image has been built.



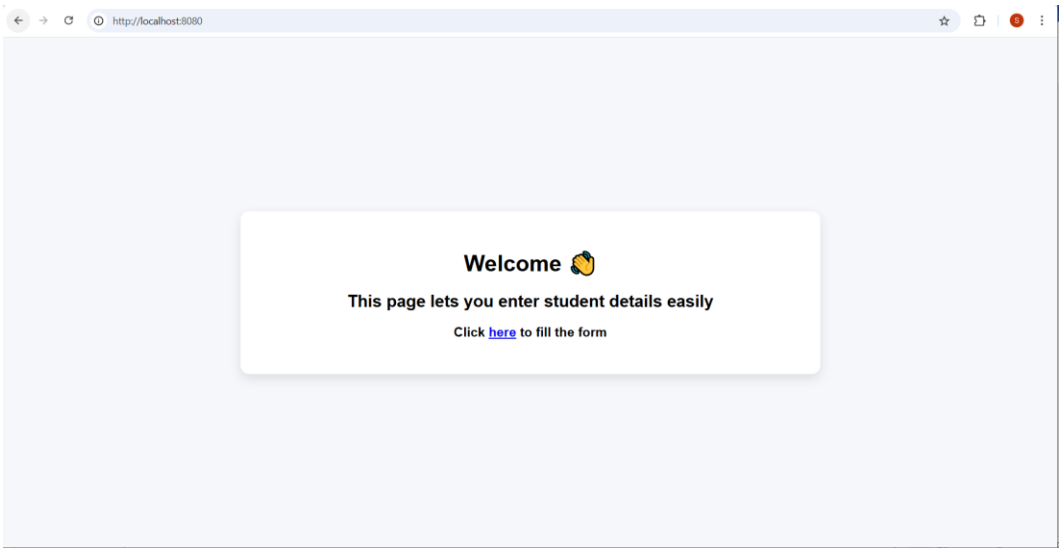
Under **Volumes** tab, you can see that `python-flask-mysql-demo_db-data` volume has been created.



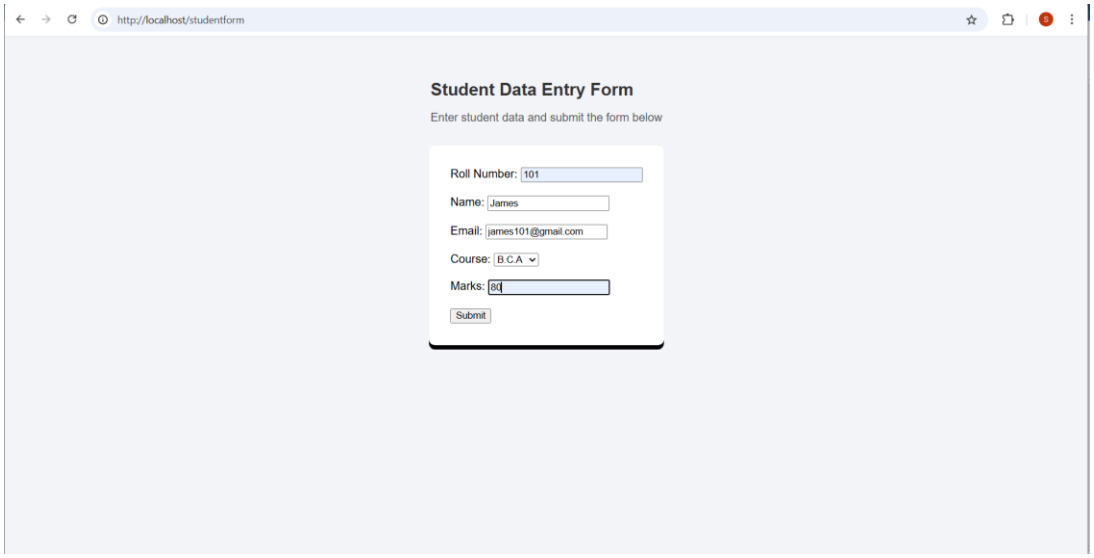
8.11. Launch Web Application:

Now, let's verify if our web application is working as expected.

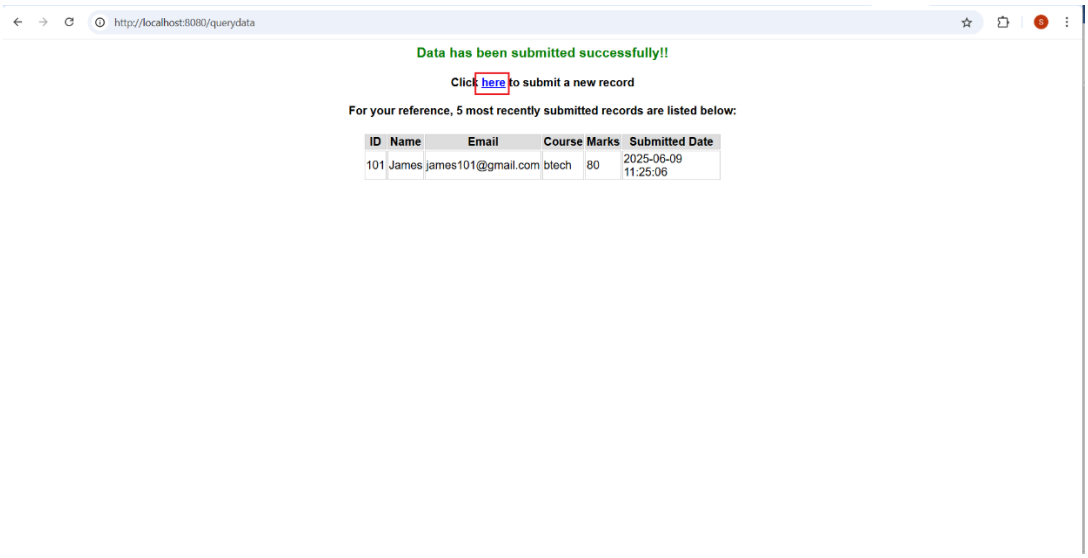
Open web browser and hit http://localhost/ or http://localhost:8080/ URL after which you can see the below welcome page



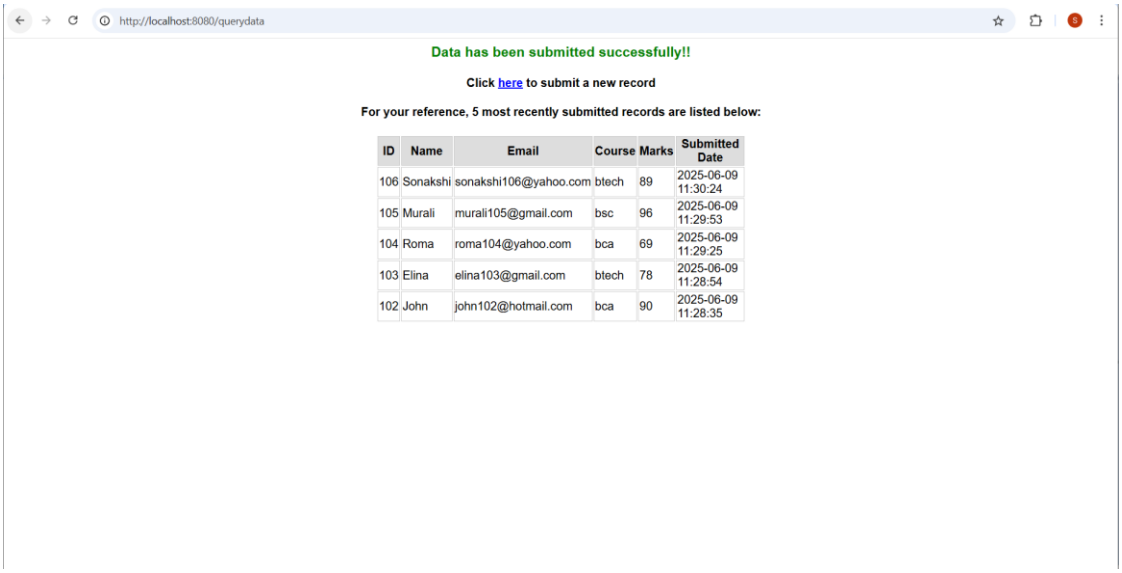
Click on here in the welcome page after which it displays the **Student Data Entry Form** where enter some data and click on **Submit** button.



Once the form is submitted successfully, it displays a message that **Data has been submitted successfully** and displays the submitted record data as shown below:



To enter a new record, click on **here** in the above webpage and submit another 6 to 7 records. After every submission, it displays the last 5 records submitted as shown below:

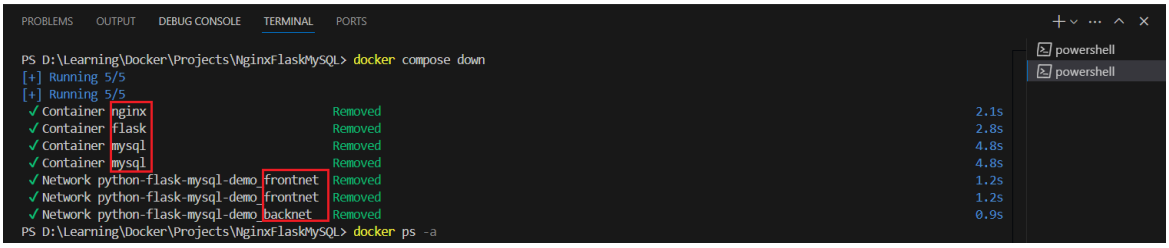


8.12. Stop Docker Compose:

Now, let's stop the Docker compose and verify if the application is down.

Go to **VS Code** and open a **New Terminal** and run the following command to stop the Docker Compose:

```
docker compose down
```



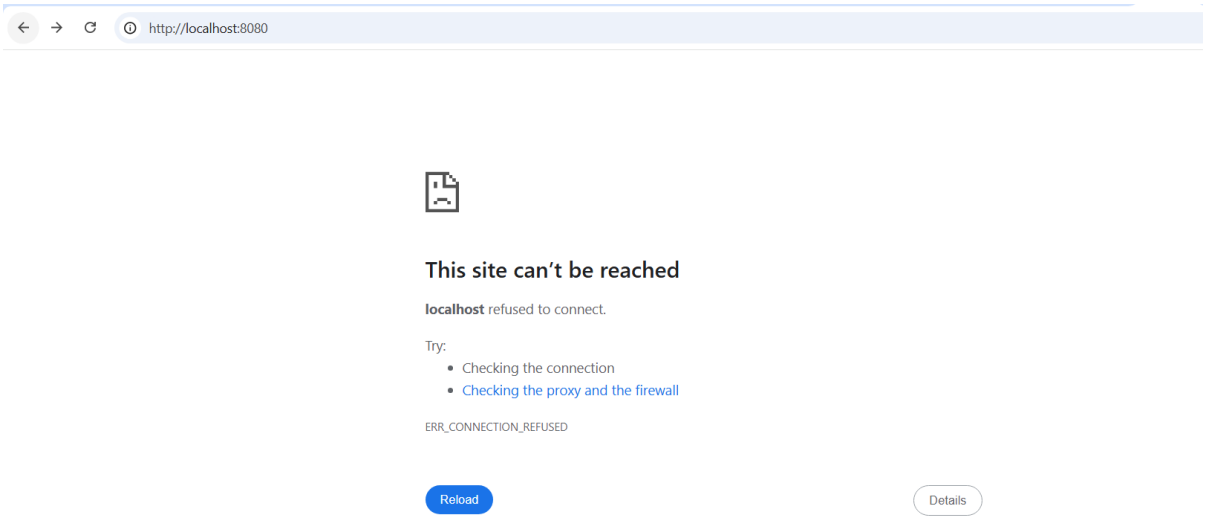
As you can see above, Docker Compose has removed `nginx`, `flask` and `mysql` containers and `frontnet` and `backnet` networks.

Let's verify if containers and network have been removed using the following commands:

```
docker ps -a
docker network ls
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS D:\Learning\Docker\Projects\NginxFlaskMySQL> docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
PS D:\Learning\Docker\Projects\NginxFlaskMySQL>
PS D:\Learning\Docker\Projects\NginxFlaskMySQL> docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
72305091424e   bridge    bridge    local
6bc62a19d6bf   host      host      local
202439de8a5a   none      null      local
PS D:\Learning\Docker\Projects\NginxFlaskMySQL>
```

Refresh the web browser to see that application is down.



8.13. Start Docker Compose Detached:

This time, let us bring up docker compose in detached mode using the following command

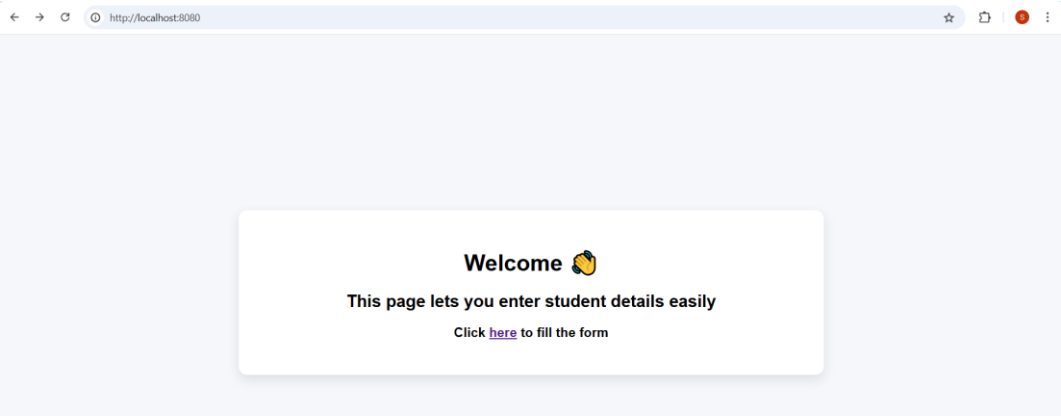
```
docker compose up -d
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS D:\Learning\Docker\Projects\NginxFlaskMySQL> docker compose up -d
[+] Running 5/5
 ✓ Network python-flask-mysql-demo_frontnet  Created                    0.4s
 ✓ Network python-flask-mysql-demo_backnet   Created                    0.3s
 ✓ Container mysql                           Healthy                   26.9s
 ✓ Container flask                           Started                   27.6s
 ✓ Container nginx                           Started                   28.1s
PS D:\Learning\Docker\Projects\NginxFlaskMySQL>
```

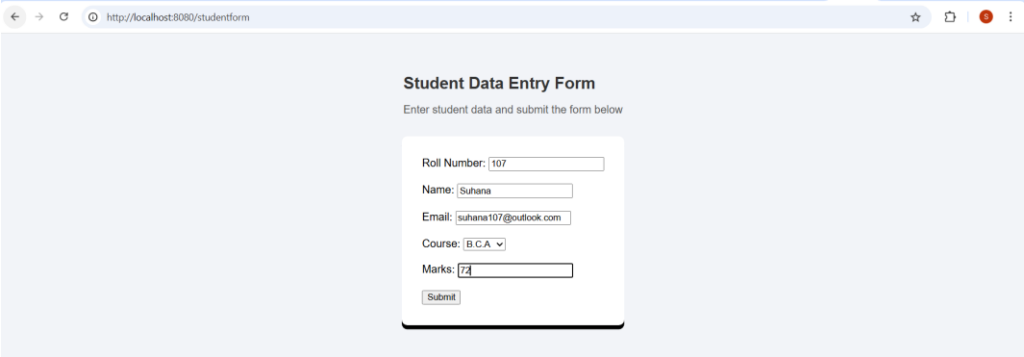
As you see above, the container logs have been suppressed and not being displayed on the console.

8.14. Verify Web Application:

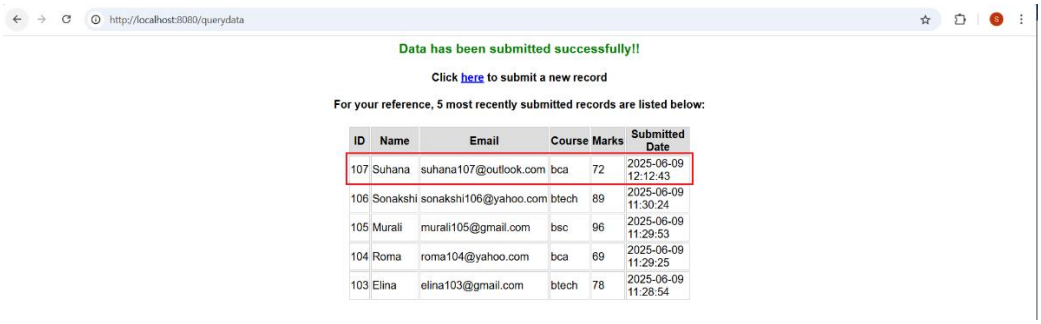
Refresh the web page http://localhost/ or http://localhost:8080/ and we can see that the application is up.



Let us enter a new record and click on **Submit** button.



We can see the latest record submitted along with the previously submitted records:



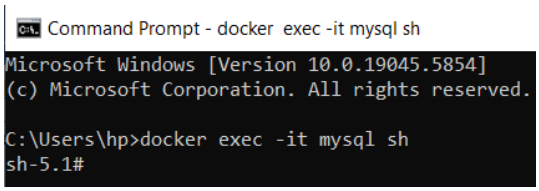
Though the containers have been re-created, it persisted old data since we configured in Docker Compose file to create a volume for data storage.

8.15. Validate Database:

Finally, let's verify if we can connect to `mysql` container and see the data.

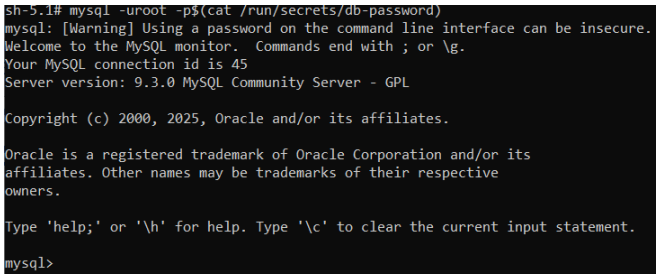
Open **Command Prompt** or **Windows Powershell** and run the following command to connect to `mysql` container in interactive mode:

```
docker exec -it mysql sh
```



On the `sh#` prompt, run the following command to connect to the database. Note that the database password is stored in `/run/secrets/db-password` file which has been created when the `mysql` container was built.

```
mysql -uroot -p$(cat /run/secrets/db-password)
```



On the `mysql>` prompt, run the following queries to verify data in the table:

```
mysql -uroot -p$(cat /run/secrets/db-password)
show databases;
use flask_app;
show tables;
select * from Student_Marks;
```

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| flask_app          |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.197 sec)

mysql> use flask_app;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+--------------------+
| Tables_in_flask_app |
+--------------------+
| Student_Marks      |
+--------------------+
1 row in set (0.003 sec)

mysql> select  * from Student_Marks;
+------+----------+----------------------+--------+-------+---------------------+
| ID   | Name     | Email                | Course | Marks | Submitted_Date      |
+------+----------+----------------------+--------+-------+---------------------+
|  101 | James    | james101@gmail.com   | btech  |    80 | 2025-06-09 11:25:06 |
|  102 | John     | john102@hotmail.com  | bca    |    90 | 2025-06-09 11:28:35 |
|  103 | Elina    | elina103@gmail.com   | btech  |    78 | 2025-06-09 11:28:54 |
|  104 | Roma     | roma104@yahoo.com    | bca    |    69 | 2025-06-09 11:29:25 |
|  105 | Murali   | murali105@gmail.com  | bsc    |    96 | 2025-06-09 11:29:53 |
|  106 | Sonakshi | sonakshi106@yahoo.com| btech  |    89 | 2025-06-09 11:30:24 |
|  107 | Suhana   | suhana107@outlook.com| bca    |    72 | 2025-06-09 12:12:43 |
+------+----------+----------------------+--------+-------+---------------------+
7 rows in set (0.001 sec)

mysql>
```

9. Docker Swarm:

Docker Swarm or **Docker Swarm Mode** is a built-in feature of Docker Engine from version 1.12 that was introduced in June 2016. **Docker Swarm** is a container orchestration tool that allows to manage and deploy Docker applications across multiple Docker hosts as a single virtual system. It consists of a group of physical or virtual machines that run Docker and configured to join as a cluster. The activities of the cluster are managed by the **Swarm Manager**, and machines that have joined the cluster are referred to as **Nodes**. All nodes in Docker Swarm are Docker daemons which interact with each other using Docker API. The cluster management and orchestration features of Docker Swarm embedded in Docker Engine are built using a toolkit called SwarmKit which is a separate Moby project developed by Docker.

The key components of Docker Swarm include **Swarm Manager** which internally consists of **API**, **Orchestrator**, **Allocator**, **Scheduler**, and **Dispatcher**, Swarm worker which internally consists of **Worker**, and **Executor**. The basic Docker Swarm commands can be referred here.

9.1. Setup Swarm using Docker-in-Docker:

Setting up a Docker Swarm with multiple nodes on local host requires spinning up multiple virtual machines locally and Docker Swarm setup on VMs consume lot of memory *(each virtual machine with Docker running consumes at least 1 GB just to get started)* and time consuming. As an alternative, we can use Docker containers to act as cluster nodes and setup Docker Swarm on these nodes.

Here, we will set up Docker Swarm cluster with **2 Swarm Managers** *(one manager is the local host and other manager is the container acting as cluster node)* and **3 Swarm Workers** *(all workers are containers acting as cluster nodes).*

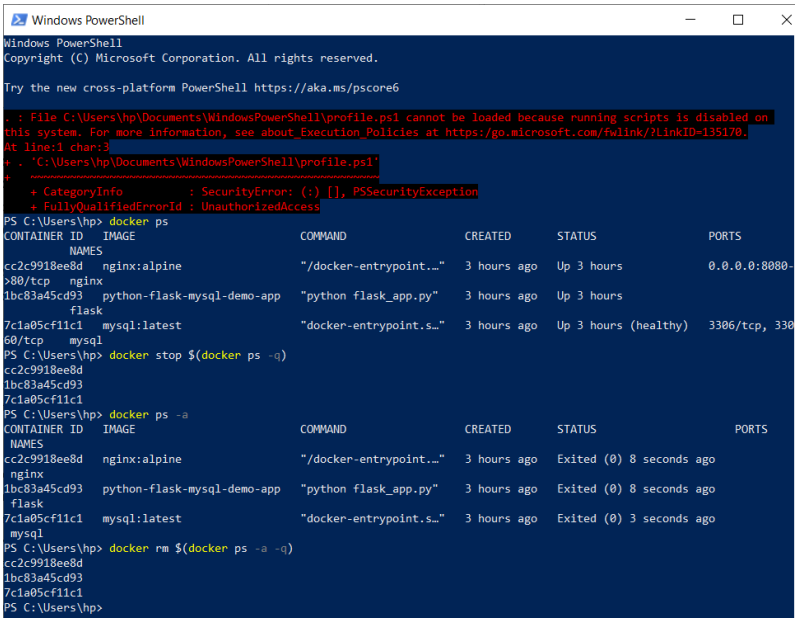
9.2. Stop and Delete Containers:

Before setting Docker Swarm, let us first stop all running containers and remove all containers and delete all available images along with any saved volumes.

Open a new **Command Prompt** or **Windows PowerShell** and run the following commands:

Stop and delete all containers:

```
docker ps
docker stop $(docker ps -q)
docker ps -a
docker rm $(docker ps -a -q)
```



Delete all images:

```
docker image ls
docker rmi $(docker image ls -q)
docker rmi -f $(docker image ls -q)
docker image ls
```

```
PS C:\Users\hp> docker image ls
REPOSITORY                    TAG       IMAGE ID       CREATED       SIZE
python-flask-mysql-demo-app   latest    1eee30b317e1   5 hours ago   62.3MB
nginx                         alpine    6769dc3a703c   7 weeks ago   48.2MB
mysql                         latest    edbdd97bf78b   7 weeks ago   859MB
PS C:\Users\hp> docker rmi $(docker image ls -q)
Untagged: python-flask-mysql-demo-app:latest
Deleted: sha256:1eee30b317e193e2e9e225ce68745de1f0ffb61d131607f3dcb1c70d39184dd6
Untagged: nginx:alpine
Untagged: nginx@sha256:65645c7bb6a0661892a8b03b89d0743208a18dd2f3f17a54ef4b76fb8e2f2a10
Deleted: sha256:6769dc3a703c719c1d2756bda113659be28ae16cf0da58dd5fd823d6b9a050ea
Untagged: mysql:latest
Untagged: mysql@sha256:04768cb63395f56140b4e92cad7c8d9f48dfa181075316e955da75aadca8a7cd
Deleted: sha256:edbdd97bf78b4338bbb96fa1348c8743a328b97ea3290b20adcab25bc17637de
Deleted: sha256:5ee46883cd23ccea0c130113edfca5ef2b202eb1866c6a0e868fc1f556755c75
Deleted: sha256:cc7bc92145face1c727f2014d0341f21d18247e0c7df0543e2d6ac020a6dbdbc
Deleted: sha256:a0229cd926f7a0e0afeea987e0b9d4419964e02d6d84330aee7615863687deb0
Deleted: sha256:54366fcda8e9d2e5cd0f3452adb3b2ab34c6216aa73e24889cbdcdb15788f399
Deleted: sha256:c3e04f405b930b3c39ffeeaa9583e72d1f007fc3d6a208a894ac8c793f02fa84
Deleted: sha256:8ec93f6326350a63a5858a425afcd3028ed2dcc6303784c51359b48c032916ad
Deleted: sha256:24882858314385467bf296310805d4b7229bff39c29ed9a434689adc37543291
Deleted: sha256:35caa70a7a17d9478f62ffd10c7fe27493877644fdbe82d181bc5bb93e8e0f62
Deleted: sha256:c222c78b1ef751569013c1ec5a3ad2b5a8254b3600920aaaf358e0b32872bf23
Deleted: sha256:825f4932222fe8e4ee14d896e0f324da04fb8420e5335f3ee039f32fa47e5ee3
PS C:\Users\hp> docker rmi -f $(docker image ls -q)
docker: 'docker rmi' requires at least 1 argument

Usage:  docker rmi [OPTIONS] IMAGE [IMAGE...]

See 'docker rmi --help' for more information
PS C:\Users\hp> docker image ls
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE
PS C:\Users\hp>
```

Delete all volumes:

```
docker volume ls
docker volume rm $(docker volume ls -q)
docker volume ls
```

```
PS C:\Users\hp> docker volume ls
DRIVER    VOLUME NAME
local     python-flask-mysql-demo_db-data
PS C:\Users\hp> docker volume rm $(docker volume ls -q)
python-flask-mysql-demo_db-data
PS C:\Users\hp> docker volume ls
DRIVER    VOLUME NAME
PS C:\Users\hp>
```

Going forward, we may need to execute few Linux commands related to Docker Swarm for which we need to use **GitBash** command prompt.

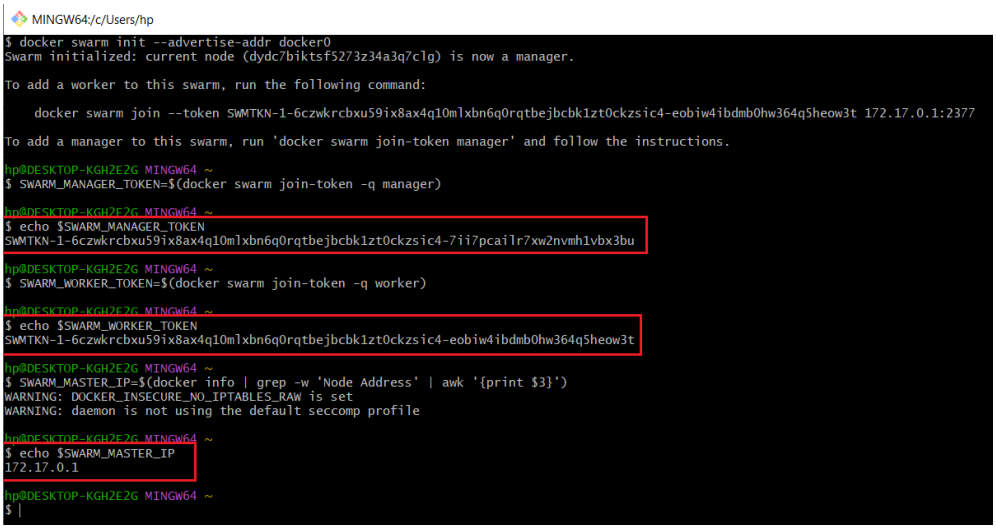
9.3. Initialize Swarm Cluster:

Open a **Gitbash** prompt and run the following commands to initialize the Docker Swarm with the local host as the Swarm Manager and setup environment variables for the token to be used to join manager and worker nodes and for the IP address of the master node.

```
docker swarm init --advertise-addr docker0

SWARM_MANAGER_TOKEN=$(docker swarm join-token -q manager)
echo $SWARM_MANAGER_TOKEN
```

```
SWARM_WORKER_TOKEN=$(docker swarm join-token -q worker)
echo $SWARM_WORKER_TOKEN

SWARM_MASTER_IP=$(docker info | grep -w 'Node Address' | awk '{print
$3}')
echo $SWARM_MASTER_IP
```

Note: In the above command, `docker0` is a virtual bridge interface created by Docker during installation and it has a default IP address of `172.17.0.1` but randomly chooses an address and subnet from a private defined range. All the Docker containers are connected to this bridge and use the NAT rules created by docker to communicate with the outside world.



If you don't want to use `docker0` bridge, the Docker Swarm can also be initialized using the default local host IP `127.0.0.1` as below:

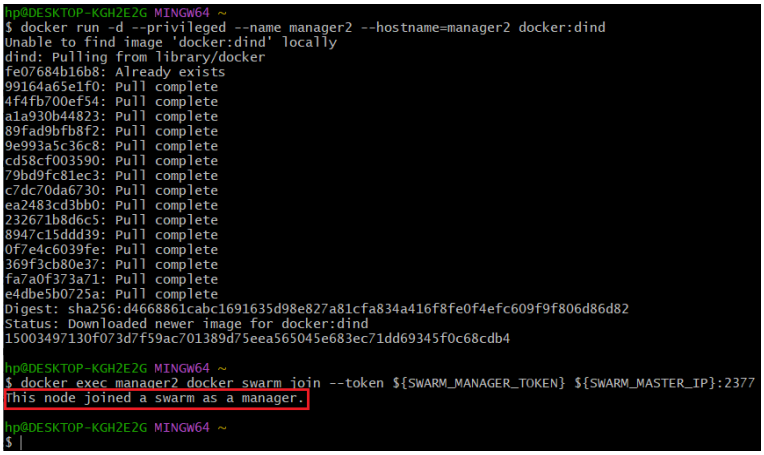
```
docker swarm init --advertise-addr 127.0.0.1
```

9.4. Add Manager Node:

Run the following command to launch a Docker container and add it as a manager node to the Docker Swarm:

```
docker run -d --privileged --name manager2 --hostname=manager2
docker:dind

docker exec manager2 docker swarm join --token ${SWARM_MANAGER_TOKEN}
${SWARM_MASTER_IP}:2377
```

It takes few minutes to download the Docker image and start the container

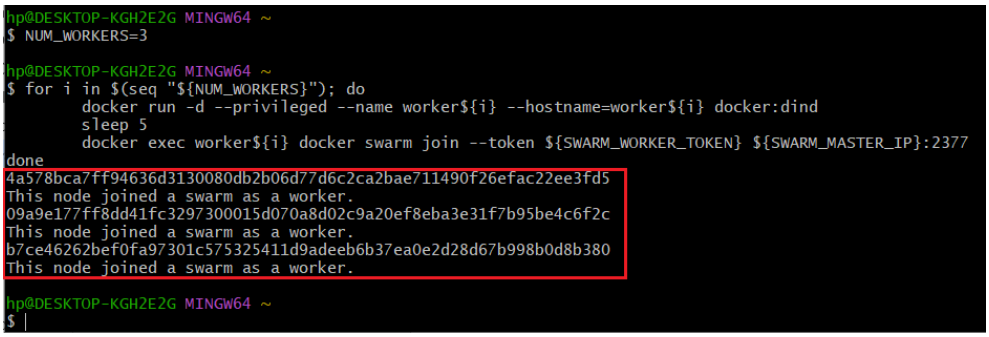


Once the node has joined the Swarm successfully, it displays message **“This node joined a swarm as a manager”.**

9.5. Add Worker Nodes:

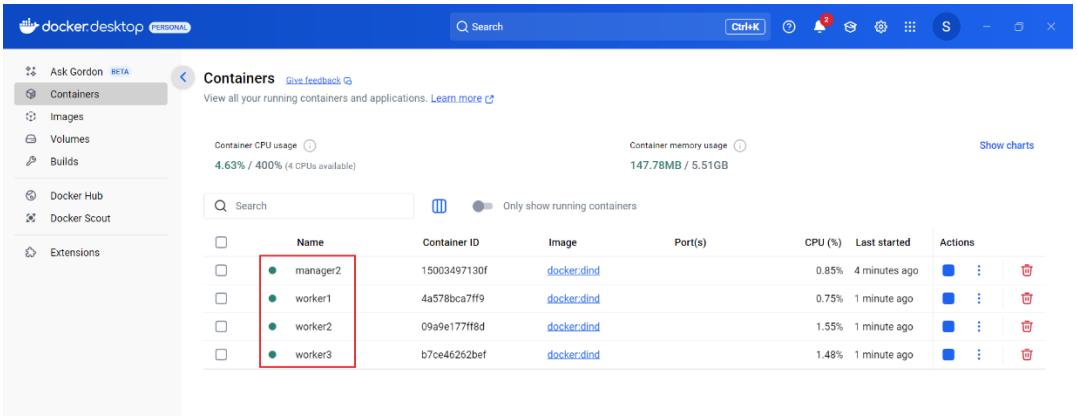
Run the following commands to launch 3 Docker containers and add those as worker nodes to the Docker Swarm:

```
NUM_WORKERS=3

for i in $(seq "${NUM_WORKERS}"); do
	docker run -d --privileged --name worker${i} --
hostname=worker${i} docker:dind
	sleep 5
	docker exec worker${i} docker swarm join --token
${SWARM_WORKER_TOKEN} ${SWARM_MASTER_IP}:2377
done
```



Once the above commands are executed successfully, it displays the container ID and a message **“This node joined a swarm as a worker”.**

Open **Docker Desktop** application where you can see 4 containers namely `manager2`, `worker1`, `worker2` and `worker3` which are in running state.



9.6. Start Visualizer Service:

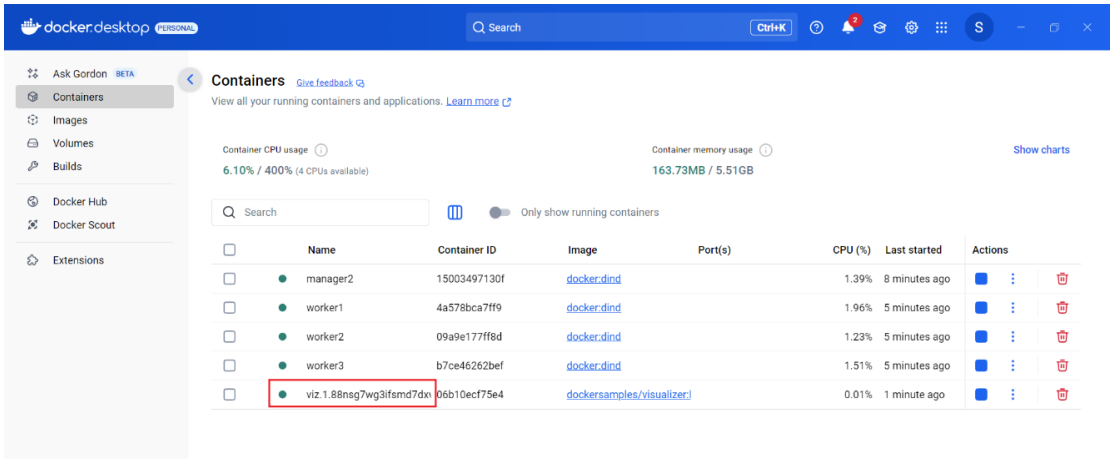
Docker provides **visualizer** image to monitor the state of Docker Swarm cluster. This visualizer image is a sample app created by Docker Samples for learning Docker Swarm and is available in Docker Hub.

Run the following command to create the `visualizer` service on the manager node. It might take some time to complete due to the required image to be download and service to be deployed on the node.

```
docker service create --name=viz --publish=8085:8080/tcp --
constraint=node.role==manager --
mount=type=bind,src=//var/run/docker.sock,dst=//var/run/docker.sock
dockersamples/visualizer
```

```
hp@DESKTOP-KGH2E2G MINGW64 ~
$ docker service create --name=viz --publish=8085:8080/tcp --constraint=node.role==manager --mount=type=bind,src=//var/run/docker.sock,dst=//var/run/docker.sock
 dockersamples/visualizer
kyqek6i8mz8j9g1jn4qsn1xef
overall progress: 1 out of 1 tasks
1/1: running   [==================================================>]
verify: Service kyqek6i8mz8j9g1jn4qsn1xef converged

hp@DESKTOP-KGH2E2G MINGW64 ~
$
```

In **Docker Desktop**, we can see a new container name prefixed with `viz` has been created and running since it was deployed on the primary manager node which is our local Docker host.

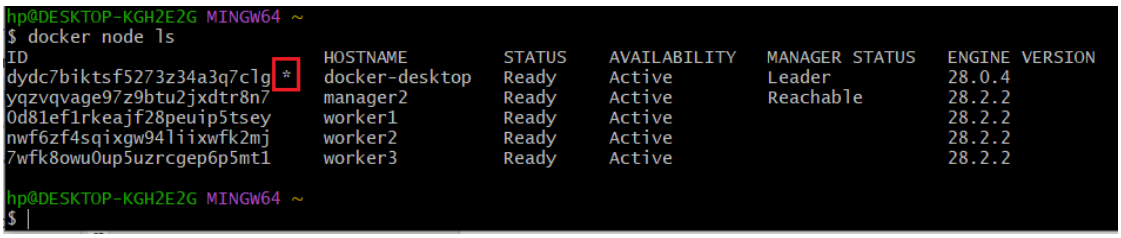


9.7. Verify Docker Swarm:

Now, let us verify if the Docker Swarm cluster is up and running with 2 manager nodes and 3 worker nodes.

Run the following command on the **Git Bash**. Note that this command works only on the manager node.

```
docker node ls
```

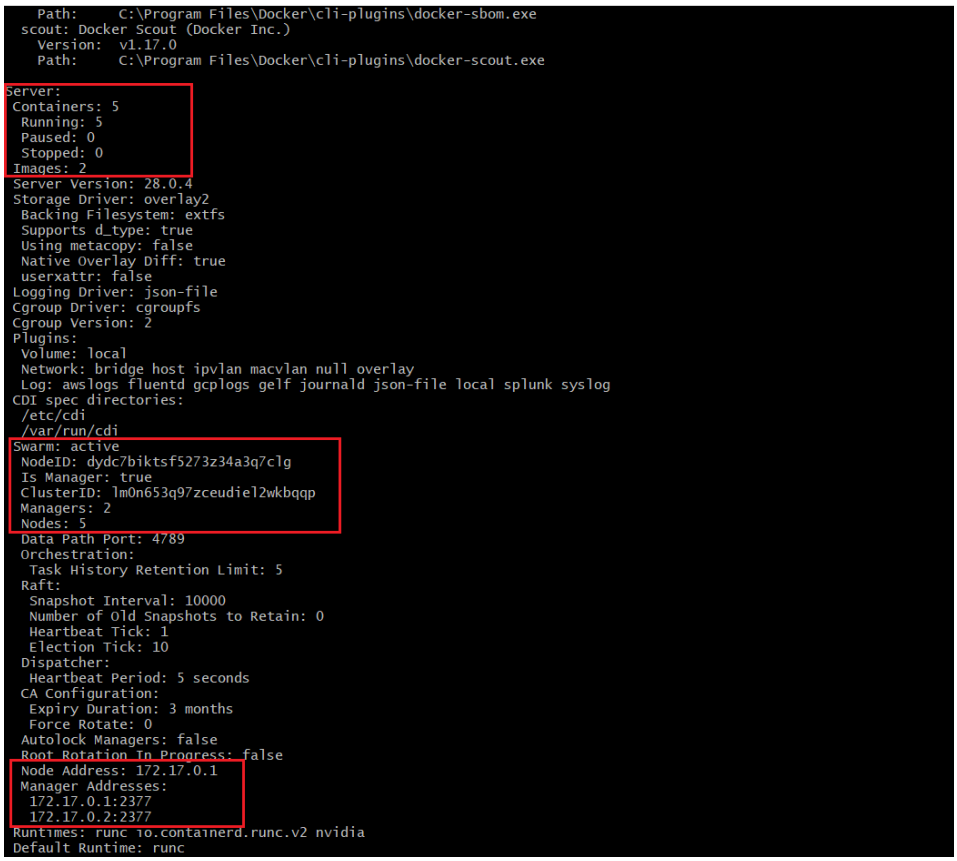


As we can see above, it displays the information of all Docker Swarm nodes:

- The ***** next to the node **ID** indicates that we are currently connected to this node.
- The **STATUS** column with **Ready** value indicates the node is recognized by the manager and can participate in the cluster. If the **STATUS** shows **DOWN**, the node is not recognized by the manager and cannot participate in the cluster
- The **AVAILABILITY** column with **Active** value indicates the node participates in the cluster and acts as a worker to accept new tasks from the swarm manager. If the **STATUS** shows **DOWN**, the node is not recognized by the manager and cannot participate in the cluster.
- The **MANAGER_STATUS** column with **Leader** value indicates the node is a manager and is also the cluster's leader. The **Reachable** value indicates the node is a manager and can become the leader when the current leader is not working. The empty value indicates the node is worker and can be promoted to a manager when needed.

The current status of Docker Swarm can also be verified using the below command which can be executed on any node:

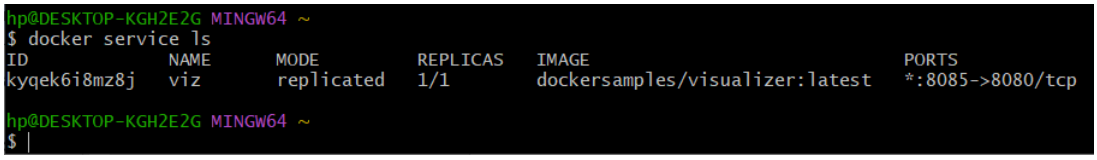
```
docker info
```



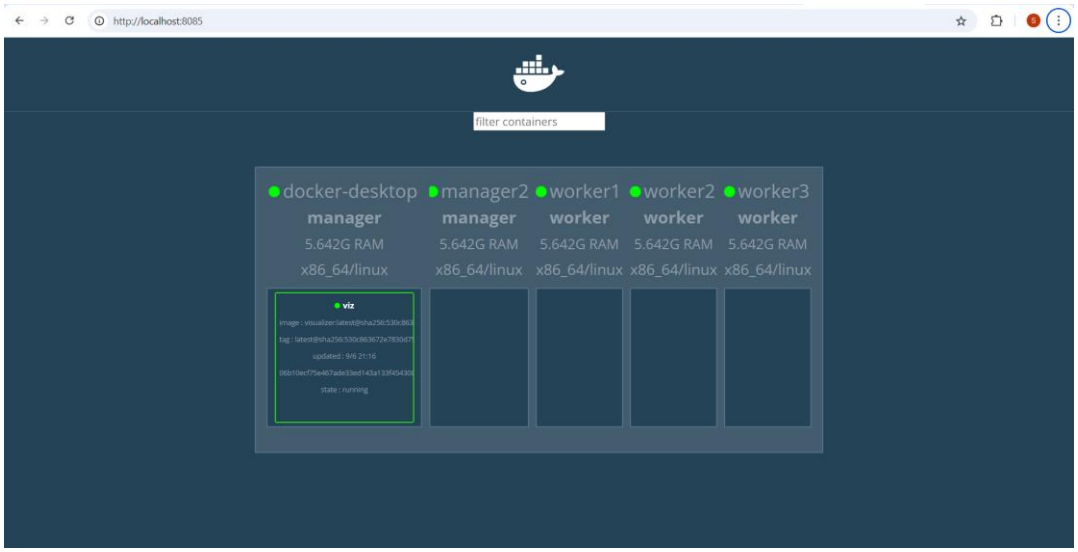
As we can see above, it displays the current node information along with the **Swarm** status as **Active** with **5 Nodes** of which **2** are **Manager** nodes.

Verify if the Visualizer service is up and running using the below command

```
docker service ls
```



Open **Visualizer** UI at http://localhost:8085/ to see the Swarm cluster. At this moment, it displays single `viz` container which is a Visualizer service on the primary manager node.

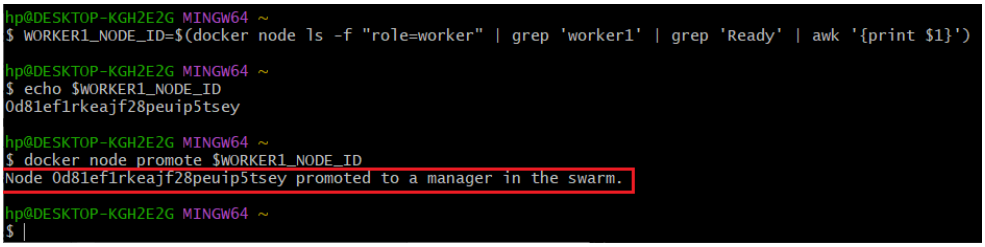


9.8. Promote or Demote Node:

At any point, a worker node can be promoted as a manager and a manager node can be demoted as a worker.

For the purpose of this demo, run the following command to promote `worker1` node as manager

```
WORKER1_NODE_ID=$(docker node ls -f "role=worker" | grep 'worker1' | grep 'Ready' | awk '{print $1}')

echo $WORKER1_NODE_ID

docker node promote $WORKER1_NODE_ID
```



Verify if the `worker1` node got promoted to manager using the below command:

```
docker node ls
```

```
hp@DESKTOP-KGH2E2G MINGW64 ~
$ docker node ls
ID                            HOSTNAME          STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
dydc7biktsf5273z34a3q7clg *   docker-desktop    Ready     Active         Leader           28.0.4
yqzvqvage97z9btu2jxdtr8n7     manager2          Ready     Active         Reachable        28.2.2
0d81ef1rkeajf28peuip5tsey     worker1           Ready     Active         Reachable        28.2.2
nwf6zf4sqixgw94liixwfk2mj     worker2           Ready     Active                          28.2.2
7wfk8owu0up5uzrcgep6p5mt1     worker3           Ready     Active                          28.2.2

hp@DESKTOP-KGH2E2G MINGW64 ~
$
```

As you can see above, `MANAGER STATUS` of `worker1` has been changed to `Reachable` which indicates it is a manager.

Let us demote this new manager node to worker using the following command and verify its status:

```
MANAGER_NODE_ID=$(docker node ls -f "role=manager" | grep 'Reachable' | grep 'worker1' | awk '{print $1}')

echo $MANAGER_NODE_ID

docker node demote $MANAGER_NODE_ID

docker node ls
```

```
hp@DESKTOP-KGH2E2G MINGW64 ~
$ MANAGER_NODE_ID=$(docker node ls -f "role=manager" | grep 'Reachable' | grep 'worker1' | awk '{print $1}')

hp@DESKTOP-KGH2E2G MINGW64 ~
$ echo $MANAGER_NODE_ID
0d81ef1rkeajf28peuip5tsey

hp@DESKTOP-KGH2E2G MINGW64 ~
$ docker node demote $MANAGER_NODE_ID
Manager 0d81ef1rkeajf28peuip5tsey demoted in the swarm.

hp@DESKTOP-KGH2E2G MINGW64 ~
$ docker node ls
ID                            HOSTNAME          STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
dydc7biktsf5273z34a3q7clg *   docker-desktop    Ready     Active         Leader           28.0.4
yqzvqvage97z9btu2jxdtr8n7     manager2          Ready     Active         Reachable        28.2.2
0d81ef1rkeajf28peuip5tsey     worker1           Ready     Active                          28.2.2
nwf6zf4sqixgw94liixwfk2mj     worker2           Ready     Active                          28.2.2
7wfk8owu0up5uzrcgep6p5mt1     worker3           Ready     Active                          28.2.2

hp@DESKTOP-KGH2E2G MINGW64 ~
$
```

As you can see above, the `MANAGER STATUS` of `worker1` has been set to blank which indicates it is a worker.

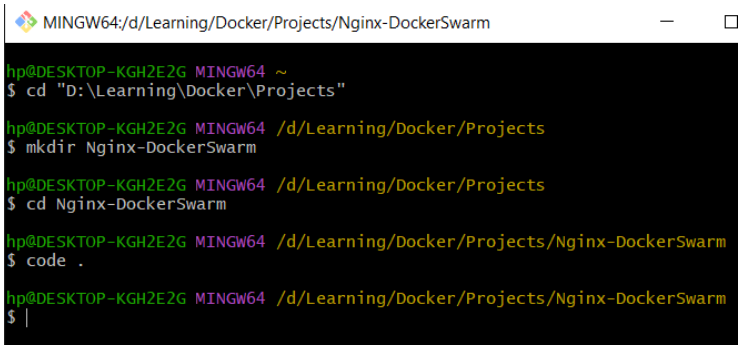
9.9. Deploy High availability Service:

Now, we will deploy the **Nginx** service to the cluster using High Availability Swarm configuration. For this purpose, we will create a compose file and deploy the service as a Swarm stack.

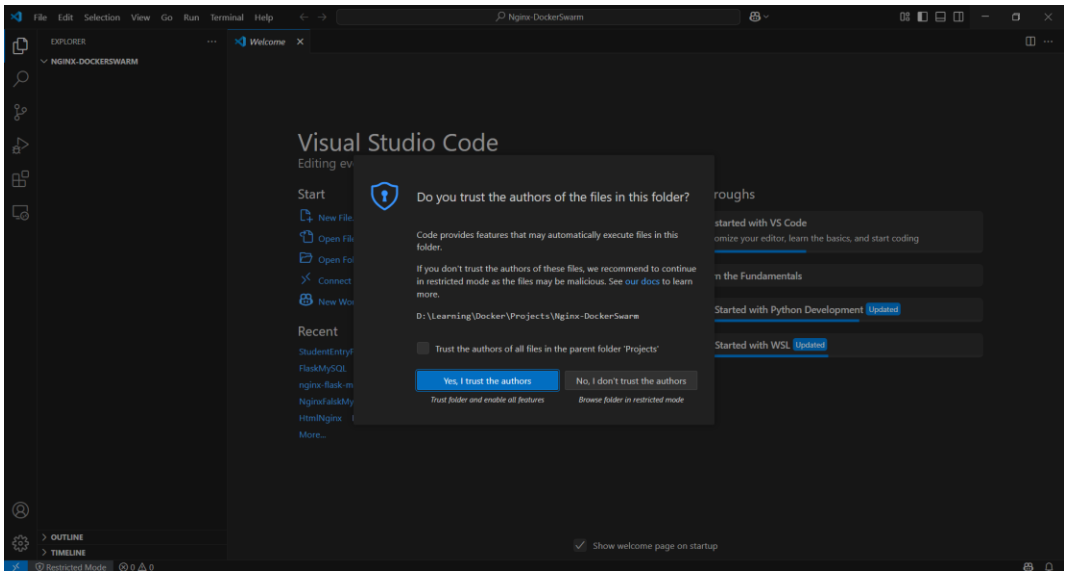
9.9.1. Launch VS Code:

Open **Git Bash** command prompt, run the following commands to create a new directory at `D:\Learning\Docker\Projects` location and open **VS Code**. You can choose any location to create a new directory.

```
cd "D:\Learning\Docker\Projects"
mkdir Nginx-DockerSwarm
cd Nginx-DockerSwarm
code .
```

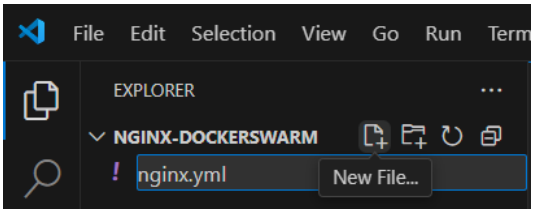


It opens **VS Code** application which might prompt you to confirm if you trust the authors of file in which case, click on **Yes, I trust the authors** button.



9.9.2. Create Compose File:

In **VS Code** application, click on **New File** icon next to `NGINX-DOCKERSWARM` project under **EXPLORER** and name the file as `nginx.yml`



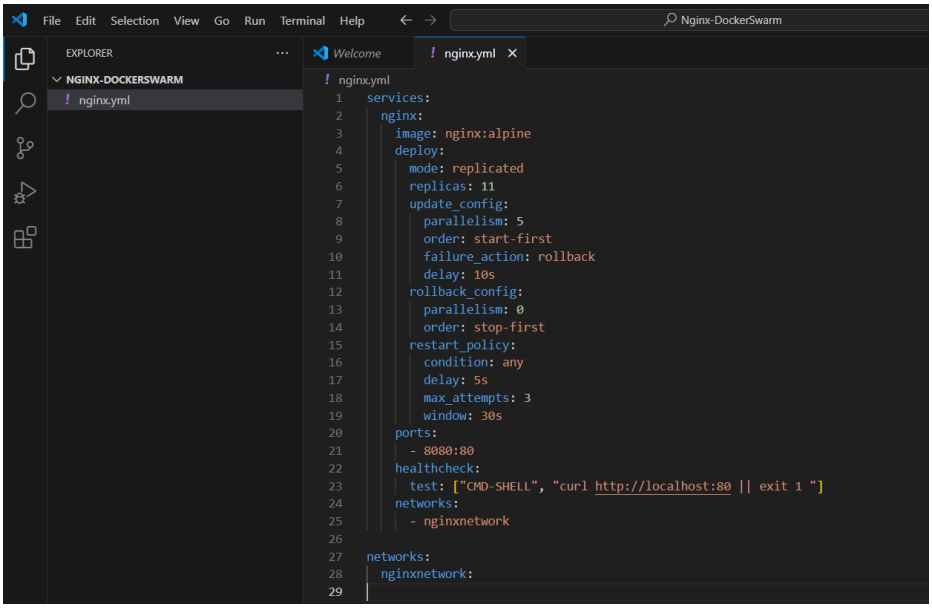
In the `nginx.yml` file, write the following lines of code and save it.

```
services:
  nginx:
    image: nginx:alpine
    deploy:
      mode: replicated
      replicas: 11
      update_config:
        parallelism: 5
        order: start-first
        failure_action: rollback
        delay: 10s
      rollback_config:
        parallelism: 0
        order: stop-first
      restart_policy:
        condition: any
        delay: 5s
        max_attempts: 3
        window: 30s
    ports:
      - 8080:80
    healthcheck:
      test: ["CMD-SHELL", "curl http://localhost:80 || exit 1 "]
    networks:
      - nginxnetwork

networks:
  nginxnetwork:
```

The above code performs the following:

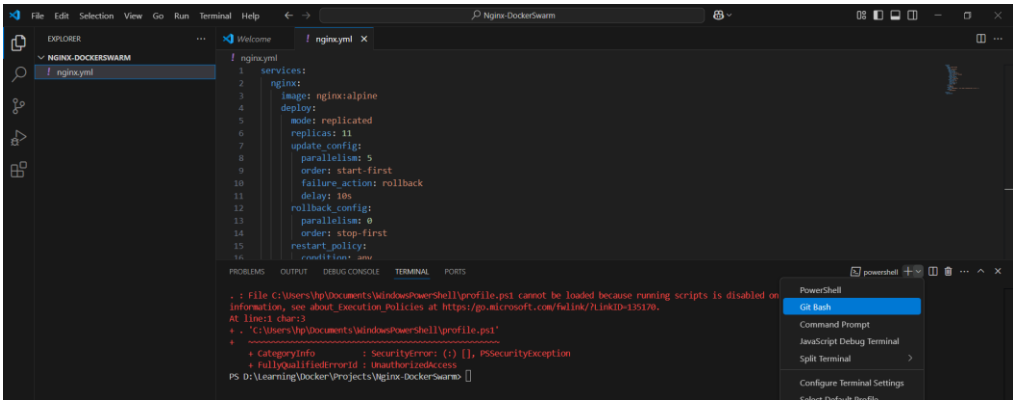
- Create `nginx` service that
    - Pulls `nginx:alpine` image
    - Deploys service in replicated mode with:
        - 11 replicas
        - updates 5 containers at once by starting the new task first while the running tasks briefly overlap and rollbacks containers when an update fails and updates containers at an interval of 10 seconds
        - rollbacks all containers simultaneously by stopping the old task before starting a new one

        - restarts containers regardless of exit status and waits for `5 seconds` between each restart attempt for maximum of `3` restart attempts and waits for `30 seconds` before attempting another restart in case of service failure.
    - Maps local host port `8080` to container port `80`.
    - Performs healthcheck to verify web URL of `nginx` server and exit with `1` if it returns any failure.
    - Maps the container to `ngnixnetwork` network.
- Create `nginxnetwork` network



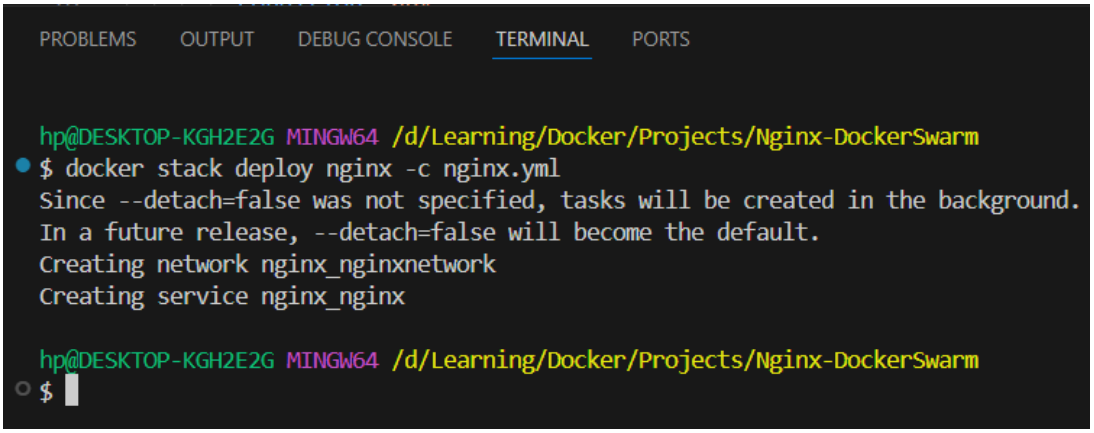
9.9.3. Deploy Stack:

In VS Code, go to **Terminal** menu and select **New Terminal** and choose **Git Bash** terminal from the dropdown in the **Terminal** tab below.



On the Terminal, run the following command to deploy `nginx` service on the Docker Swarm:

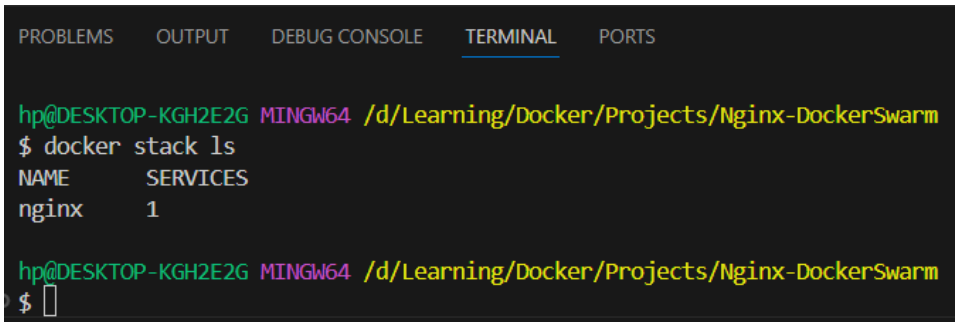
```
docker stack deploy nginx -c nginx.yml
```



9.9.4. Verify Stack and Services:

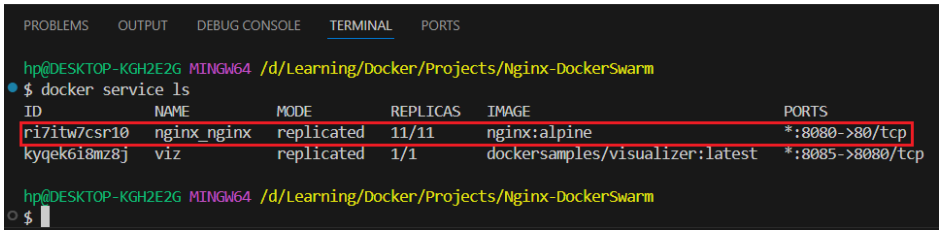
Verify if the stack has been created using the below command:

```
docker stack ls
```



After couple of minutes, check the status of the service using the below command:

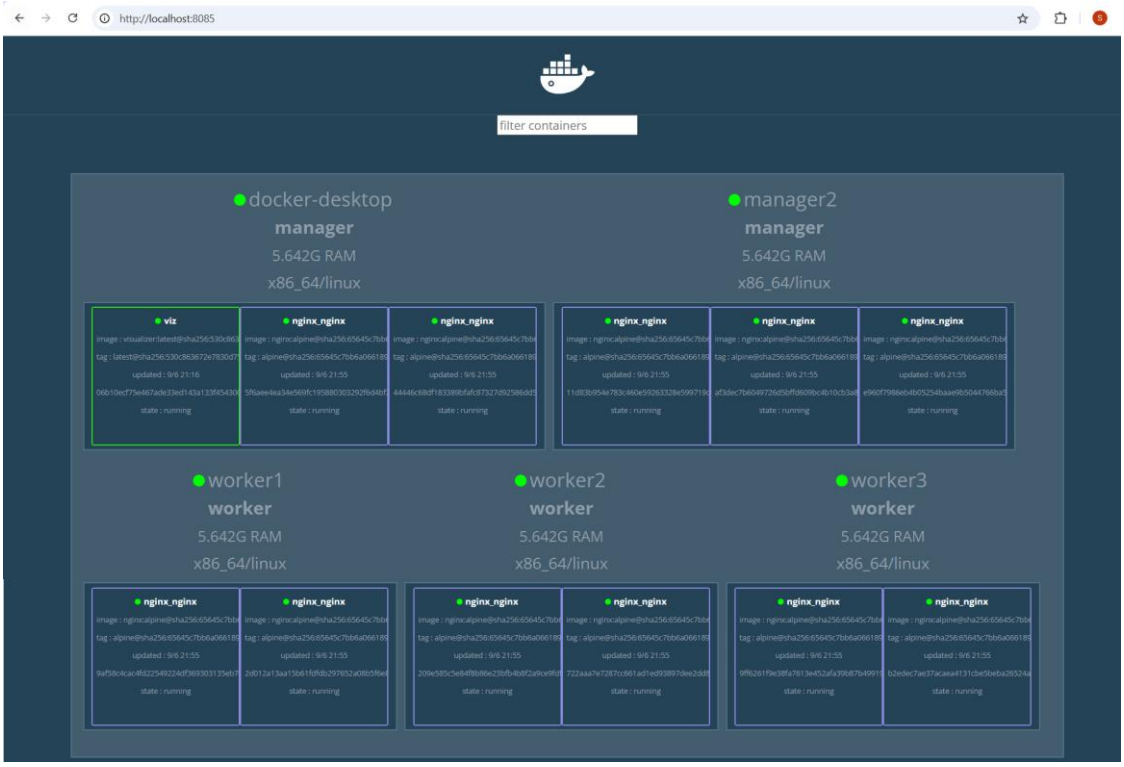
```
docker service ls
```



It displays `nginx` service created with 11 replicas.

9.9.5. Verify Replicas:

Go to Visualizer application at http://localhost:8085 and you can see how 11 replicas got distributed onto each node.



Use the following command to see the list of nodes running the `nginx` service:

```
docker service ps $(docker service ls --filter name=nginx -q)
```

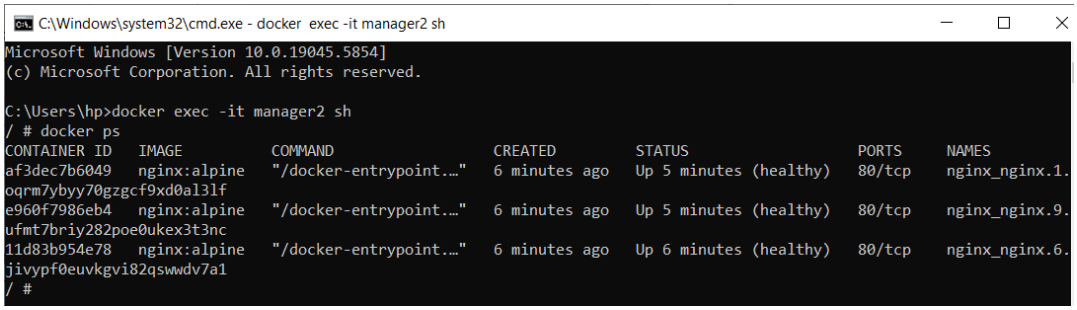
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker service ps $(docker service ls --filter name=nginx -q)
ID             NAME             IMAGE          NODE             DESIRED STATE   CURRENT STATE            ERROR     PORTS
oqrm7ybyy70g   nginx_nginx.1    nginx:alpine   manager2         Running         Running 3 minutes ago
78z0f5wwntiz   nginx_nginx.2    nginx:alpine   worker1          Running         Running 3 minutes ago
4bkahuh1td3q   nginx_nginx.3    nginx:alpine   worker3          Running         Running 3 minutes ago
tntatll5ebzj   nginx_nginx.4    nginx:alpine   docker-desktop   Running         Running 4 minutes ago
md6j8flm8oas   nginx_nginx.5    nginx:alpine   worker2          Running         Running 3 minutes ago
jivypf0euvkg   nginx_nginx.6    nginx:alpine   manager2         Running         Running 3 minutes ago
tz6nkxtocqvv   nginx_nginx.7    nginx:alpine   docker-desktop   Running         Running 4 minutes ago
iww05y7z1b91   nginx_nginx.8    nginx:alpine   worker3          Running         Running 3 minutes ago
ufmt7briy282   nginx_nginx.9    nginx:alpine   manager2         Running         Running 3 minutes ago
4mf990mv8893   nginx_nginx.10   nginx:alpine   worker1          Running         Running 3 minutes ago
bkvmf3q5csah   nginx_nginx.11   nginx:alpine   worker2          Running         Running 3 minutes ago

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$
```

9.9.6. Validate Task Containers:

Now, to check the details about the respective container for a task, we should login to the relevant node.

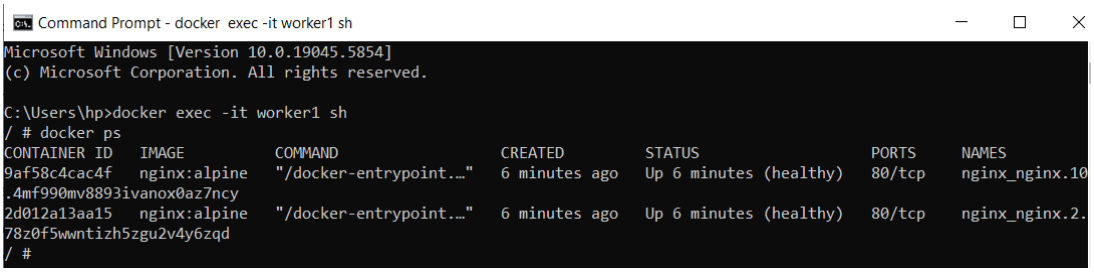
Open new **Command Prompt** or **Windows Powershell** and run the below commands to connect to `manager2` instance and verify the containers running:

```
docker exec -it manager2 sh
docker ps
```



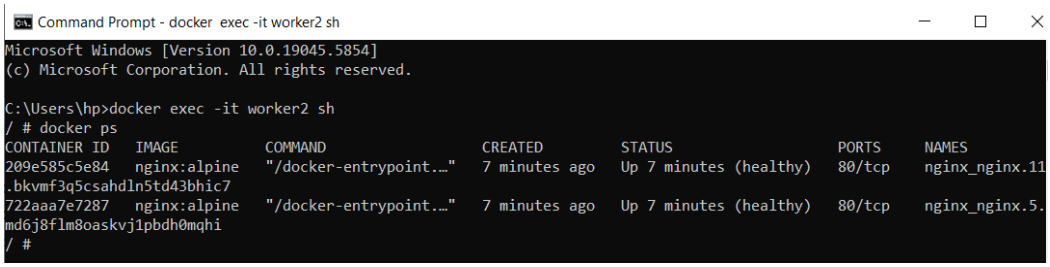
Open new **Command Prompt** and run the below commands to connect to `worker1` instance and verify the containers running:

```
docker exec -it worker1 sh
docker ps
```



Open new **Command Prompt** and run the below commands to connect to `worker2` instance and verify the containers running:

```
docker exec -it worker2 sh
docker ps
```



Open new **Command Prompt** and run the below commands to connect to `worker3` instance and verify the containers running:

```
docker exec -it worker3 sh
docker ps
```

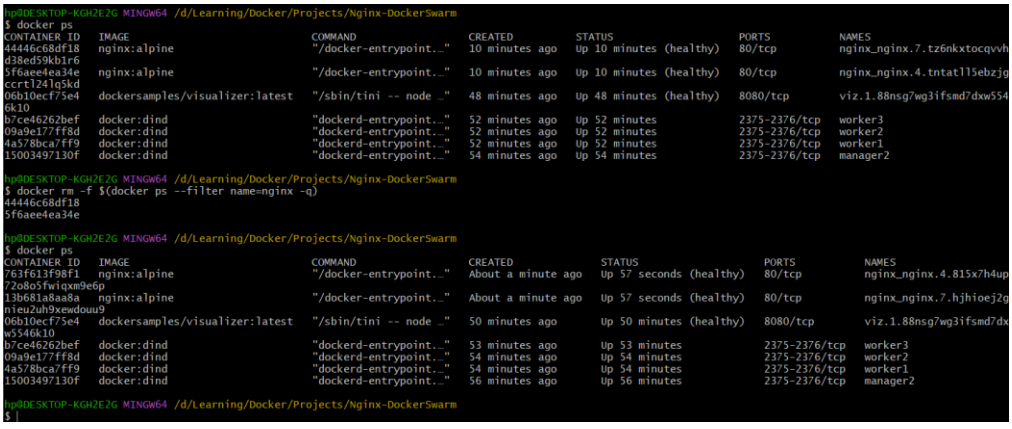
9.10. Test Resilience:

Now, let us see if the Docker Swarm can handle to unexpected problems.

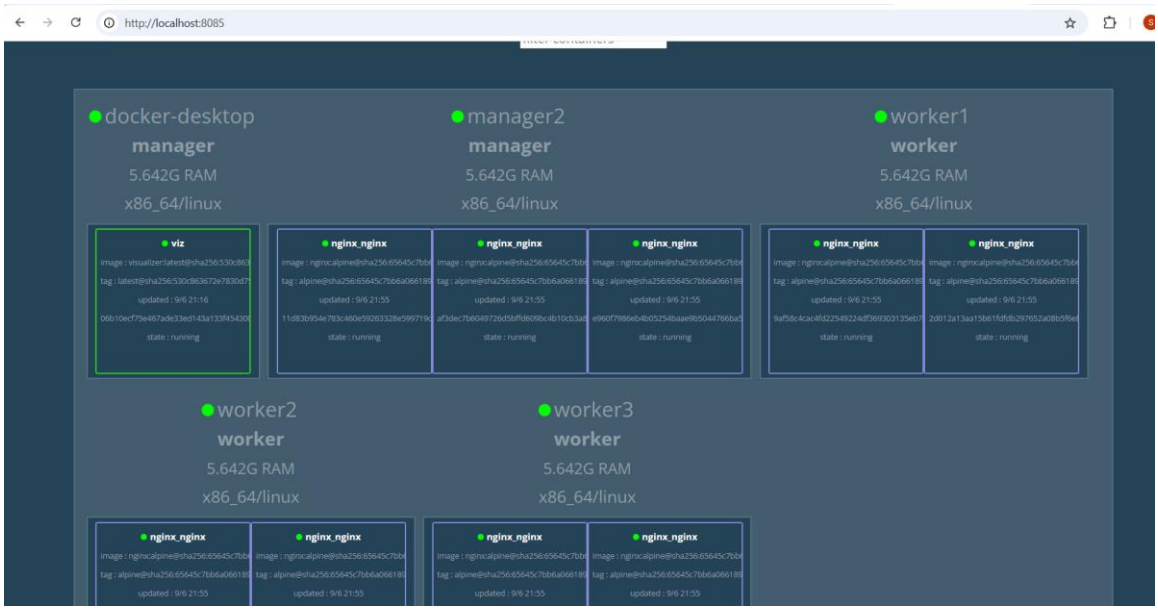
9.10.1. Delete Containers:

As there are 3 `nginx` containers running on the leader node, let us shutdown these containers and verify if Docker Swarm can detect and start new ones.

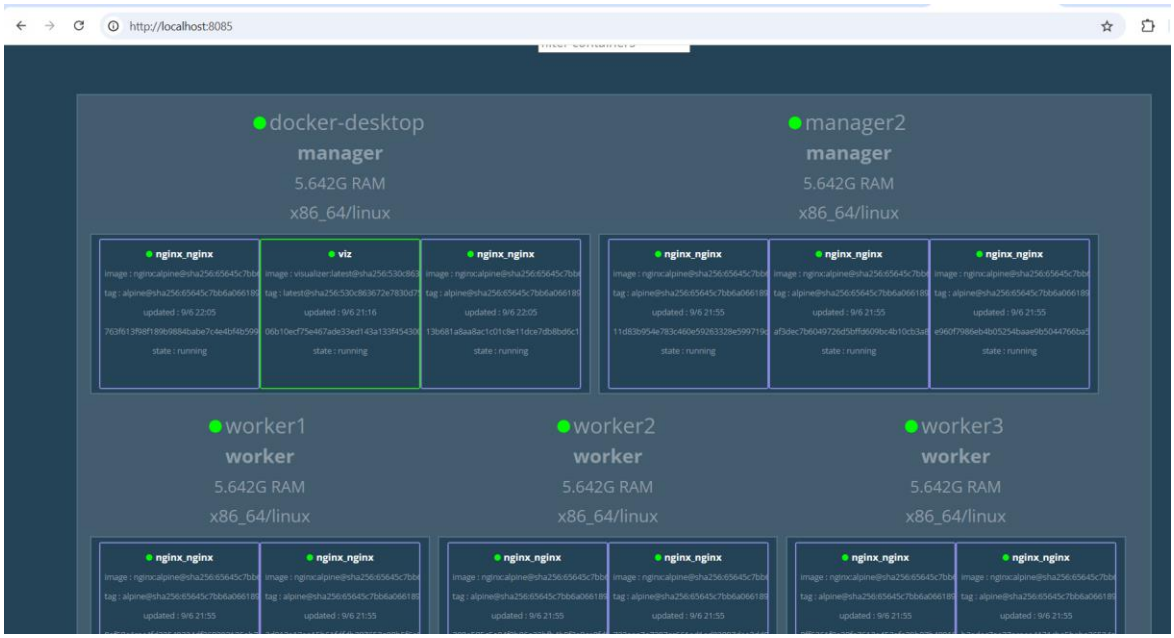
```
docker ps
docker rm -f $(docker ps --filter name=nginx -q)
docker ps
```



Do a quick check in the Visualizer application where you can see 2 containers have been removed.



Immediately, 2 new containers have been started on the leader node as soon as the old containers were shutdown.

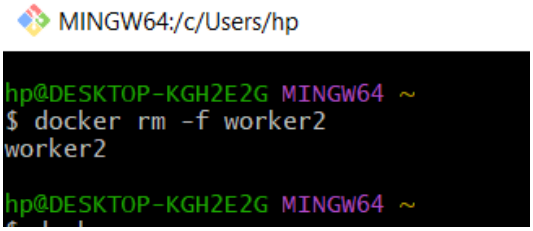


9.10.2. Crash Node:

Now, let us make it even worse of crashing a worker node.

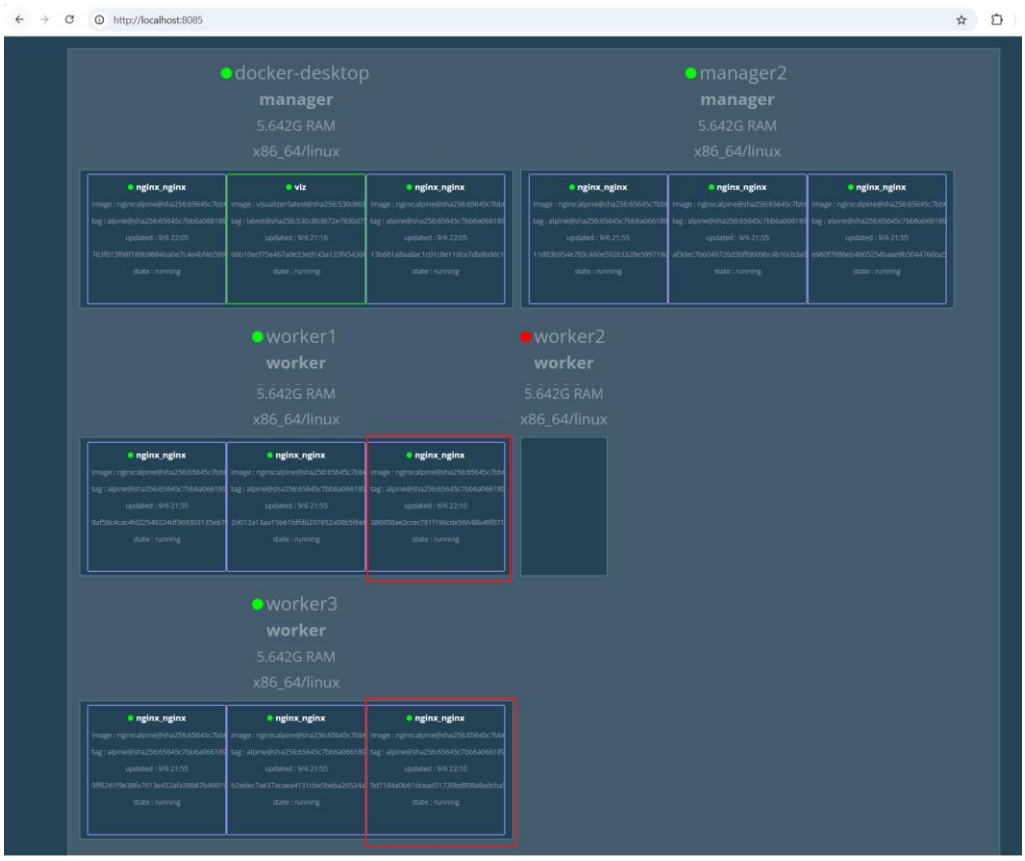
In the **GitBash** prompt, run the following command to remove `worker2` container that is acting as a Swarm node.

```
docker rm -f worker2
```



The Visualizer also reports that `worker2` node is down *(status changed to Red)* and containers on `worker2` are removed.

But in few seconds, the containers that were running on `worker2` are rescheduled to other nodes which is visible in Visualizer:

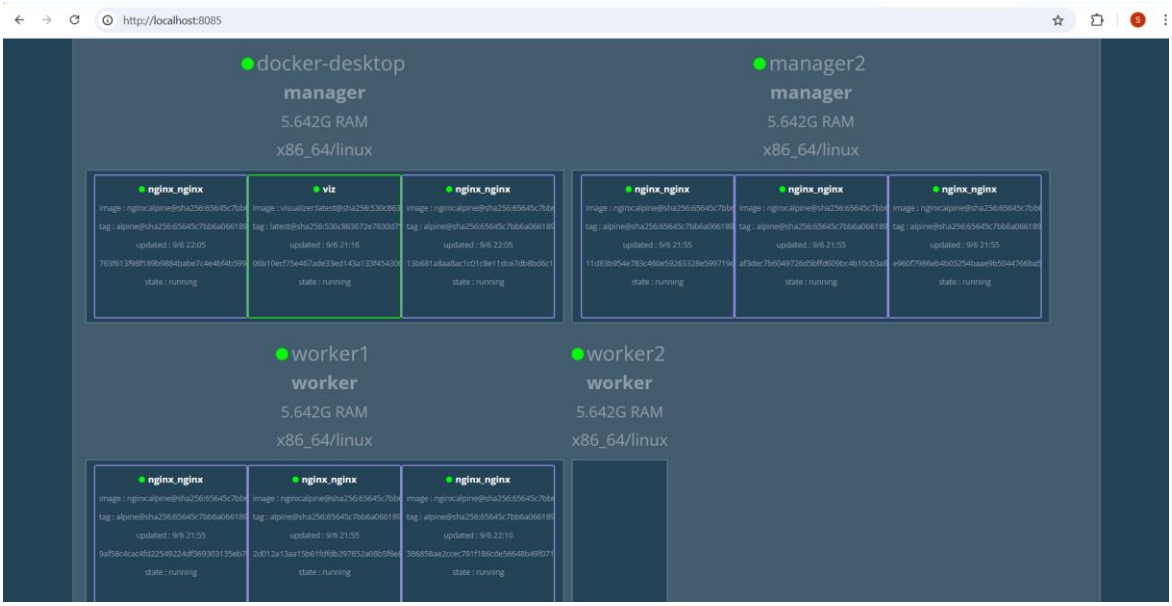


Let us restart the failed node by executing the following commands on the **GitBash**:

```
SWARM_WORKER_TOKEN=$(docker swarm join-token -q worker)
SWARM_MASTER_IP=$(docker info | grep -w 'Node Address' | awk '{print
$3}')

i=2
docker node rm worker${i}
docker run -d --privileged --name worker${i} --hostname=worker${i}
docker:dind
docker exec worker${i} docker swarm join --token
${SWARM_WORKER_TOKEN} ${SWARM_MASTER_IP}:2377
docker node ls
```

```
hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ SWARM_WORKER_TOKEN=$(docker swarm join-token -q worker)

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ SWARM_MASTER_IP=$(docker info | grep -w 'Node Address' | awk '{print $3}')
WARNING: DOCKER_INSECURE_NO_IPTABLES_RAW is set
WARNING: daemon is not using the default seccomp profile
WARNING: Running Swarm in a two-manager configuration. This configuration provides
         no fault tolerance, and poses a high risk to lose control over the cluster.
         Refer to https://docs.docker.com/engine/swarm/admin_guide/ to configure the
         Swarm for fault-tolerance.

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ i=2

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker node rm worker${i}
worker2

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker run -d --privileged --name worker${i} --hostname=worker${i} docker:dind
0e399b7166fde81ff3e1e19841c5ae2e89bb8ba21b3a56eec149be8ca4796257

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker exec worker${i} docker swarm join --token ${SWARM_WORKER_TOKEN} ${SWARM_MASTER_IP}:2377
docker node ls
This node joined a swarm as a worker.
ID                            HOSTNAME          STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
dydc7biktsf5273z34a3q7clg *   docker-desktop    Ready     Active         Leader           28.0.4
yqzvqvage97z9btu2jxdtr8n7     manager2          Ready     Active         Reachable        28.2.2
0d81ef1rkeajf28peuip5tsey     worker1           Ready     Active                          28.2.2
uizqqa89a5tsvdl7xl6fklv55     worker2           Ready     Active                          28.2.2
7wfk8owu0up5uzrcgep6p5mt1     worker3           Ready     Active                          28.2.2

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$
```

As we see above, `worker2` node is added back to Swarm. Refresh the Visualizer application which displays `worker2` node is back online. Though the node is online, the containers are not rebalanced to the new worker node automatically.



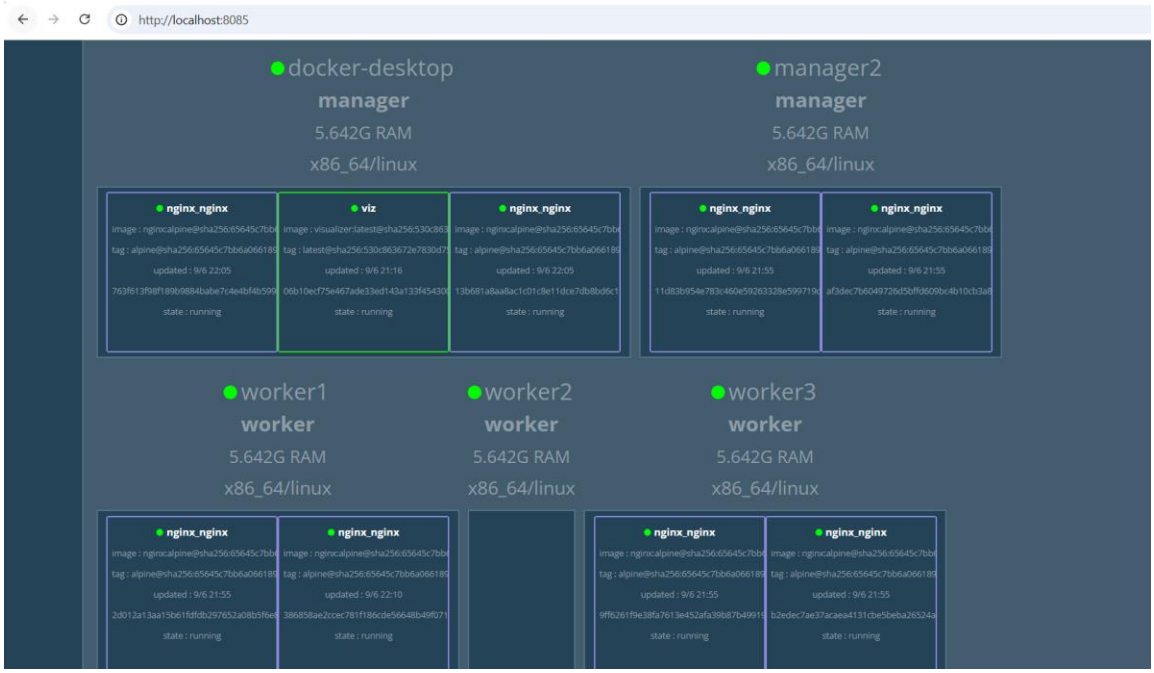
Now, we need to rebalance containers manually.

First, run the following command to scale down the containers:

```
docker service scale nginx_nginx=8
```

```
hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker service scale nginx_nginx=8
nginx_nginx scaled to 8
overall progress: 8 out of 8 tasks
1/8: running   [==================================================>]
2/8: running   [==================================================>]
3/8: running   [==================================================>]
4/8: running   [==================================================>]
5/8: running   [==================================================>]
6/8: running   [==================================================>]
7/8: running   [==================================================>]
8/8: running   [==================================================>]
verify: Service nginx_nginx converged

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$
```

Visualizer reflects 8 containers running:

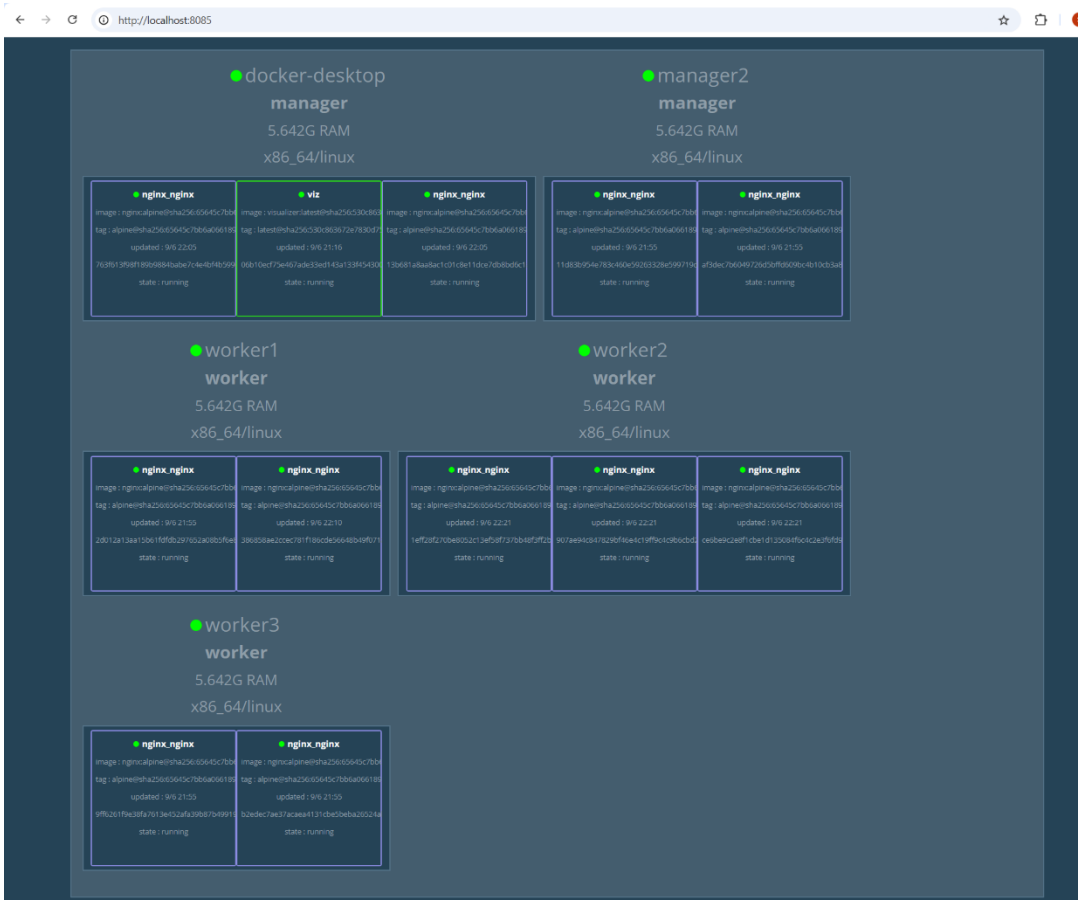


Next, run the following command to scale up the containers:

```
docker service scale nginx_nginx=11
```

```
hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker service scale nginx_nginx=11
nginx_nginx scaled to 11
overall progress: 11 out of 11 tasks
1/11: running   [==================================================>]
2/11: running   [==================================================>]
3/11: running   [==================================================>]
4/11: running   [==================================================>]
5/11: running   [==================================================>]
6/11: running   [==================================================>]
7/11: running   [==================================================>]
8/11: running   [==================================================>]
9/11: running   [==================================================>]
10/11: running   [==================================================>]
11/11: running   [==================================================>]
verify: Service nginx_nginx converged

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$
```

In **Visualizer**, you can see the containers are scheduled on `worker2` node.

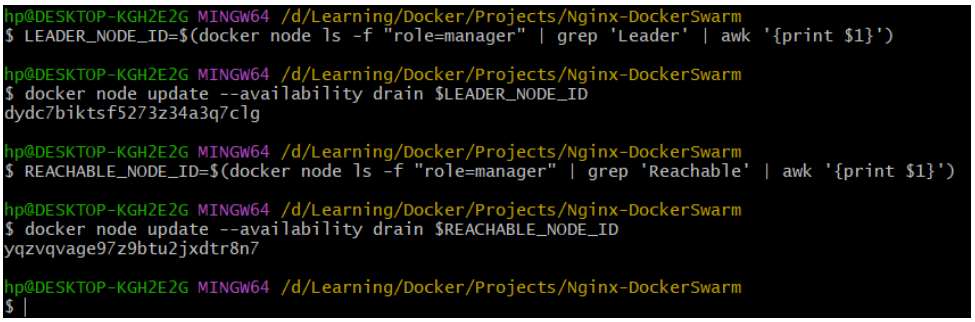


9.11. Drain Manager Nodes:

At this moment, all nodes (both managers and workers) are running with **Active** availability and accepting new tasks from the Swarm manager (including the leader).

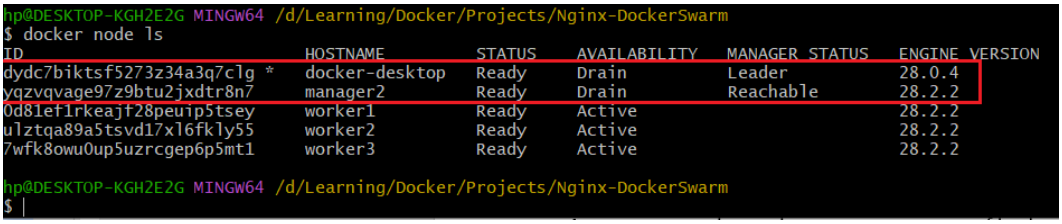
Since the manager nodes should ideally be only responsible for management-related tasks, let us drain manager nodes to avoid scheduling tasks on these nodes using the below commands:

```
LEADER_NODE_ID=$(docker node ls -f "role=manager" | grep 'Leader' |
awk '{print $1}')
docker node update --availability drain $LEADER_NODE_ID
REACHABLE_NODE_ID=$(docker node ls -f "role=manager" | grep
'Reachable' | awk '{print $1}')
docker node update --availability drain $REACHABLE_NODE_ID
```



Run the following command to verify if the availability of the manager node is set to `Drain`:

```
docker node ls
```



As you can see above, the `AVAILABILITY` of both manager nodes has been changed to `Drain`.

Since both manager nodes are updated to `Drain`, the `viz` container running on the leader manager node is stopped as there is no scaling set for the Visualizer and `ngnix` containers that were running on both manager nodes are subsequently stopped and reassigned to worker nodes to the maintain the scalability of 11 replicas at any given time.

Check the service status using the below command:

```
docker service ls
```

```
hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker service ls
ID             NAME           MODE         REPLICAS   IMAGE                          PORTS
ri7itw7csr10   nginx_nginx    replicated   11/11      nginx:alpine                   *:8080->80/tcp
kyqek6i8mz8j   viz            replicated   0/1        dockersamples/visualizer:latest   *:8085->8080/tcp

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$
```

It displays `0/1` are replicated which means the service is actually down.

Run the following command to check the distribution of `nginx` containers on various nodes

```
docker service ps nginx_nginx
```

```
hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker service ps nginx_nginx
ID             NAME                IMAGE          NODE                         DESIRED STATE   CURRENT STATE            ERROR
sgv3mq2i8wd6   nginx_nginx.1       nginx:alpine   worker2                      Running         Running 2 minutes ago
ogrm7ybyy70g    \_ nginx_nginx.1   nginx:alpine   manager2                     Shutdown        Shutdown 2 minutes ago
78z0f5wwntiz   nginx_nginx.2       nginx:alpine   worker1                      Running         Running 32 minutes ago
4bkahuh1td3q   nginx_nginx.3       nginx:alpine   worker3                      Running         Running 32 minutes ago
9zjeakzp3z9l   nginx_nginx.4       nginx:alpine   worker3                      Running         Running 2 minutes ago
815x7h4up72o    \_ nginx_nginx.4   nginx:alpine   docker-desktop               Shutdown        Shutdown 3 minutes ago
tntatll5ebzj    \_ nginx_nginx.4   nginx:alpine   docker-desktop               Shutdown        Failed 23 minutes ago    "task: non-zero exit (137)"
mvncqdeojy17   nginx_nginx.5       nginx:alpine   worker1                      Running         Running 17 minutes ago
md6j8flm8oas    \_ nginx_nginx.5   nginx:alpine   nwf6zf4sqixqw94liixwfk2mj    Shutdown        Orphaned 13 minutes ago
ucnectat8iu1   nginx_nginx.6       nginx:alpine   worker1                      Running         Running 2 minutes ago
jivypf0euvkq    \_ nginx_nginx.6   nginx:alpine   manager2                     Shutdown        Shutdown 2 minutes ago
qdmpr2yuw2ig   nginx_nginx.7       nginx:alpine   worker1                      Running         Running 2 minutes ago
hjhioej2qnie    \_ nginx_nginx.7   nginx:alpine   docker-desktop               Shutdown        Shutdown 3 minutes ago
tz6nkxtocqvv    \_ nginx_nginx.7   nginx:alpine   docker-desktop               Shutdown        Failed 23 minutes ago    "task: non-zero exit (137)"
iww05y7z1b91   nginx_nginx.8       nginx:alpine   worker3                      Running         Running 32 minutes ago
b14eu01wqsyo   nginx_nginx.9       nginx:alpine   worker2                      Running         Running 7 minutes ago
bjv7ke0b7s8k   nginx_nginx.10      nginx:alpine   worker2                      Running         Running 7 minutes ago
nyuteb0v816u   nginx_nginx.11      nginx:alpine   worker2                      Running         Running 7 minutes ago
bkvmf3q5csah    \_ nginx_nginx.11  nginx:alpine   nwf6zf4sqixqw94liixwfk2mj    Shutdown        Orphaned 13 minutes ago

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$
```

Note that at any point of time, these nodes can be returned to an `Active` state by executing the following command:

```
docker node update --availability active $LEADER_NODE_ID
docker node update --availability active $REACHABLE_NODE_ID
```

Congratulations!! You have successfully installed Docker and built simple application and pushed to Docker Hub. You have also seen how to use Docker Desktop application and use Docker Compose tool to build multi-container application and deployed to Docker Swarm cluster.