DOCKER OVERVIEW & SETUP

Sri Adilakshmi Marrivada

TABLE OF CONTENTS

1 DOCKER OVERVIEW 4

2 DOCKER CONTAINERS VS VIRTUAL MACHINES 6

3 DOCKER ARCHITECTURE 7

3.1 Docker Host 8
3.2 Docker Engine 8
3.3 Docker Objects 9
3.4 Docker Registry 12
3.5 Docker Compose 12
3.6 Docker Swarm 13

4 DOCKER LINUX VS WINDOWS ARCHITECTURE 13

4.1 Docker Linux Architecture 13
4.2 Docker Windows Architecture 18
4.2.1 Windows Server Containers 22
4.2.2 Hyper-V Container Architecture 23

5 DOCKER INSTALLATION ON WINDOWS 26

5.1 Install Docker 26
5.2 Start Docker Desktop 31
5.3 Verify Docker 34

6 DOCKER COMMANDS 35

6.1 Docker Common Commands 35
6.2 Docker Image Commands 37
6.3 Docker Container Commands 42
6.4 Docker Volume Commands 55
6.5 Docker Network Commands 57
6.6 Docker Registry Commands 60

7 DOCKERFILE 64

8 DOCKER IMAGE FROM DOCKERFILE 70

8.1 Create Dockerfile 70
8.2 Add Instructions 70
8.3 Build Docker Image 71
8.4 Run Docker Image 72

9 DOCKER DESKTOP 73

9.1 LAUNCH DOCKER DESKTOP 74
9.2 CONTAINERS VIEW 76
9.3 IMAGES VIEW 82
9.4 VOLUMES VIEW 86
9.5 BUILDS VIEW 90
9.6 DOCKER HUB VIEW 94
9.7 DOCKER SCOUT VIEW 94
9.8 DOCKER EXTENSIONS VIEW 95
9.9 CHANGE SETTINGS 96

10 CREATE PYTHON DOCKER APPLICATION 98

10.1 VERIFY PYTHON VERSION 98
10.2 LAUNCH VS CODE 99
10.3 CREATE VIRTUAL ENVIRONMENT 101
10.4 CREATE PYTHON FILE 104
10.5 CREATE DOCKER FILE 107
10.6 BUILD DOCKER IMAGE 108
10.7 RUN DOCKER IMAGE 110

11 CREATE HTML DOCKER APPLICATION 113

11.1 LAUNCH VS CODE 113
11.2 CREATE HTML FILE 114
11.3 CREATE DOCKER FILE 116
11.4 BUILD DOCKER IMAGE 117
11.5 RUN DOCKER IMAGE 119
11.6 MODIFY HTML CODE 121

12 PUBLISH DOCKER IMAGE TO DOCKER HUB 124

12.1 CREATE REPOSITORY 124
12.2 TAG IMAGE TO USER REPOSITORY 126
12.3 LOGIN & PUSH USER REPO DOCKER IMAGE 126
12.4 RUN USER REPO DOCKER IMAGE 128

13 DOCKER COMPOSE 131

13.1 DOCKER COMPOSE FILE 131
13.2 DOCKER COMPOSE COMMANDS 136
13.3 CREATE WEB APPLICATION 142
13.3.1 Stop and Delete Containers 142
13.3.2 Launch VS Code 144
13.3.3 Configure MySQL Database 145
13.3.4 Stop and Delete Containers 146
13.3.5 Configure Web App 155

13.3.6 Configure Nginx Server ... *157*
13.3.7 Create Docker Compose File ... *159*
13.3.8 Create Environment File ... *163*
13.3.9 Start Docker Compose ... *163*
13.3.10 Launch Web Application ... *166*
13.3.11 Stop Docker Compose ... *168*
13.3.12 Start Docker Compose Detached ... *170*
13.3.13 Verify Web Application ... *170*
13.3.14 Validate Database ... *171*

14 DOCKER SWARM ... 173

14.1 RAFT CONSENSUS ALGORITHM ... 176
14.2 SERVICES VS TASKS ... 177
14.3 SWARM COMMANDS ... 178
14.3.1 Docker Swarm Commands ... *178*
14.3.2 Docker Node Commands ... *180*
14.3.3 Docker Service Commands ... *181*
14.3.4 Docker Secret Commands ... *183*
14.3.5 Docker Stack Commands ... *183*
14.4 SETUP SWARM USING DOCKER-IN-DOCKER ... 185
14.4.1 Configure Nginx Server ... *185*
14.4.2 Initialize Swarm Cluster ... *187*
14.4.3 Add Manager Node ... *188*
14.4.4 Add Worker Nodes ... *189*
14.4.5 Start Visualizer Service ... *190*
14.4.6 Verify Docker Swarm ... *191*
14.4.7 Promote or Demote Node ... *193*
14.4.8 Configure Nginx Server ... *195*
14.4.9 Test Resilience ... *202*
14.4.10 Drain Manager Nodes ... *209*

1 DOCKER 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 but 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.

The Docker container system utilizes the **operating system virtualization** feature without carrying extra resources of Hyper-V or VMWare enabling Docker containers to run directly within the machine kernel. The isolation and security factors of Docker allows to execute multiple containers simultaneously on the same machine.

Though containerization technology is not something new and has been there for years *(for example, Linux container technologies such as BSD jails, LXC, etc.)*, Docker has gained more popularity due to its salient features:

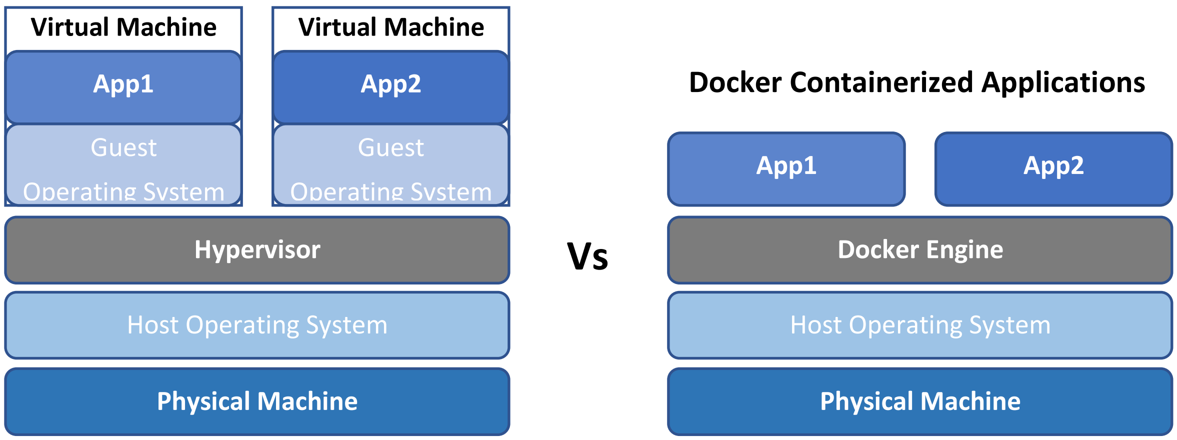
- **Lightweight nature:** Docker containers are light weight and fast since they run on operating system kernel and take up few resources and less space offering great speed and quick startup time.
- **Portable:** Docker containers are isolated from the environment and can run on any environment with different operating systems and hardware platforms i.e. one can create a container in personal laptop (on-premises) and run it in cloud platform (public or private cloud.

- **Consistency:** Since applications with their dependencies are packaged into containers which are isolated from the environment, they work with same behavior and performance across Dev, Test and Production environments
- **Easy to use:** Docker containers are easy to use. Anyone can take the advantage of containers to quickly test applications in their system. Docker allows to pack container in a personal laptop and run it in cloud platform.
- **Time Saving:** Developers can encapsulate the entire run time environment including the application, dependencies, binaries and configuration files into one single container which can be easily deployed across environments in few seconds. Updates or changes made to an application's code are implemented and deployed quickly using Docker containers.
- **Scalability and Modularity:** Docker makes it easy to split the application's functionality into individual small containers. For example, database can run in one container, java script can run in another container and both containers can be linked together to create an application. This helps to easily scale up or scale down or update components independently in the future.
- **Resource Efficiency:** With faster turnaround time of containers, Docker enables concurrent execution of multiple application instances on the same machine making hardware resources are efficiently used.

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

2 DOCKER CONTAINERS VS VIRTUAL MACHINES

Docker containers uses operating system virtualization feature but it is important to understand that **Docker is not a or does not work like a Virtual Machine** (VM).



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. Multiple VMs run simultaneously on a single physical machine. VMs are heavy in nature since they have full OS with its own memory management with the overhead of virtual device drivers.

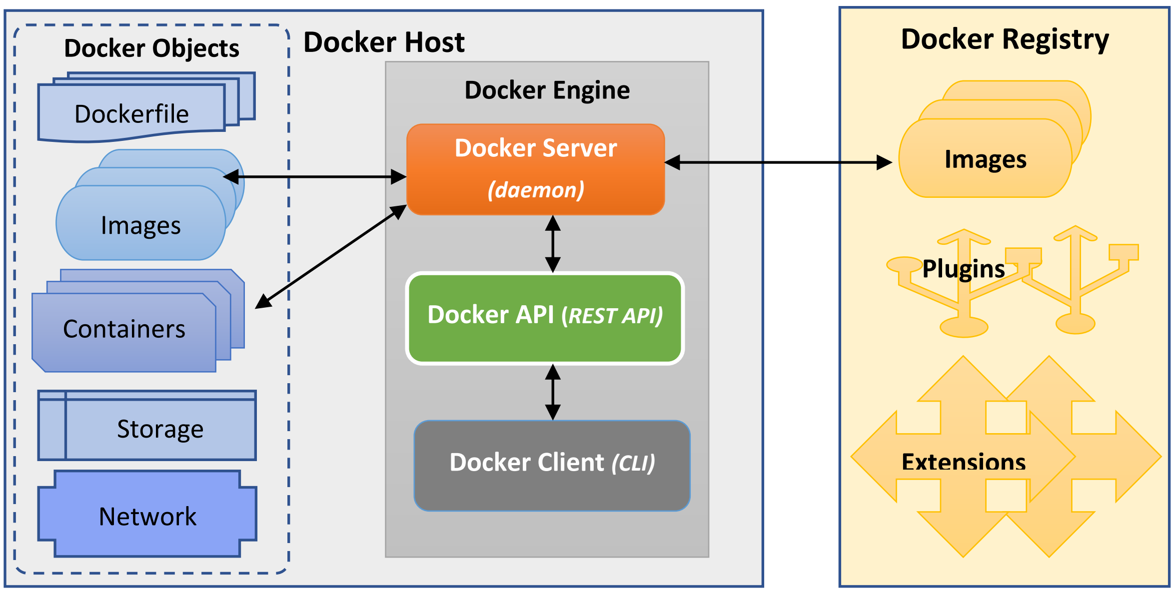
Container is not a VM since it does not require a separate OS and is deployed on a physical machine with a host OS. Container consists of everything needed including application, binaries, libraries for application to work but it depends on the underlying machine OS kernel to run. Containers isolate application environments from one another, and share the underlying OS kernel with other containers. Container uses the resource isolation for CPU and memory, and separate namespaces to separate the application's view of the operating system. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space.

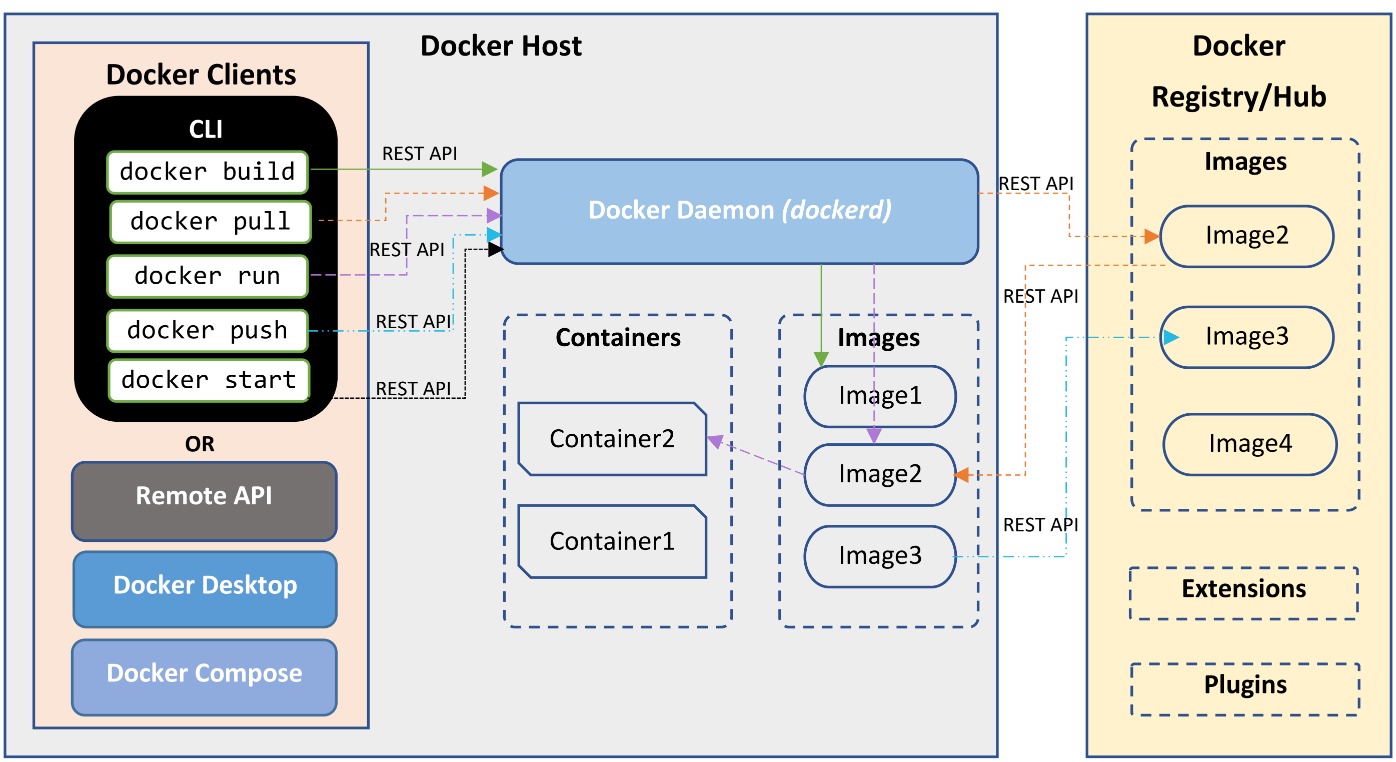
Unlike Virtual Machines which are executed with hypervisor that maintains guest operating systems on top of host OS, Docker containers are executed with Docker engine with in the host OS kernel which makes containers lightweight as they take up less space compared to VMs and have faster start up ensuring improved performance.

3 DOCKER ARCHITECTURE

The Docker platform consists of several components in its architecture in which the main components include:

1. Docker Host
2. Docker Engine
3. Docker Objects
4. Docker Registry





3.1 Docker Host

A Docker host is a physical or virtual machine running either Linux or Windows or MacOS which has Docker installed.

- **Docker in Linux** leverages Linux kernel features such as `namespaces` and `cgroups` to create isolated environments called containers so that multiple applications can run on the same machine without interfering with each other.
- **Docker in Windows** can run Linux containers or Windows containers.
  For running Linux containers on Windows, Docker uses a Windows feature named **WSL2** (Windows Subsystem for Linux 2) that allows to run Linux distributions without VMs enabling Docker to take the advantage of Linux kernel inside WSL2 to create containers.
  For running Windows containers, Docker uses Windows native hypervisor-based virtualization technology named **Hyper-V** to create containers.
- **Docker in MacOS** uses the **HyperKit** which is a hypervisor built on top of the Hypervisor framework in macOS to create containers. For Apple Silicone Macs, Docker has introduced **VMM** (Virtual Machine Manager) which is a container-optimized hypervisor.

3.2 Docker Engine

The Docker engine is the core component of Docker and follows client-server model to create and run containers. It comprises of three components that include **Docker Daemon**, **Docker API** and **Docker Client**. The Docker client communicates with Docker daemon using Docker API. Both Docker client and daemon can run on the same machine or they can run on different machines in which case Docket client will interact with a remote daemon.

1. **Docker Daemon:** The Docker daemon (`dockerd`) is a backend service that runs on the operating system and is responsible for building, running and distributing containers. It listens for **Docker API** requests coming from clients to manage Docker objects such as images, containers, network, volumes, etc. Essentially the Docker daemon serves as the control center for the entire Docker implementation.

2. **Docker API:** Docker API is a REST API service that the Docker client uses to communicate with daemon over UNIX sockets or a network interface.

3. **Docker Client:** Docker provides a client application named `docker` which is a command line interface (CLI) that most users use to interact with Docker. When docker commands are issued, the client access Docker API and sends those commands to `dockerd` to run. Alternatively, developers can use remote REST API, or Docker SDK to communicate with `dockerd`. Docker client can communicate with `dockerd` in three ways:
   a. **By Unix Socket** *(unix://socket_path)* which is the default communication method. The default socket path used by `dockerd` is `/var/run/docker.sock`, which is why only root user can use `docker` after installation in Linux.
   b. **By TCP request** *(tcp://host:port)* which is recommended to configure in production environment.
   c. **By file descriptor** *(fd://)* which is used in `systemd` service.

3.3 Docker Objects

Docker objects are sub-components of a Docker deployment to package and distribute applications. These objects include images, containers, networks, volumes, plugins and others.

1. **Dockerfile:**
   Docker file is a simple text file that uses Domain Specific Language (DSL) with a set of commands or instructions to generate a Docker image. These instructions include specifying the OS, languages, Docker environment variables, file locations, network ports, and other components needed to build the image. Each instruction in a Dockerfile represents a layer of the docker image. Docker daemon runs all instructions from top to bottom in Dockerfile and builds an image. When Dockerfile is updated and image is rebuilt, only those layers that have changed are rebuilt in the image which makes images lightweight, small and fast.

2. **Docker images:**
   A Docker image is a collection of files and other metadata including operating system, application code, libraries, config files and other dependencies bundled together that are needed to run the application within a Docker container. Docker image is a read-only template which is built from set of instructions written in Dockerfile. It acts as a blue print for creating a Docker container. Developers can either create their own images using Dockerfile or download images from Docker Hub and run them.

3. **Docker Containers:**

   Docker containers are the running instances of Docker images. While Docker images are read-only files, containers are live and executable content of images. Each docker container is a self-contained run-time environment that shares the kernel of the host system and separated from all other containers and host system. Docker containers can be created, started, stopped or deleted using Docker commands. Once a container is removed, any changes to its state that are not stored in persistent storage will also be removed.

4. **Docker Storage:**

   By default, Docker containers store their data on a writable container layer using **Storage driver**. This writable container layer sits on top of the read-only image layers. When a container is removed, data stored on the container layer would be lost.

   There are four ways to store data outside the docker container:

   - **Volumes** – Docker volumes *(also called as Named volumes)* are directories that exist on the host file system and mounted to Docker containers for storing data. These volumes are managed by Docker itself and cannot be updated by any non-Docker process. On Linux host, Docker volumes can be found at `/var/lib/docker/volumes` location.
   - **Bind mounts** – Bind mounts *(also called as Bind volumes)* create a direct link between host system path and a container. It allows to mount any directory or file on the host system to the container so that both host system and container can access or modify mounted files simultaneously.
   - **tmpfs mounts** – tmpfs mounts is applicable for Linux host only and stores data directly on Linux host in-memory without writing data on to disk in which case the data in tmpfs will be lost when container is stopped or host system is rebooted. This storage is useful to cache temporary data or store sensitive information.
   - **Named pipes** – Named pipes is applicable for Windows host only and works similar to **tmpfs** where data is stored in-memory and lost when container is stopped or host is rebooted. In Windows, named pipes enable processes to communicate with each other, which can store their data in the container's temporary memory.

   Apart of these, Docker provides **Storage plugins** for Docker container to connect with external storage platforms such as Amazon EBS. Docker also provides an option of **Volume container** which is a dedicated container created to host a data volume which can be mounted to other containers. This volume container works independently from application container and can be shared with other containers.

Overall, in terms of persisted storage, Docker offers four options – **Data Volumes, Volume Containers, Directory Mounts** (nothing but *Bind mounts*) and **Storage plugins**.

5. **Docker Network:**

Docker Network is a virtual network created by Docker to allows containers communicate with each other as well as services located on the same host or on different machines and. Containers have networking enabled by default to make outgoing connections. Containers on the same host communicate each other without the need for ports to be exposed to the host machine.

Docker has 6 types of network drivers to provide networking functionality:

- **bridge**: This is the default network driver for the container that enables communication between containers running on the same host.
- **host**: It enables containers to use the host network itself without any network isolation between host and containers.
- **none**: It disables all networking for a container in which case that container is not accessible from outside of it.
- **overlay**: It enables connection between multiple Docker daemons so that containers can run on different hosts and Docker Swarm services communicate with each other.
- **ipvlan**: It provides full control over IPV4 and IPv6 addressing.
- **macvlan**: It allows to specify MAC addresses instead of IP addresses to the containers.

6. **Docker Plugins:**

Docker plugins are add-ons to enhance the Docker engine functionality by providing advanced storage drivers, networking tools, logging and monitoring mechanisms. Docker plugins are lightweight components which can be enabled or disabled at any time. Docker provides various plugins for **Networking** *(Weave net, Calico, etc.),* **Storage** *(Rex-Ray, Flocker,* *etc.)*, **Logging** *(Fluentd, Syslog, etc.)* and **Monitoring** *(cAdvisor, Prometheus, etc.)*. Docker also allows to load third-party plug-ins into containers.

7. **Docker Extensions:**
   Docker extensions are third-party tools that can be integrated with Docker Desktop to enhance its functionality. Some popular extensions include Docker Scout, Portainer, Lens, etc.

3.4 Docker Registry

A Docker registry is system for storing and sharing Docker images. This registry is split into multiple Docker repositories which can be public or private as per need and hold different versions of images that are maintained through identification tagging. The default Docker registry is the **Docker Hub** from where developers can upload and download images. However, Docker allows to configure a private registry *(which is available locally or on cloud platforms)* as the default one.

Docker Hub is a cloud-based Software-as-a-Service (SaaS) registry provided by Docker that allows users to store and share images. It provides both public and private repositories for maintaining the docker images as per the choice to whether the image should be publicly exposed or not. It is a world's largest library and holds over 100,000 container images including those images produced by Docker, Inc., certified images belonging to the Docker Trusted Registry, images sourced from commercial software vendors, open source projects and individual developers.

When a docker command such as `docker pull` or `docker run` is used, Docker by default searches and pulls the image (if available) in Docker Hub. Developer can maintain a private Docker registry or use any cloud-based Docker registry *(such as Azure Container Registry)* and configure that as a default registry to pull or push images.

3.5 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. This tool is used to configure multi-container application services, view container statuses, stream log output and run single-instance processes.

3.6 Docker Swarm

Docker Swarm is another component of Docker that supports cluster load balancing for Docker. It is a container orchestration tool that allows to manage and deploy Docker applications across multiple Docker hosts as a single virtual system. Multiple Docker hosts are joined together in Swarm to act as one, which lets users quickly scale container deployments to multiple hosts.

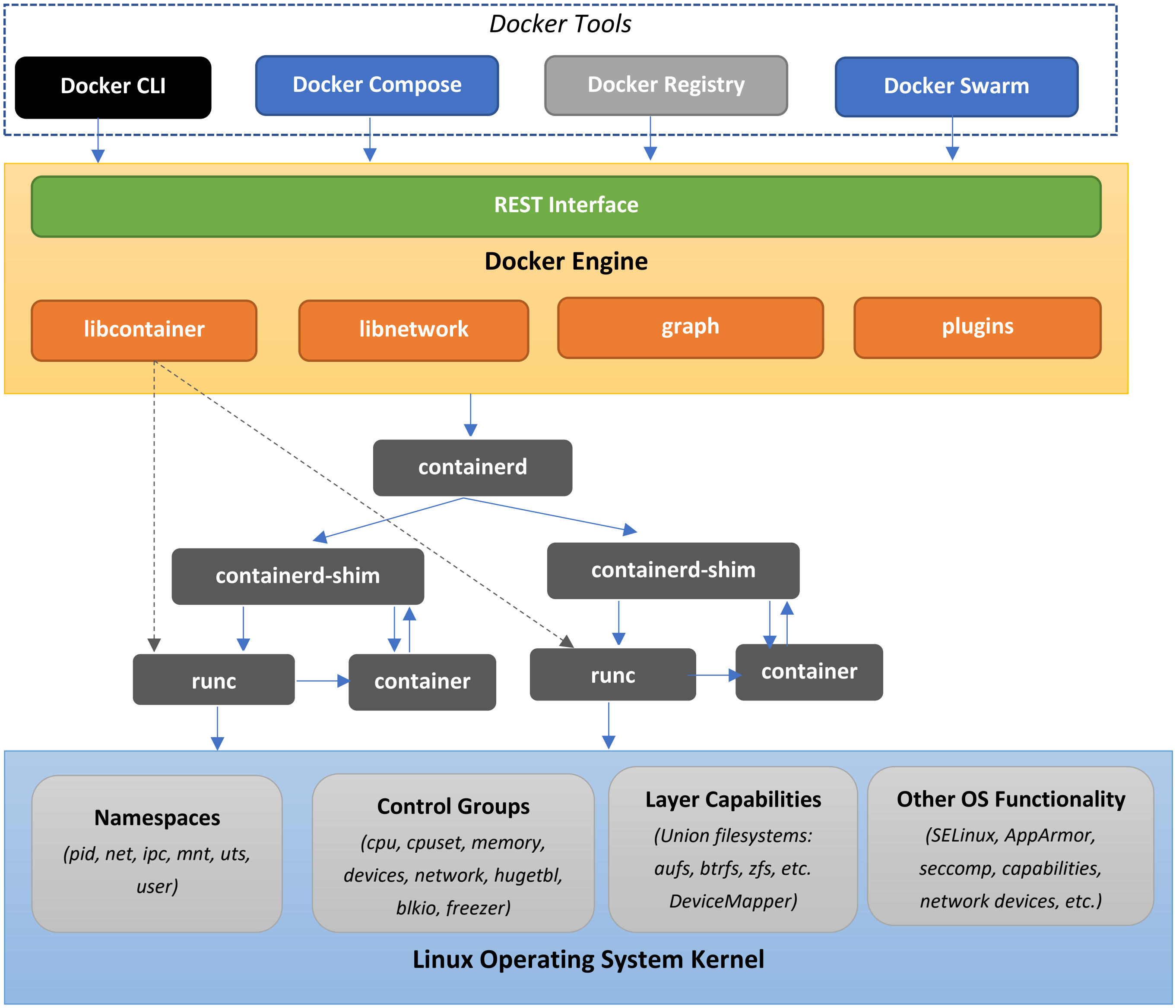
4 DOCKER LINUX VS WINDOWS ARCHITECTURE

Docker is natively supported on Linux operating system but from 2016, Docker has introduced the native support for Windows as well in collaboration with Microsoft. With Microsoft Windows Server 2016 and Windows 10 operating system, Microsoft introduced the new feature called **Windows Containers.** Docker supports running windows containers natively on Windows 10, Windows 2016 and Windows 2019 operating systems only.

4.1 Docker Linux Architecture

Docker is developed in **Go** programming language and natively supported on **Linux** operating system. Docker takes the advantage of several underlying features of Linux kernel to deliver its functionality including creation of containers which mainly need kernel features such as *namespaces*, *control groups* and *union file system*.

Linux container is nothing but a Linux application that runs in an isolated Linux environment. The same container can be run on a Windows OS using virtualization to emulate a Linux environment, but the container is still running on Linux.



In the Docker Linux architecture, all Docker tools such as **Docker CLI, Docker Compose, Docker Registry, Docker Swarm**, etc. talks to the **Docker Engine** (**dockerd,** a long-lived daemon) using **REST** based API calls. The Docker engine in turn communicates with **containerd** *(long-lived daemon)* which is a high-level container management tool responsible for managing the container lifecycle including creating, stopping, starting and deleting containers by using **runc** as the default container runtime. **Containerd-shim** is a light-weight daemon that launches runc. Once runc launches the container, it exits and then containerd-shim takes over and becomes the parent process for the container process, enabling containerd to monitor and control the container's lifecycle. **Runc** is a light-weight command-line tool for **libcontainer** library *(developed by Docker in Go language and part of Docker Engine)* used to access the Linux Kernel for spawning and running containers according to the Open Container Initiative (OCI) runtime specification. When the containerd *(high-level container runtime or container manager)* sends the container command to the runc *(low-level container runtime)* via containerd-shim, runc makes a corresponding system call to the underlying Linux kernel. For example, when a command to create a container is issued, runc first makes the `clone()` system call to the kernel and this system call is responsible for creating a Linux namespace and then it makes other system calls that are required by process inside the container.

Libcontainerd talks to containerd and run which uses runc to launch containers.
In the underlying architecture, Docker leverages several Linux kernel features for containers:

- **Namespaces:** Kernel namespace provides the isolated workspace called container. When a container is run, Docker creates a set of `namespaces` for each container and these namespaces provide additional layer of isolation. Each aspect of container operates within its dedicated namespace and access of that aspect is limited to that namespace.
  Docker uses the following namespaces:
    - **PID namespace** isolates the process id (PID) numbers so that processes inside a container can only see and interact with other processes inside the same container. Each container gets its own PID namespace, meaning processes in different containers can have the same PID, but they are isolated from each other.
    - **NET namespace** isolates network interfaces, IP addresses, routing tables and port numbers. It creates network isolation between containers and the host or between containers themselves (using bridge, overlay, or other network drivers).
    - **IPC namespace** isolates the system's inter-process communication (IPC) resources such as shared memory and semaphores so that processes inside a container cannot access or interfere with IPC objects outside the container
    - **MNT namespace** isolates the file system mount points so each container can have its own root file system and mount points. This ensures that containers do not share or interfere with the host file system unless explicitly allowed (for example, through Docker volumes).
    - **UTS namespace** (Unix Time-sharing System) isolates the hostname and domain name of the container. Containers can have their own independent hostname, separate from the host or other containers.

    - **User namespace** provides user ID and group ID isolation. It allows containers to have a different user ID and group ID (UID/GID) mapping from the host so that a layer of security is provided by allowing the container's root user to map to a non-root user on the host, reducing the risk of privilege escalation.

- **Control groups (Cgroups):** Docker uses kernel control groups for resource allocation and isolation. An application can only use the resources that are designated by a cgroup. Control groups enable the Docker Engine to share hardware resources with containers and enforce limits and constraints.

  Docker Engine uses the following cgroups:
  - **CPU cgroup** to limit and control the CPU usage of containers.
  - **CPUSet cgroup** to restrict containers to specific CPUs or CPU cores and control how containers use CPU resources. It is useful for real time applications and NUMA systems with localized memory per CPU.
  - **Memory cgroup** to limit how much memory a container can use. Docker can set hard and soft limits on memory usage, control swap usage for containers, manage container termination if they exceed their memory limits using an Out-of-Memory (OOM) killer.
  - **HugeTBL cgroup** to limit how many huge pages (large memory pages) a container can consume.
  - **BlkIO (Block IO) cgroup** to control the block I/O (disk read/write) access for containers.
  - **Freezer cgroup** to freeze or suspend all processes in a container without killing them, effectively pausing the container's operation. This is useful for cluster batch scheduling, process migration, debugging and resource management purposes.
  - **Devices cgroup** to control whether a container can access devices such as block storage, character devices, or others.
  - **Network cgroups** *(net_cls, net_prio)* to classify network traffic and manage the network bandwidth between containers. cgroup for tagging the traffic control.

- **Union File Systems (Union FS)**: Docker commonly uses a Union filesystem called **OverlayFS** that enables the files and directories of separate filesystems, known as layers, to be transparently overlaid on top of each other to create a new virtual, layered filesystem. When Docker builds a container image, OverlayFS creates a layered file system that consists of a series of read-only layers attached to an image and a single writable layer that is used by the container's runtime environment.

Docker Engine can use multiple UnionFS variants including:

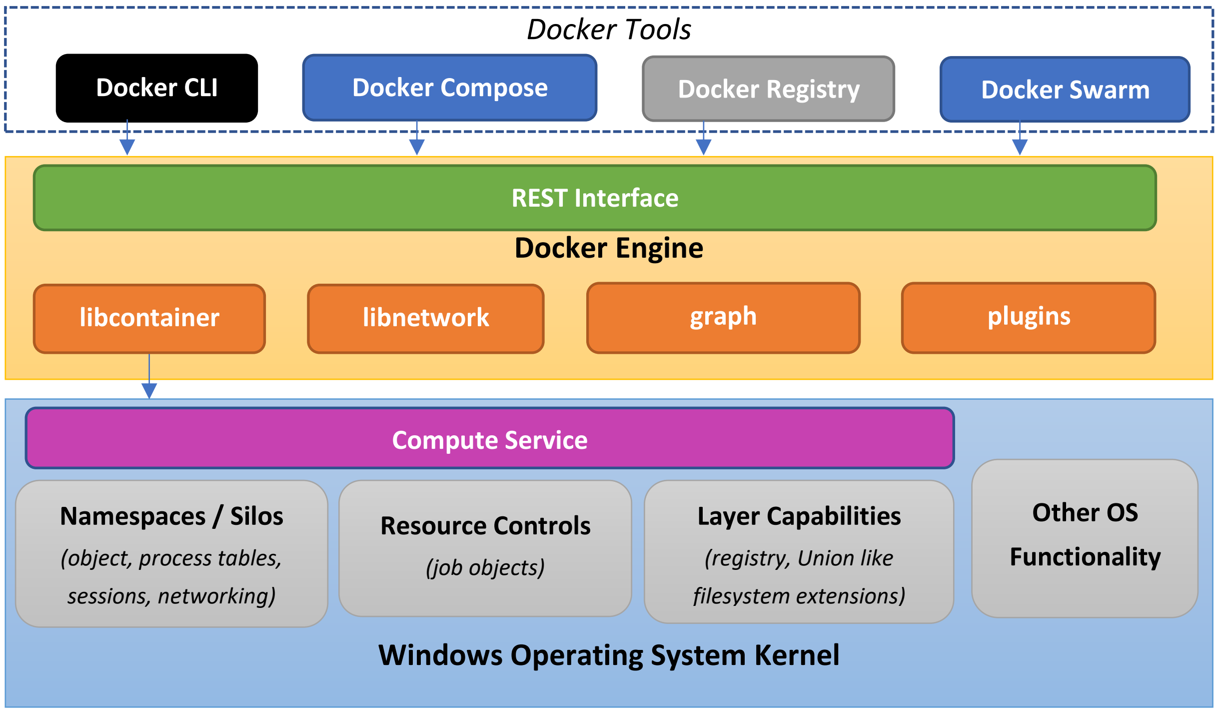
- **aufs** – Advanced multi-layered Unification Filesystem (aufs) was used by Docker as default UnionFS in previous versions and replaced with OverlayFS in latest versions.
- **btrfs** – B-Tree Filesystem
- **vfs –** Virtual Filesystem
- **zfs** – Zettabyte Filesystem
- **DeviceMapper**

- **Security:** Docker makes use of **AppArmor**, **Seccomp**, **Capabilities** kernel features for security purposes.
  - **AppArmor** enforces security policies on containers to restrict their capabilities and actions. . It works by applying profiles that define what resources (e.g., files, network, system calls) a container can access, limiting the potential damage in case of a compromise.
  - **Seccomp** (Secure Computing) restricts the system call issues by a program. It works by filtering system calls, allowing only a predefined set of calls to pass through, effectively sandboxing the process.
  - **Capabilities** are fine-grained permissions that control what specific privileged actions a container can perform, without giving it full root access. Linux capabilities break down the root user's all-powerful privileges into smaller, manageable units.

Namespace, *CGroup*, and *UnionFS* are the basic building blocks of a container. Docker Engine combines the *namespaces*, *control groups* and *union filesystem* into a wrapper called a container format. The default container format is `libcontainer`.

4.2 Docker Windows Architecture

In Windows, the Docker architecture looks somewhat different at host level in comparison to Linux.



Docker architecture in Windows looks same for the top-level components such as Docker CLI, Compose, Registry, Swarm etc. which talk to Docker engine though REST API as in Linux but the host level architecture is different in Windows. The kernel mode in Windows includes not only kernel but also HAL (`hal.dll`) and other system services. It also includes various managers for objects processes, memory, security, cache, plug-in-play (PnP), power, configuration and I/O that are collectively called as Windows Executive (`ntoskrnl.exe`). Windows kernel does not have features such as `namespaces` and `control groups` for which, Microsoft has introduced **Compute Service Layer** at OS level which provides namespaces, resource controls and union filesystem capabilities. Compute Service Layer acts as a public interface (via *`vmcompute.dll`*) for containers and manages containers like starting, stopping, deleting, etc. but does not maintain any state as such. This layer replaces *`containerd`* and *`runc`* components on Windows and abstracts low-level capabilities that the kernel provides. Compute Service Layer also provides two language bindings – HCSShim *(library written in Go and used* *in containerd and moby)* and [dotnet-computevirtualization] *(library written in C# .NET)* so that external applications other than Docker can plugin as needed.

In the underlying architecture, Docker leverages the following Windows kernel features for containers:

- **Job Objects:**
  Job object is a kernel object that allows to manage a group of processes as a single unit. Any modifications *(example, enforcing limits such as working set size and process priority or terminating all processes associated with a job)* performed on a job object affect all processes associated the job object. Jobs also record accounting information such as IO counters, page fault count, total processes, total terminated processes, etc.. for every process that has been part of a job, even the ones that have been killed. An application consists of one or more processes. In Windows, processes consist of resources that are used when an instance of a program is executed and each process consists of unique process ID, private virtual address, an executable program, threads, collection of open handles, security, context, copy of environment variables, etc. A job can be created using `CreateJobObject` function of the `Win32` API.

- **Resource Controls:**
  Resource Controls work similar to C-groups in Linux. Resource controls in Windows allow administrators to manage and allocate system resources like CPU, memory, and disk space to processes, applications, or users. Various types of resource controls include *CPU/Processor controls, Memory/RAM Controls, Disk/Storage Controls*, and *Network/Throughput Controls*. Windows containers use job objects to group and track processes associated with each container. Resource controls are implemented on the parent job object, which ensures that processes within the container are subject to the defined limits.

- **Silos** *(namespaces)***:**
  Windows do not have *namespaces* term which is in Linux. Instead, Windows has a concept called **Silos** which are an extension to the existing **Job Objects** feature that represents a set of processes that can be managed as a single unit and a set resource controls that can be imposed via limits such as maximum working set memory, maximum CPU usage, etc. Silos

include existing Job Object capabilities and a new set of virtualized resources that can be termed as namespaces.

Each process running in a Silo has a separate copy or isolated view of the following resources (or namespaces):

- **Object namespace** is a system level namespace hidden from users. The Windows kernel-mode object manager component manages all objects including files, devices, synchronization primitives, registry keys, jobs, etc. and stores objects in a virtual directory system, also known as the *object namespace*. Just like Linux, Windows also has **** (slash root) at NT level for all devices, example `C:\Windows` actually maps to `\DosDevices\C:\Windows` at kernel level. Object namespace contains all device entry points such as `\DosDevices\C:`, `\Registry`, `\Device\Tcp` (used in case of networking), `\Silos`, etc. Windows maintains **chroot** mechanism (as in Linux) to have per container namespace so that when a container is started, it has its own object namespace under `\Silos` directory, example `C:\` in a container maps to `\Silos\foo\DosDevices\C:`.
- **Process table** represents the list of processes running on the Windows system. Each container has its isolated view of the process table that lists processes running inside a container. Run `Get-Process` command in a PowerShell to access the process table.
- **Sessions** which host a collection of processes where each process belongs to exactly one Session and each Session has a Session ID which identifies it. Each container has its own view of the sessions opened inside a container and these are isolated from the host Windows system sessions.

- **File system:**
  Since Windows NTFS (New Technology File System) is huge and all Windows applications are dependent on NTFS semantics such as transactions, file IDs, USN Journal, object IDs, etc., it is difficult to build a full union file system for Windows containers. Instead, Windows implemented a hybrid model of having virtual block device with a NTFS partition per container where the files are symlinked to the host file system to keep block devices small. When a container is started, it will have small NTFS partition with a file metadata having symlinks. Files that are updated inside a container get stored in its virtual block device.

- **Registry:**
  Windows Registry is a simple file system. Microsoft built a true union file system for registry. It saves cloning a full set of registry hives per container. When a docker diff command is used to see changed files, registry hive shows up there as a special case.

- **Networking**:
  Networking in Windows containers works similar to VM networking. Each container has a virtual network adapter (**vNIC**) which is connected to a Hyper-V virtual switch (**vSwitch**). Windows supports five different network drivers such *as nat, overlay, transparent, l2bridge,* and *l2tunnel*. By default, containers running on Windows uses *nat* network which uses an internal vSwitch and a Windows component named **WinNAT**.
  Windows has 2 types of vSwitches – **Internal vSwitch** (that is not directly connected to a network adapter on the container host) and **external vSwitch** (that is directly connected to a network adapter on the container host).

Starting Windows Server 2016 and Windows 10 natively supports Docker containers and offers two container types:

- **Windows Server Containers (WSC)** – These containers share the Windows OS kernel and run in a similar way to Linux containers.
- **Hyper-V Containers** – These containers can use the same images as WSC but run in a very thin hypervisor layer called Hyper-V VM so that each container has its own kernel.

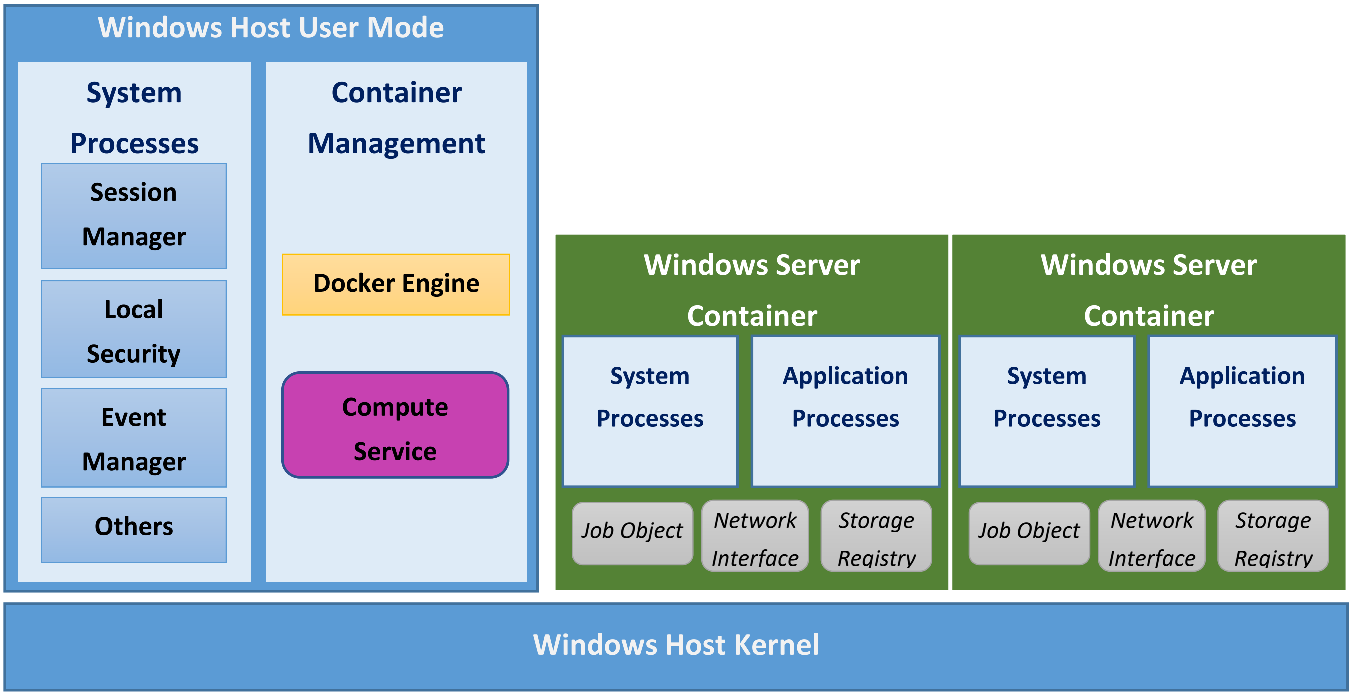
From Windows Server 2022, there is an introduction of new type of containers called **Host Process Containers** which work similar to Windows Server Containers except that containers can be created in the host's network namespace instead of their own. These containers are specially designed to have better support for Kubernetes cluster management. Refer to Microsoft Tech Community on more details about this third type container.

Windows provide three isolation modes (`process, hyperv, hostprocess`) to run three types of Windows containers and switching between isolation modes is easy by passing –`-isolation` parameter to the `docker` tool.

Microsoft provides a Windows Container Version Compatibility matrix available to know exactly what versions of which containers work on which host OS and under what isolation modes.

4.2.1 WINDOWS SERVER CONTAINERS

Windows Server Containers (WSC) are also called as Process Containers.



At the bottom of the Windows Server Container (WSC) architecture, there is a shared kernel just like in Linux. On the left side, there is a Host **User Mode** in which, some **System Processes** such as *Session manager, Local security, Event manager, etc.* and **Container Management processes** such as *Docker Engine* and *Compute Service* are running on the Windows host system. **User mode** is one feature that Windows provides for applications to operate. In Windows, any process operates in two different modes – `user mode` *(all applications operate in this mode)* and `kernel mode` *(core operating system components function in this mode).* The most important components are on the right side of the WSC architecture where there are **System Processes** and **Application Processes** which work differently in Windows compared to Linux. In Linux, applications can communicate to kernel using system calls which is documented and guaranteed to be stable across different kernel versions but Windows is a highly integrated operating system that exposes its API through DLLs (Dynamic Linked Library) but not by system calls. Any kernel action that the application wants to perform either on host system or inside the container should communicate with the ntdll.dll that talks to Windows manager which in turn performs the kernel action. The internal process of how DLLs actually interact with the OS is not documented though, but tightly coupled with the OS services that are running, which in turn have their own coupling with other services and DLLs which means though the kernel is shared, the application cannot be completely isolated from system services and DLLs. For this reason, each WSC has application running along with few critical system services and DLLs that are required to make Windows API calls by the application to connect to the Windows system. This makes WSC images huge in size. Due to the heavy dependency on system processes, WSC container will run properly on a Windows host which has the same operating system version as that of WSC. That is, Windows container built with Windows 10 cannot run on a Windows 2016 server or Windows 2019 server.

When a WSC container is started just to launch one application, it first starts `smss` process *(equivalent of init process)* which launches several system services and then starts the application which is why there no `FROM` scratch in a `Dockerfile` for Windows.

It is not possible to run Linux-based containers natively on Windows because containers can talk to Windows kernel through DLLs only. In order for containers to run on Windows, they need to be based from the same Windows image version that is running on host.
Microsoft has two base images available on the Docker Hub:

- **microsoft/windowsservercore** – It is basically a full Windows server that is taken and made into container image which is huge in size of almost 9.3 GB, but fully compatible with .Net 4.5 and supports all existing Windows applications.
- **microsoft/nanoserver** – It is very smaller in size of ~600MB with small API surface and missing graphic stack and other system dependencies like powershell, WMI, etc due to which the existing application might not be compatible but it has faster startup and occupies less memory.

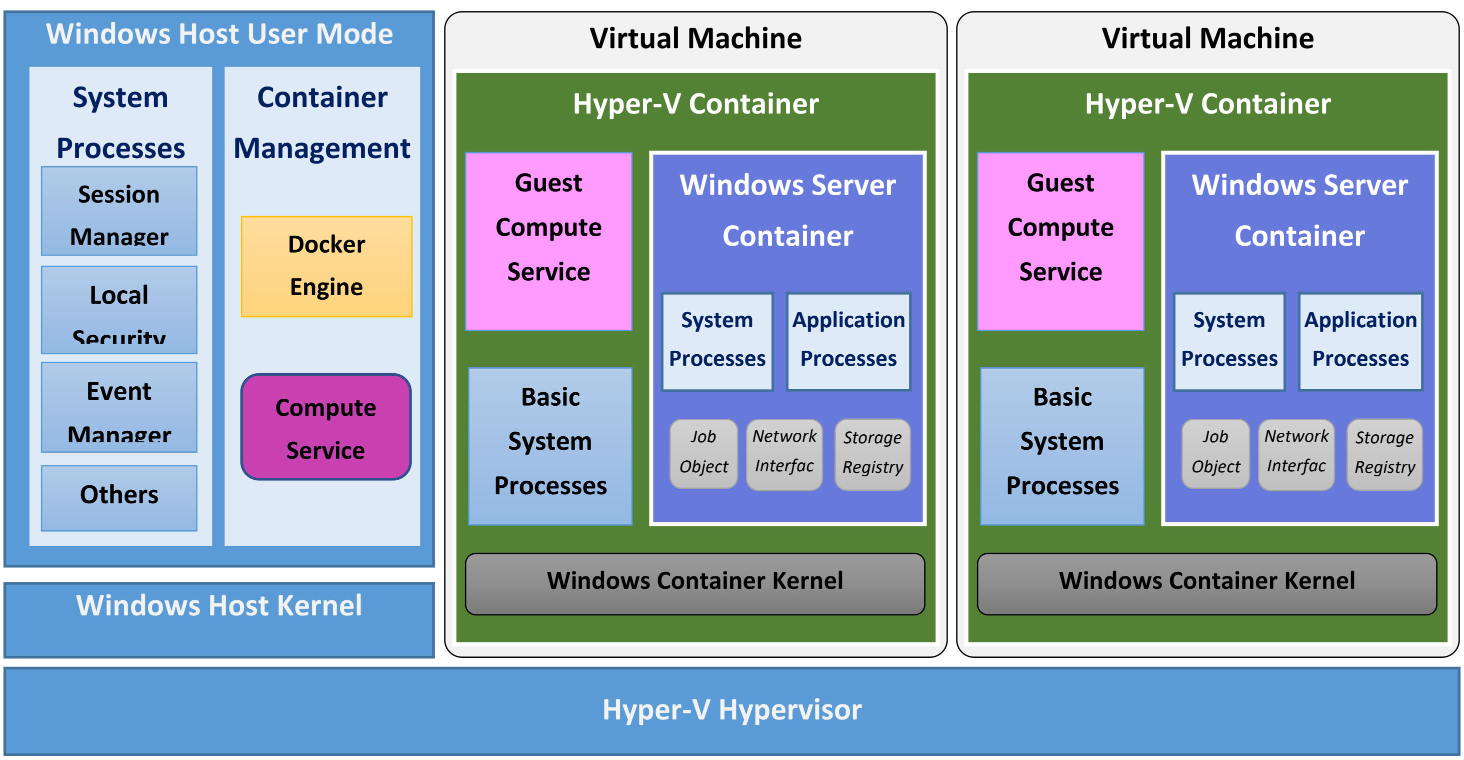
Along with the above two, Microsoft has additionally introduced two more base images **microsoft/windows** and **microsoft/windowsserver** with different API sets.

4.2.2 HYPER-V CONTAINER ARCHITECTURE

In Windows Server Containers or Windows Process Containers, there is a limitation that the container OS and host OS version must match for container to be able to communicate with host OS kernel otherwise application running inside a container can break. As an alternative to this, Hyper-V container can be used to keep containers totally isolated from the host.

Hyper-V is a hypervisor-based virtualization technology for certain x64 versions of Windows. The hypervisor is the processor-specific virtualization platform that allows multiple isolated operating systems to share a single hardware platform.

In Hyper-V containers, there is no shared kernel concept which means there is no need to match Windows build versions between Windows Host OS and the Container. Each Hyper-V container runs with in its own Hyper-V utility VM but it is not a typical VM that has to be managed or state associated and it is invisible to Docker which just takes a simple flag (`docker run --isolation=hyperv`) to run the container inside a VM. Hyper-V utility VMs are different than normal VMs in such a way that Hyper-V VM is just a read-only version of a VM so writes are not persisted. It also does not need to have an own hostname because the hosted containers have their own hostname and identity.
The setup of each Hyper-V VM is that it contains its own clone of the Windows Server kernel and hosts a separate **Guest Compute Service** for communication with the **Compute Service** abstraction of the **Host User Mode**.



In the Hyper-V Container architecture, the left portion remains same as in WSC where there is a **Host User Mode** in which some **System Processes** such as *Session manager, Local security,* *Event manager, etc.* and **Container Management processes** such as *Docker Engine* and *Compute Service* are running on the Windows host system. On the right side how, containers run inside VM is different. First Hyper-V containers use the base image defined for the application and automatically create a Hyper-V VM using that base image so that a small Windows OS is available with necessary **System Processes**, **Compute Service** API and **Kernel configuration** inside a VM. Then a Windows Server Container is created with application and necessary Windows system processes (DLLs) which enables application to communicate using DLLs with the **Guest Compute Service** which internally communicates with **Container Kernel**.

Windows container running inside a Hyper-V VM provides the complete kernel isolation and separation of the host patch/version level from that is used by the application unlike running Windows container directly on the host. Multiple Hyper-V containers can run on the same Windows host and all those containers can use the common base image and no management is required in creating the Hyper-V VM as it is taken care automatically.

Hyper-V container is a stateless i.e. when a container is stopped, VM related data which is stored on registry will be deleted. Hyper-V containers can access files on the host file system using **VMBus** over **SMB** (Server Message Block) protocol which is a Windows file sharing protocol. VMBus is a virtual communication bus that facilitates bi-directional communication between the guest virtual machine (VM) and the host, enabling the guest VM to interact with the host's resources and services. The networking inside a VM works using **Virtual NIC** (Virtual Network Interface Card) that acts as a virtualized representation of a physical network adapter, allowing the VM to connect to a network.

To make the startup time of Hyper-V containers faster, Microsoft has developed the cloning concept for utility VMs. Cloning technology works by forking *(creating an identical copy)* the VM state just before a Windows Container is created in the utility VM. When a Hyper-V container is started for the first time on a host, a utility VM is booted with few necessary processes and its memory state is frozen by pausing the virtual processes in that VM which keeps running and then a quick snapshot and copy of VM state is made and Windows Container is started in that copy. Next time a Hyper-V container is started, it starts from the snapshot made rather than rebooting the utility VM again.

On a Windows 10 host machine, Hyper-V containers are created by default since Windows Server Containers cannot be run directly due to the difference in OS versions between host and container.

Note:
To run Windows containers, one must have operating system with Windows 10 or Windows 11 Professional or Enterprise edition. Windows Home or Education editions only allow to run Linux containers using WSL2.

5 DOCKER INSTALLATION ON WINDOWS

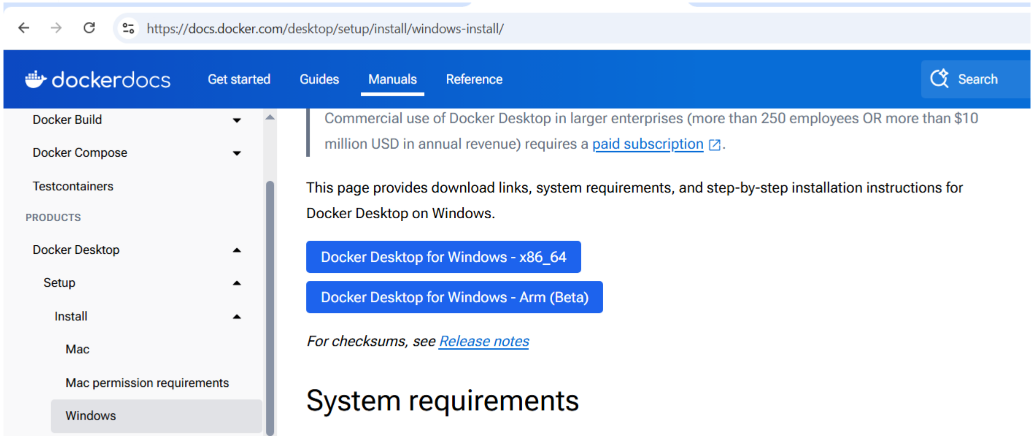
On a Linux host, Docker is installed directly on the operating system but on Windows, Docker is installed inside a Linux VM *(created using either Hyper-V or WSL2)* and Docker client is used to interact with Docker Engine.

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`

5.1 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 that `Docker Desktop Installer.exe` file installs by default 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"
```

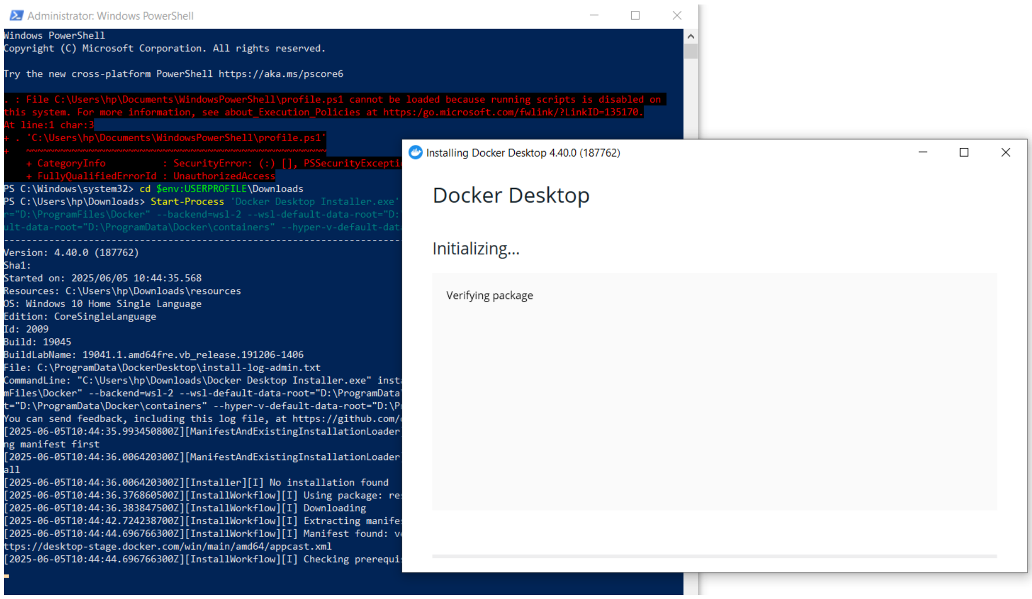
To run from **PowerShell** application in **Administrator** mode, use the following commands for Docker installation:

```
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"'
```

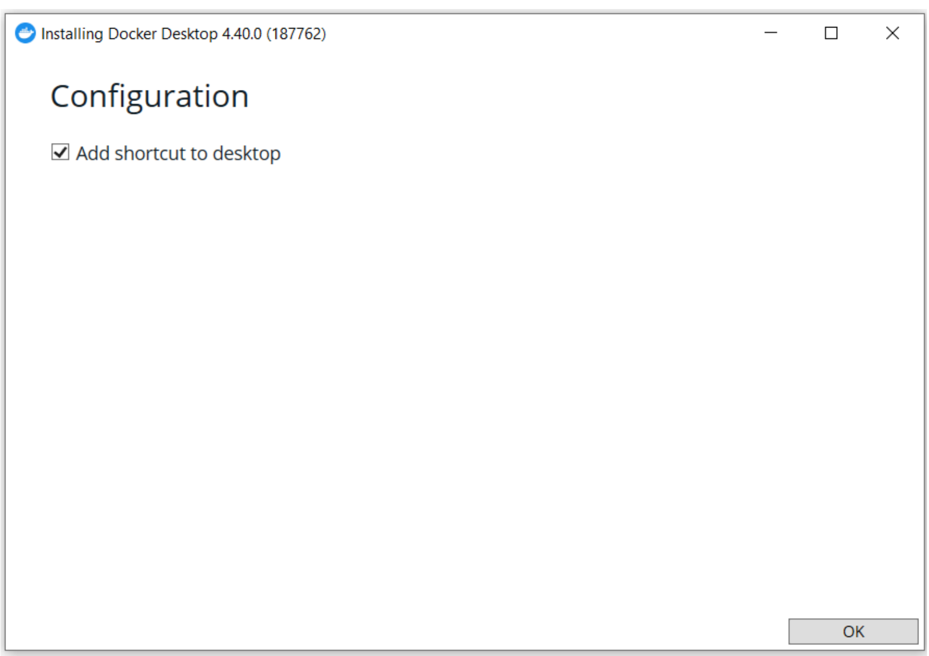
Note that in the above commands, the installation directory is specified as `D:\ProgramFiles\Docker` to install at this location and backend as `wsl-2` to use *(even if*

`--backend` flag is not specified, it uses `wsl-2` by default but `hyper-v` or `windows` values can also be specified to use a different backend).

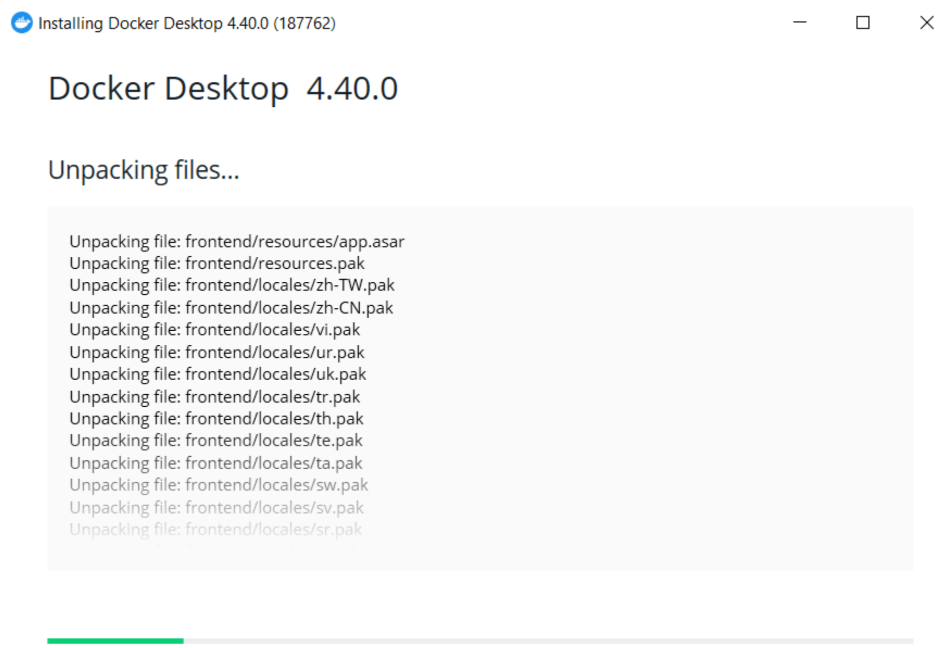
The installer then starts extracting and checks prerequisites which opens the installer wizard



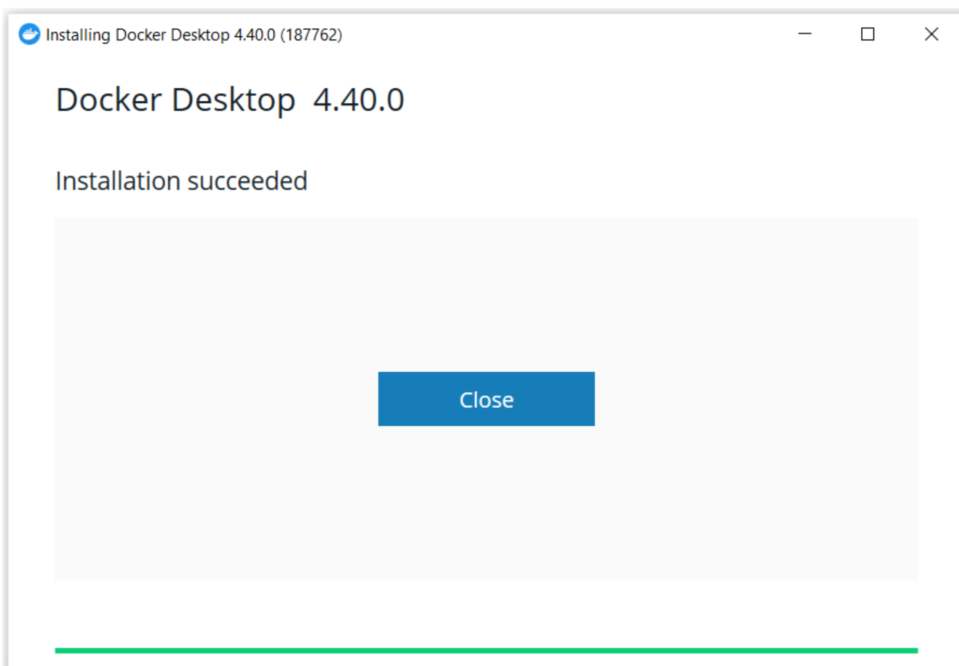
Once all prerequisites are validated, it shows the **Configuration** page where select the **Add shortcut to desktop** checkbox and press **OK** button.



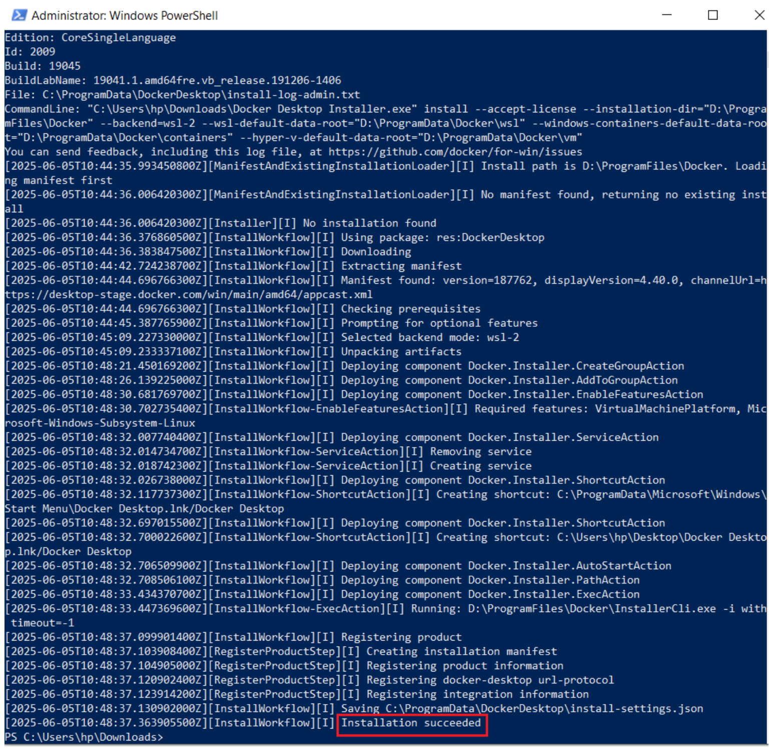
Then, it starts with the installation by unpacking files and takes few minutes to complete.



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

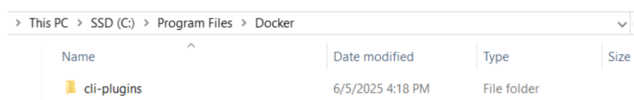


On the **Command Prompt**, we will see **Installation Succeeded** message.

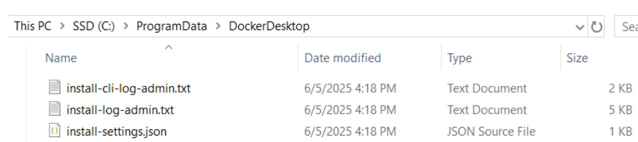


Note that though we installed Docker at `D:\ProgramFiles\Docker` location, some files are still being referenced from the default locations which cannot be avoided.

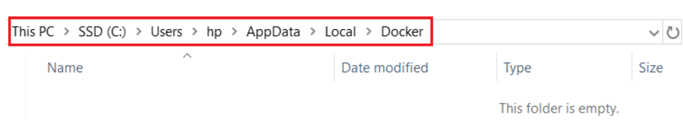
`C:\Program Files\Docker`



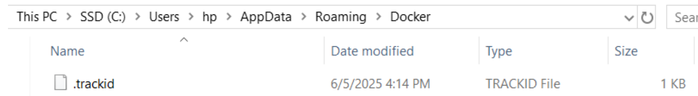
`C:\ProgramData\DockerDesktop`



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



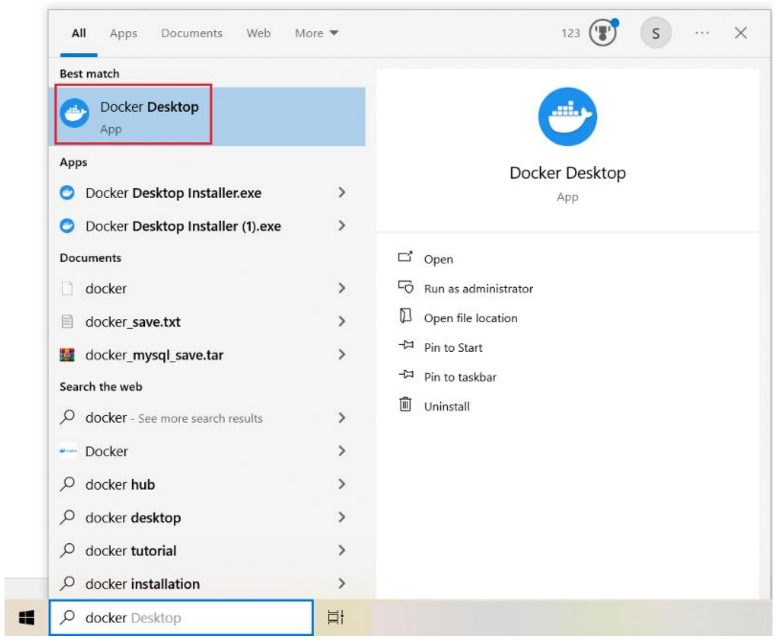
```
C:\Users\<username>\AppData\Roaming\Docker
```



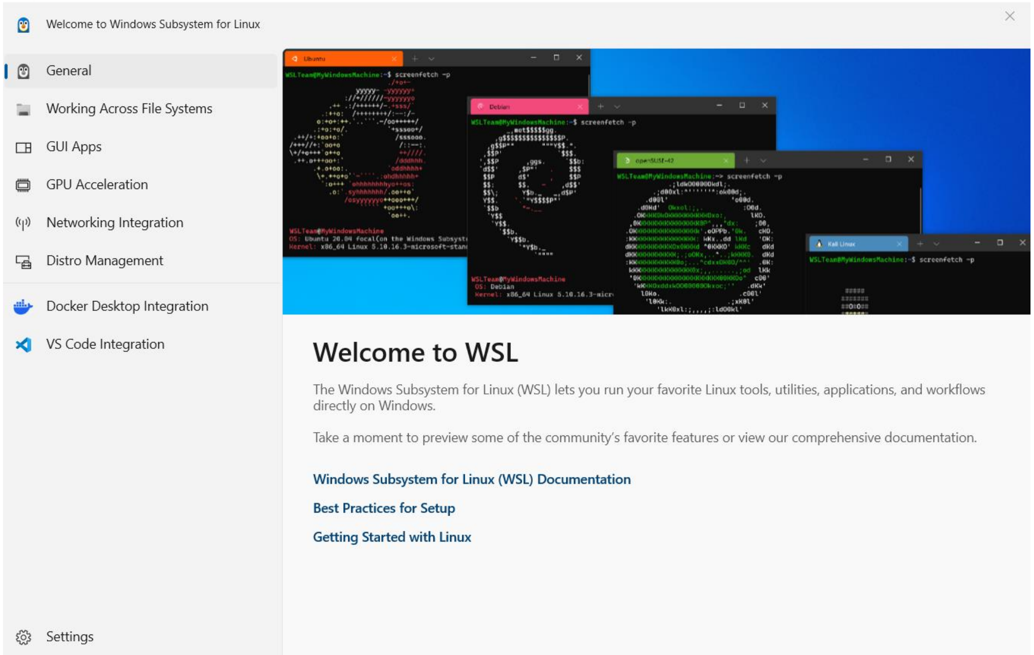
5.2 Start Docker Desktop

After Docker installation, Docker Engine does not start automatically. Launch the **Docker Desktop** application which starts the Docker Engine.

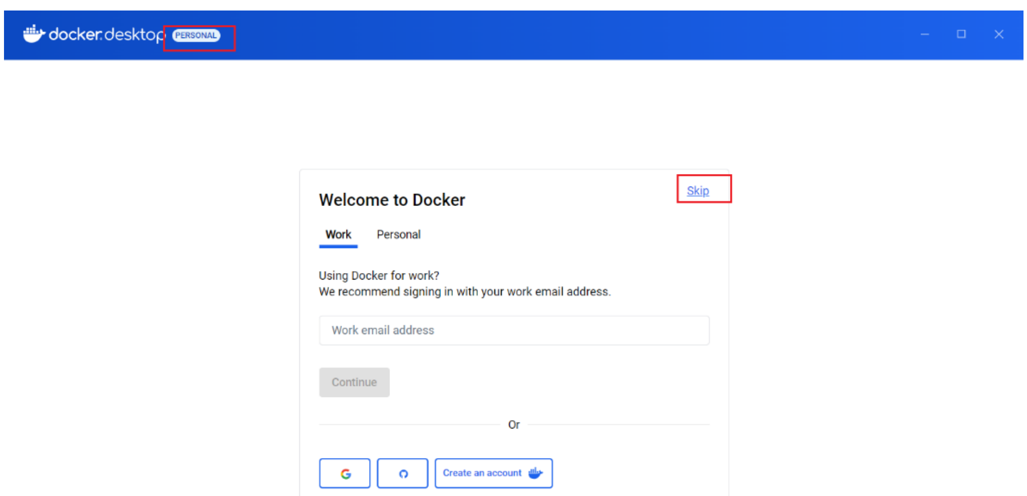
- Search for **Docker** in the search box of the task bar and select **Docker Desktop** application.



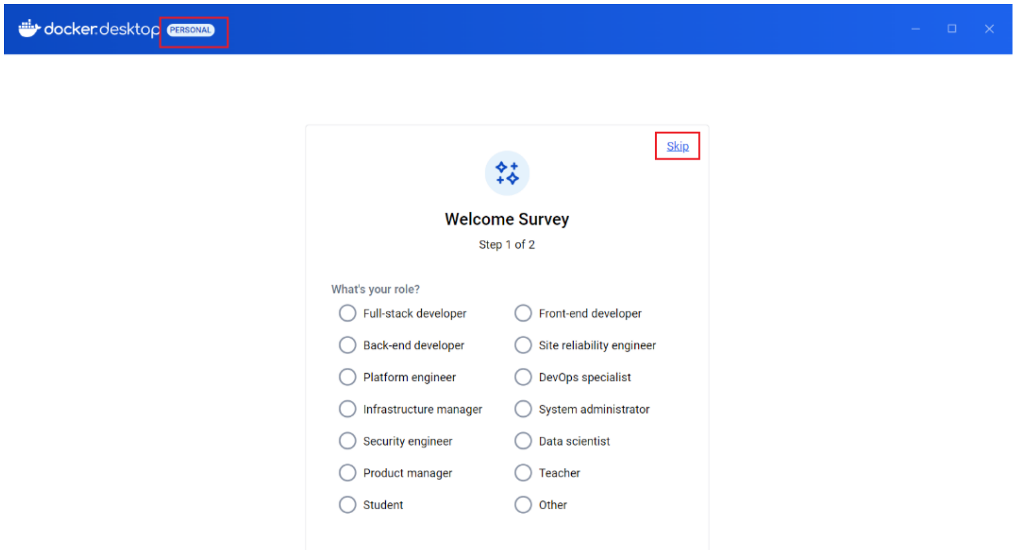
- In few seconds, it might launch **Windows Subsystem for Linux** (WSL) app since Docker uses this at the backend. This can be closed.



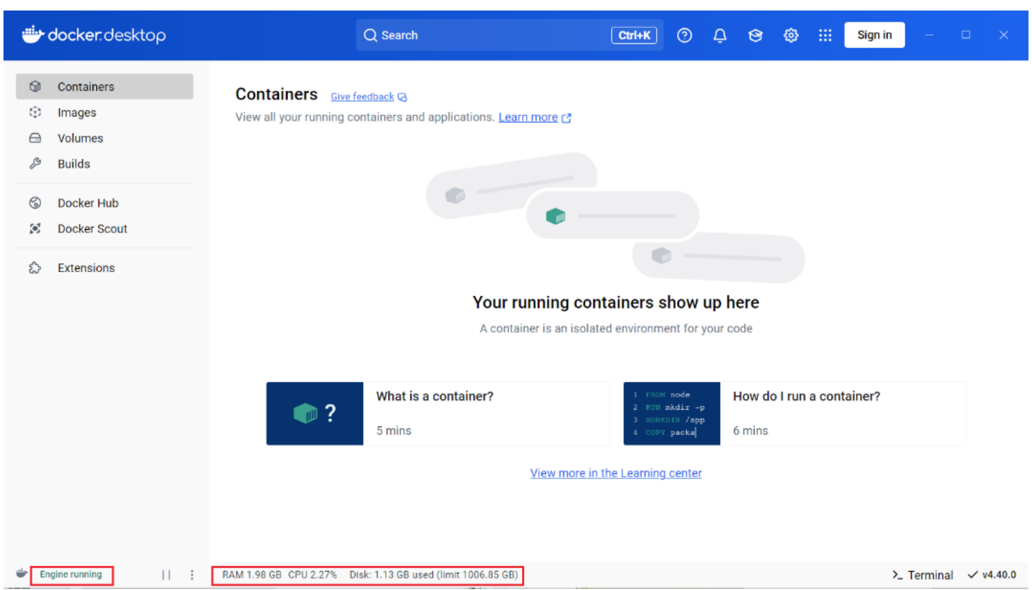
- In few seconds, it launches **Docker Desktop** application with a **Welcome** page. At the top, it displays the Docker Subscription Service Agreement. Since we are using Docker for personal use, it displays **PERSONAL** subscription. Click on **Skip** link to skip it for now.



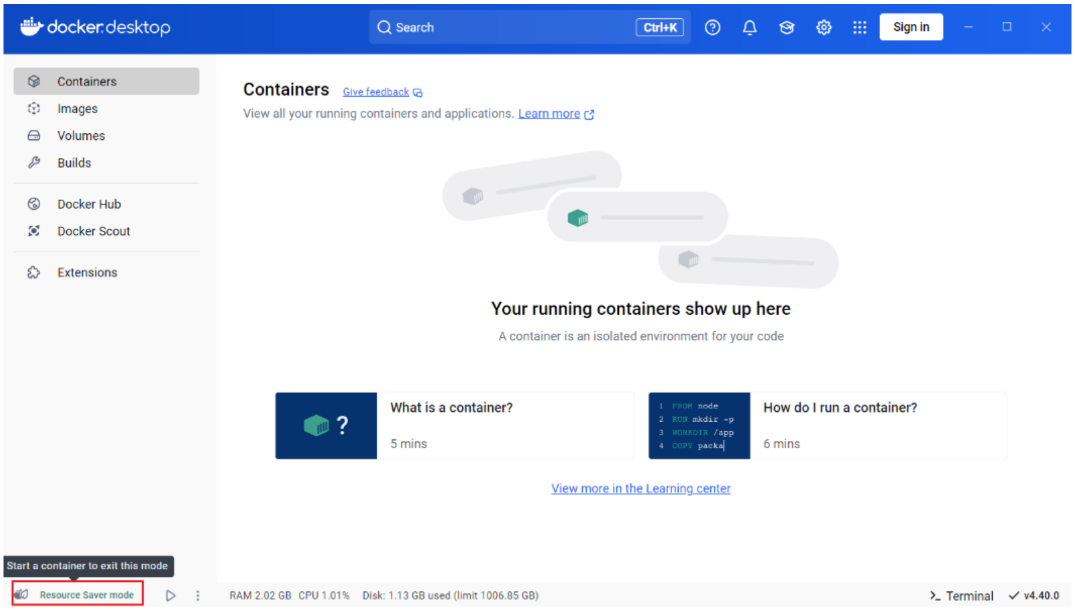
- Then, it displays a **Welcome Survey** page. Click on **Skip** link to skip this survey for now.



- 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.



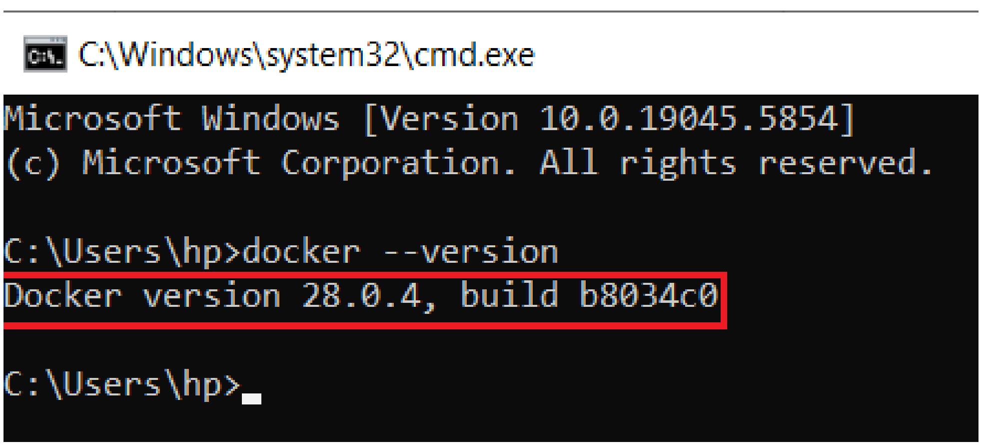
- Since no container is running at this moment, Docker engine goes into **Resource Save mode** in few seconds. It comes into running state again when a container is started.



5.3 Verify Docker

Run the following command to get the installed Docker version:

```
docker --version
```



It displays the current Docker version that was installed on the Windows system.

6 DOCKER COMMANDS

Use `docker help` to get the list of commands being used:

```
docker help
```

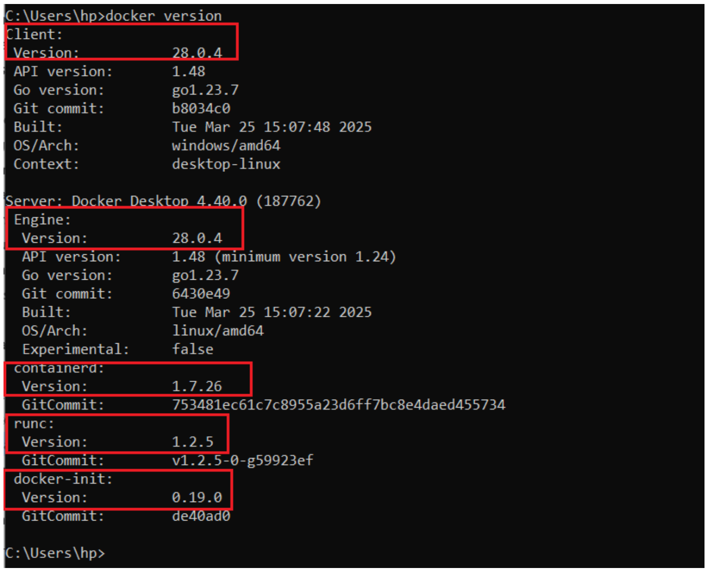
```
C:\Users\hp>docker help
Usage:  docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Common Commands:
  run         Create and run a new container from an image
  exec        Execute a command in a running container
  ps          List containers
  build       Build an image from a Dockerfile
  pull        Download an image from a registry
  push        Upload an image to a registry
  images      List images
  login       Authenticate to a registry
  logout      Log out from a registry
  search      Search Docker Hub for images
  version     Show the Docker version information
  info        Display system-wide information

Management Commands:
  ai*         Docker AI Agent - Ask Gordon
  builder     Manage builds
  buildx*     Docker Buildx
  cloud*      Docker Cloud
  compose*    Docker Compose
  container   Manage containers
  context     Manage contexts
  debug*      Get a shell into any image or container
  desktop*    Docker Desktop commands (Beta)
  dev*        Docker Dev Environments
  extension*  Manages Docker extensions
  image       Manage images
  init*       Creates Docker-related starter files for your project
  manifest    Manage Docker image manifests and manifest lists
  network     Manage networks
  plugin      Manage plugins
  sbom*       View the packaged-based Software Bill Of Materials (SBOM) for an image
  scout*      Docker Scout
  system      Manage Docker
  trust       Manage trust on Docker images
  volume      Manage volumes
```

6.1 Docker Common Commands

The common docker commands include:

- `docker version`: It displays the version details of all docker components installed on the host.

```
docker version
```



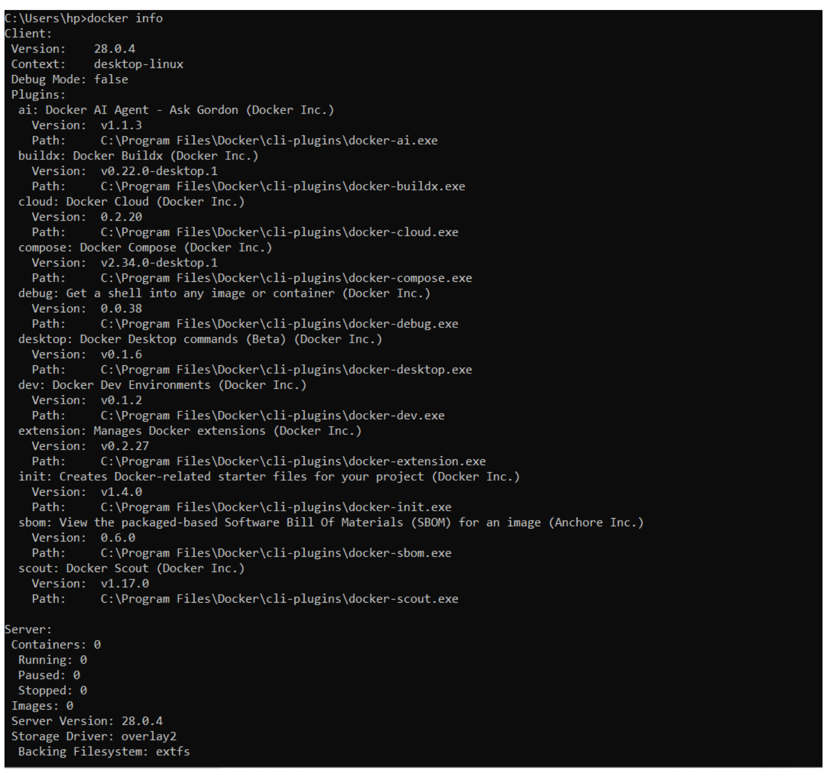
```
C:\Users\hp>docker version
Client:
 Version:           28.0.4
 API version:       1.48
 Go version:        go1.23.7
 Git commit:        b8034c0
 Built:             Tue Mar 25 15:07:48 2025
 OS/Arch:           windows/amd64
 Context:           desktop-linux

Server: Docker Desktop 4.40.0 (187762)
 Engine:
  Version:          28.0.4
  API version:      1.48 (minimum version 1.24)
  Go version:       go1.23.7
  Git commit:       6430e49
  Built:            Tue Mar 25 15:07:22 2025
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.7.26
  GitCommit:        753481ec61c7c8955a23d6ff7bc8e4daed455734
 runc:
  Version:          1.2.5
  GitCommit:        v1.2.5-0-g59923ef
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0

C:\Users\hp>
```

- `docker info`: It displays the system wide information.

```
docker info
```

or

```
docker system info
```



```
C:\Users\hp>docker info
Client:
 Version:    28.0.4
 Context:    desktop-linux
 Debug Mode: false
 Plugins:
  ai: Docker AI Agent - Ask Gordon (Docker Inc.)
    Version:  v1.1.3
    Path:     C:\Program Files\Docker\cli-plugins\docker-ai.exe
  buildx: Docker Buildx (Docker Inc.)
    Version:  v0.22.0-desktop.1
    Path:     C:\Program Files\Docker\cli-plugins\docker-buildx.exe
  cloud: Docker Cloud (Docker Inc.)
    Version:  0.2.20
    Path:     C:\Program Files\Docker\cli-plugins\docker-cloud.exe
  compose: Docker Compose (Docker Inc.)
    Version:  v2.34.0-desktop.1
    Path:     C:\Program Files\Docker\cli-plugins\docker-compose.exe
  debug: Get a shell into any image or container (Docker Inc.)
    Version:  0.0.38
    Path:     C:\Program Files\Docker\cli-plugins\docker-debug.exe
  desktop: Docker Desktop commands (Beta) (Docker Inc.)
    Version:  v0.1.6
    Path:     C:\Program Files\Docker\cli-plugins\docker-desktop.exe
  dev: Docker Dev Environments (Docker Inc.)
    Version:  v0.1.2
    Path:     C:\Program Files\Docker\cli-plugins\docker-dev.exe
  extension: Manages Docker extensions (Docker Inc.)
    Version:  v0.2.27
    Path:     C:\Program Files\Docker\cli-plugins\docker-extension.exe
  init: Creates Docker-related starter files for your project (Docker Inc.)
    Version:  v1.4.0
    Path:     C:\Program Files\Docker\cli-plugins\docker-init.exe
  sbom: View the packaged-based Software Bill Of Materials (SBOM) for an image (Anchore Inc.)
    Version:  0.6.0
    Path:     C:\Program Files\Docker\cli-plugins\docker-sbom.exe
  scout: Docker Scout (Docker Inc.)
    Version:  v1.17.0
    Path:     C:\Program Files\Docker\cli-plugins\docker-scout.exe

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 0
 Server Version: 28.0.4
 Storage Driver: overlay2
  Backing Filesystem: extfs
```

6.2 Docker Image Commands

Docker provides a set of handy commands to manage images. Some of them are listed below:

- `docker build`: It is used to build the Docker image with the help of `Dockerfile` available in the specified directory. This command allows to provide various options. Use `docker build --help` command for the complete list of options.

  Some common options are:
  - `-d` to enable debugging
  - `-f` to specify the name of Dockerfile. By default, it refers to the file named `Dockerfile`
  - `-q` to suppress the build output
  - `-t` to set the name and optionally a tag for the image

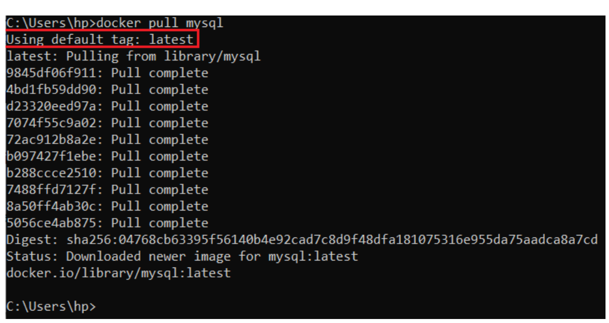
```
docker build <Dockerfile_location>
```
or
```
docker image build <Dockerfile_location>
```

- `docker pull`: It pulls the given image from the official docker registry which is **Docker Hub.** By default, it pulls the latest image from the Docker Hub but we can also mention the tag name (version) of the image to be pulled. For more options, use `docker pull --help` command.

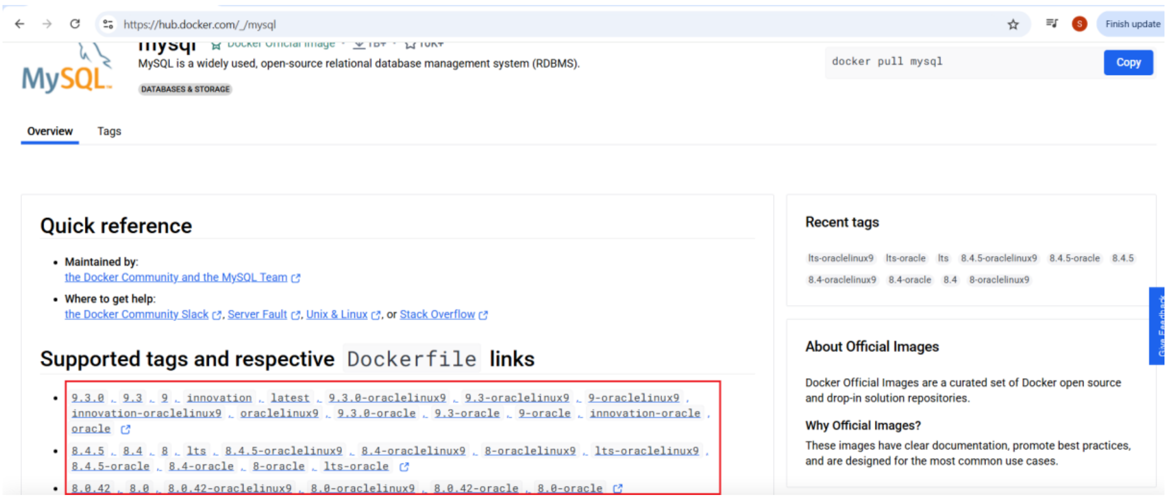
```
docker pull <image_name>
```
or
```
docker image pull <image_name>
```

Run the following command to pull the latest image of `mysql` from Docker Hub.

```
docker pull mysql
```

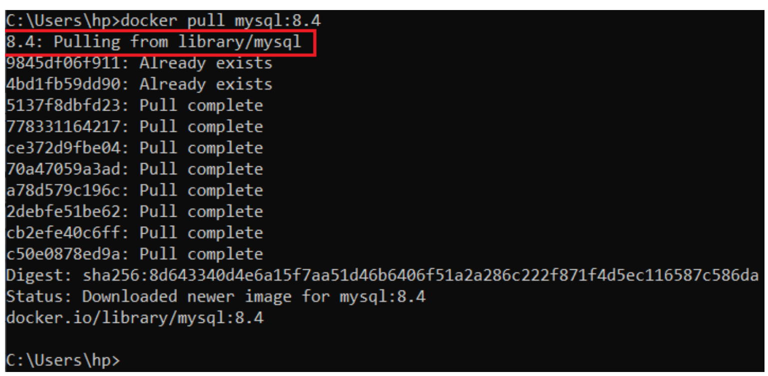


Note: Create a Docker Hub account at https://hub.docker.com/ and search for `mysql` to see the list of available tags.



Now, run the following command to pull the 8.4 version (tag) of `mysql` image.

```
docker pull mysql:8.4
```



- `docker images`: It lists out all available images that were built on host or pulled from Docker registry such as Docker Hub. For more options, use `docker image s --help` command.

```
docker images
```

or

```
docker image ls
```

```
C:\Users\hp>docker images
REPOSITORY   TAG      IMAGE ID       CREATED       SIZE
mysql        8.4      8f360cd2e6e4   7 weeks ago   777MB
mysql        latest   edbdd97bf78b   7 weeks ago   859MB

C:\Users\hp>
```

- `docker inspect`: It helps to debug the docker image if any errors occurred while building an image or pulling the image.

```
docker inspect <image_name or image_id>
```

```
C:\Users\hp>docker inspect mysql:8.4
[
    {
        "Id": "sha256:8f360cd2e6e4a37e8b8638fec65c779e4c1c1f4f785765bf2aab3fbc59f95a26",
        "RepoTags": [
            "mysql:8.4"
        ],
        "RepoDigests": [
            "mysql@sha256:8d643340d4e6a15f7aa51d46b6406f51a2a286c222f871f4d5ec116587c586da"
        ],
        "Parent": "",
        "Comment": "buildkit.dockerfile.v0",
        "Created": "2025-04-15T10:15:16Z",
        "DockerVersion": "",
        "Author": "",
        "Config": {
            "Hostname": "",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "3306/tcp": {},
                "33060/tcp": {}
            },
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                "GOSU_VERSION=1.17",
                "MYSQL_MAJOR=8.4",
                "MYSQL_VERSION=8.4.5-1.el9",
                "MYSQL_SHELL_VERSION=8.4.5-1.el9"
            ],
            "Cmd": [
                "mysqld"
            ],
            "Image": "",
            "Volumes": {
                "/var/lib/mysql": {}
            },
            "WorkingDir": "/",
            "Entrypoint": [
                "docker-entrypoint.sh"
            ],
            "OnBuild": null,
```

- `docker push`: It is used to push the docker image built in local to the Docker registry which is Docker Hub, by default.

```
docker push <image_name>
```

or

```
docker image push <image_name>
```

- `docker save`: It saves the docker image in the form of dockerfile. It can take `-o` flag to write to a specific output file instead of STDOUT.

```
docker save -o <output_file> <image_name>
```
or
```
docker image save -o <ou tput_file> <image_name>
```

Run the following command to save the `mysql` image to a tar file in the current directory:

```
docker save -o .\docker_mysql_save.tar mysql:8.4
```

```
C:\Users\hp>docker save -o .\docker_mysql_save.tar mysql:8.4

C:\Users\hp>
```

- `docker import`: It imports the contents from a tar file to create a file system image.

```
docker import <tar_file_name> <new_image_name>
```
or
```
docker image import <tar_file_name> <new_image_name>
```

Run the following command to import the `docker_mysql_save.tar` file as `mysql_import` image:

```
docker import docker_mysql_save.tar mysql_import
```

```
C:\Users\hp>docker import docker_mysql_save.tar mysql_import
sha256:a62c8d5940ffb6b2d7292de3ed0ec105a8251fc074bc273bf45d4cadc9c483c3

C:\Users\hp>docker images
REPOSITORY     TAG       IMAGE ID       CREATED         SIZE
mysql_import   latest    a62c8d5940ff   5 seconds ago   794MB
mysql          8.4       8f360cd2e6e4   7 weeks ago     777MB
mysql          latest    edbdd97bf78b   7 weeks ago     859MB

C:\Users\hp>
```

- `docker tag`: It creates a new image with a specific tag based on the given image. Tags in Docker images are used to identify and differentiate between different versions of an image.

```
docker tag <image_name> <image_name>:<tag>
```

or

```
docker image tag <image_name> <image_name>:<tag>
```

Run the following command to tag `mysql:8.4` as `mysql:v1.0`.

```
docker tag mysql:8.4 mysql:v1.0
```

```
C:\Users\hp>docker images
REPOSITORY     TAG       IMAGE ID       CREATED          SIZE
mysql_import   latest    a62c8d5940ff   33 seconds ago   794MB
mysql          8.4       8f360cd2e6e4   7 weeks ago      777MB
mysql          v1.0      8f360cd2e6e4   7 weeks ago      777MB
mysql          latest    edbdd97bf78b   7 weeks ago      859MB

C:\Users\hp>
```

- `docker history`: It displays the history of an image with details of layers created on top of the base image.

```
docker history <image_name_or_image_id>
```

or

```
docker image history <image_name_or_image_id>
```

Run the following command to check the history of the `mysql:8.4` image:

```
docker history mysql:8.4
```

```
C:\Users\hp>docker history mysql:8.4
IMAGE          CREATED       CREATED BY                                      SIZE      COMMENT
8f360cd2e6e4   7 weeks ago   CMD ["mysqld"]                                  0B        buildkit.dockerfile.v0
<missing>      7 weeks ago   EXPOSE map[3306/tcp:{} 33060/tcp:{}]            0B        buildkit.dockerfile.v0
<missing>      7 weeks ago   ENTRYPOINT ["docker-entrypoint.sh"]             0B        buildkit.dockerfile.v0
<missing>      7 weeks ago   COPY docker-entrypoint.sh /usr/local/bin/ # …   13.8kB    buildkit.dockerfile.v0
<missing>      7 weeks ago   VOLUME [/var/lib/mysql]                         0B        buildkit.dockerfile.v0
<missing>      7 weeks ago   RUN /bin/sh -c set -eux;  microdnf install -…   503MB     buildkit.dockerfile.v0
<missing>      7 weeks ago   ENV MYSQL_SHELL_VERSION=8.4.5-1.el9             0B        buildkit.dockerfile.v0
<missing>      7 weeks ago   RUN /bin/sh -c set -eu;  {   echo '[mysql-to…   219B      buildkit.dockerfile.v0
<missing>      7 weeks ago   RUN /bin/sh -c set -eux;  microdnf install -…   140MB     buildkit.dockerfile.v0
<missing>      7 weeks ago   RUN /bin/sh -c set -eu;  {   echo '[mysql8.4…   225B      buildkit.dockerfile.v0
<missing>      7 weeks ago   ENV MYSQL_VERSION=8.4.5-1.el9                   0B        buildkit.dockerfile.v0
<missing>      7 weeks ago   ENV MYSQL_MAJOR=8.4                             0B        buildkit.dockerfile.v0
<missing>      7 weeks ago   RUN /bin/sh -c set -eux;  key='BCA4 3417 C3B…   3.17kB    buildkit.dockerfile.v0
<missing>      7 weeks ago   RUN /bin/sh -c set -eux;  microdnf install -…   16.5MB    buildkit.dockerfile.v0
<missing>      7 weeks ago   RUN /bin/sh -c set -eux;  arch="$(uname -m)"…   2.36MB    buildkit.dockerfile.v0
<missing>      7 weeks ago   ENV GOSU_VERSION=1.17                           0B        buildkit.dockerfile.v0
<missing>      7 weeks ago   RUN /bin/sh -c set -eux;  groupadd --system …   2.77kB    buildkit.dockerfile.v0
<missing>      7 weeks ago   CMD ["/bin/bash"]                               0B        buildkit.dockerfile.v0
<missing>      7 weeks ago   ADD oraclelinux-9-slim-amd64-rootfs.tar.xz /…   114MB     buildkit.dockerfile.v0

C:\Users\hp>
```

- `docker rmi`: It removes the specified docker image.

```
docker rmi <image_name>
```
or
```
docker image rm <image_name>
```
or
```
docker image remove <image_name>
```

Run the below command to delete `mysql:v1.0` image.

```
docker rmi mysql:v1.0
```

```
C:\Users\hp>docker rmi mysql:v1.0
Untagged: mysql:v1.0

C:\Users\hp>docker images
REPOSITORY     TAG       IMAGE ID       CREATED              SIZE
mysql_import   latest    a62c8d5940ff   About a minute ago   794MB
mysql          8.4       8f360cd2e6e4   7 weeks ago          777MB
mysql          latest    edbdd97bf78b   7 weeks ago          859MB

C:\Users\hp>
```

- `docker image prune`: It removes all unused images that are not referenced by any container on the Docker host.

```
docker image prune
```

```
C:\Users\hp>docker image prune
WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N] y
Total reclaimed space: 0B

C:\Users\hp>
```

6.3 Docker Container Commands

Docker container commands are helpful to manage the lifecycle of containers such as create, start, stop, inspect, or remove containers. Some of them are listed below:

- `docker run`: This command is a combination of `docker create` and `docker start` commands. It is used to create a container (if not already exists) from the specified image and start it. If the container already exists, it simply launches the container. This command allows to provide various options. Use `docker run --help` command for the complete list of options.

Some common options are:

- `-c` to set CPU for the container to use
- `-d` to run container in the background (detached mode) and print container ID
- `-e` to set environment variables
- `-h` to specify container host name
- `-i` to run in interactive mode which keeps STDIN open even when not attached
- `-l` to set metadata on a container
- `-m` to set memory for the container to use
- `--name` to assign a name to the container
- `--network` to connect a container to the network
- `-p` to publish all container ports to random ports
- `-t` to allocate a pseudo-TTY
- `-u` to specify username or UID with or without group or GID
- `-v` to bind mount a volume to the container
- `-w` to set the working directory inside the container

We can specify either image name or image id to run the container and by default, the container gets an ID with an imaginary name such as `zen_dubinsky`, `heuristic_villani`, etc. which can be changed using `--name` option.

```
docker run <image_name>
```
or
```
docker container run <image_name>
```
or
```
docker run <image_id>
```

Note: Some images may expect additional parameters to start a container. For example, if we run `docker run mysql` command to start the container, it will fail since `mysql` image expects one of the parameters `MYSQL_ROOT_PASSWORD` or `MYSQL_ALLOW_EMPTY_PASSWORD` or `MYSQL_RANDOM_ROOT_PASSWORD` to start. Read Docker Hub MySQL documentation for more details.

```
C:\Users\hp>docker run mysql
2025-06-05 11:32:30+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 9.3.0-1.el9 started.
2025-06-05 11:32:31+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
2025-06-05 11:32:31+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 9.3.0-1.el9 started.
2025-06-05 11:32:31+00:00 [ERROR] [Entrypoint]: Database is uninitialized and password option is not specified
    You need to specify one of the following as an environment variable:
    - MYSQL_ROOT_PASSWORD
    - MYSQL_ALLOW_EMPTY_PASSWORD
    - MYSQL_RANDOM_ROOT_PASSWORD

C:\Users\hp>
```

Run the following command to create `mysql` container with a root password `root123`.

```
docker run --name mysql -e MYSQL_ROOT_PASSWORD=root123 mysql
```

```
C:\Users\hp>docker run --name mysql -e MYSQL_ROOT_PASSWORD=root123 mysql
2025-06-05 11:33:17+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 9.3.0-1.el9 started.
2025-06-05 11:33:18+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
2025-06-05 11:33:18+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 9.3.0-1.el9 started.
2025-06-05 11:33:18+00:00 [Note] [Entrypoint]: Initializing database files
2025-06-05T11:33:18.452758Z 0 [System] [MY-015017] [Server] MySQL Server Initialization - start.
2025-06-05T11:33:18.455508Z 0 [System] [MY-013169] [Server] /usr/sbin/mysqld (mysqld 9.3.0) initializing of server in pr
ogress as process 80
2025-06-05T11:33:18.590045Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
2025-06-05T11:33:27.480149Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
2025-06-05T11:33:46.497605Z 6 [Warning] [MY-010453] [Server] root@localhost is created with an empty password ! Please c
onsider switching off the --initialize-insecure option.
2025-06-05T11:34:07.981291Z 0 [System] [MY-015018] [Server] MySQL Server Initialization - end.
2025-06-05 11:34:08+00:00 [Note] [Entrypoint]: Database files initialized
2025-06-05 11:34:08+00:00 [Note] [Entrypoint]: Starting temporary server
2025-06-05T11:34:08.117144Z 0 [System] [MY-015015] [Server] MySQL Server - start.
2025-06-05T11:34:08.557219Z 0 [System] [MY-010116] [Server] /usr/sbin/mysqld (mysqld 9.3.0) starting as process 121
2025-06-05T11:34:08.646313Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
2025-06-05T11:34:18.213076Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
2025-06-05T11:34:20.549976Z 0 [Warning] [MY-010068] [Server] CA certificate ca.pem is self signed.
2025-06-05T11:34:20.550184Z 0 [System] [MY-013602] [Server] Channel mysql_main configured to support TLS. Encrypted conn
ections are now supported for this channel.
2025-06-05T11:34:20.627856Z 0 [Warning] [MY-011810] [Server] Insecure configuration for --pid-file: Location '/var/run/m
ysqld' in the path is accessible to all OS users. Consider choosing a different directory.
2025-06-05T11:34:20.784122Z 0 [System] [MY-011323] [Server] X Plugin ready for connections. Socket: /var/run/mysqld/mysq
lx.sock
2025-06-05T11:34:20.784418Z 0 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections. Version: '9.3.0'  s
ocket: '/var/run/mysqld/mysqld.sock'  port: 0  MySQL Community Server - GPL.
2025-06-05 11:34:20+00:00 [Note] [Entrypoint]: Temporary server started.
'/var/lib/mysql/mysql.sock' -> '/var/run/mysqld/mysqld.sock'
Warning: Unable to load '/usr/share/zoneinfo/iso3166.tab' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/leap-seconds.list' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/leapseconds' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/tzdata.zi' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/zone.tab' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/zone1970.tab' as time zone. Skipping it.

2025-06-05 11:34:28+00:00 [Note] [Entrypoint]: Stopping temporary server
2025-06-05T11:34:28.063324Z 11 [System] [MY-013172] [Server] Received SHUTDOWN from user root. Shutting down mysqld (Ver
sion: 9.3.0).
2025-06-05T11:34:31.191132Z 0 [System] [MY-010910] [Server] /usr/sbin/mysqld: Shutdown complete (mysqld 9.3.0)  MySQL Co
mmunity Server - GPL.
2025-06-05T11:34:31.191186Z 0 [System] [MY-015016] [Server] MySQL Server - end.
2025-06-05 11:34:32+00:00 [Note] [Entrypoint]: Temporary server stopped

2025-06-05 11:34:32+00:00 [Note] [Entrypoint]: MySQL init process done. Ready for start up.

2025-06-05T11:34:32.114084Z 0 [System] [MY-015015] [Server] MySQL Server - start.
2025-06-05T11:34:32.539849Z 0 [System] [MY-010116] [Server] /usr/sbin/mysqld (mysqld 9.3.0) starting as process 1
2025-06-05T11:34:32.599127Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
2025-06-05T11:34:42.337031Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
2025-06-05T11:34:44.668589Z 0 [Warning] [MY-010068] [Server] CA certificate ca.pem is self signed.
2025-06-05T11:34:44.668952Z 0 [System] [MY-013602] [Server] Channel mysql_main configured to support TLS. Encrypted conn
ections are now supported for this channel.
2025-06-05T11:34:44.752776Z 0 [Warning] [MY-011810] [Server] Insecure configuration for --pid-file: Location '/var/run/m
ysqld' in the path is accessible to all OS users. Consider choosing a different directory.
2025-06-05T11:34:44.965818Z 0 [System] [MY-011323] [Server] X Plugin ready for connections. Bind-address: '::' port: 330
60, socket: /var/run/mysqld/mysqlx.sock
2025-06-05T11:34:44.966287Z 0 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections. Version: '9.3.0'  s
ocket: '/var/run/mysqld/mysqld.sock'  port: 3306  MySQL Community Server - GPL.
```

As we can see, MySQL has been initialized successfully and ready to listen for connections. Do not close this window.

- `docker ps`: It is used to get a list of all running containers on the host. It can accept various options as below. Use `docker ps --help` command for complete options:
  - `-a` to show all containers whether stopped or running
  - `-l` to show only latest containers
  - `-q` to show only id of the containers.

```
docker ps
```
or
```
docker container ls
```
or
```
docker container list
```

Open a new command prompt and run these commands:

```
docker ps
```

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.19045.5854]
(c) Microsoft Corporation. All rights reserved.

C:\Users\hp>docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS                 NAMES
7a931bb4e7dd   mysql     "docker-entrypoint.s…"   5 minutes ago   Up 5 minutes   3306/tcp, 33060/tcp   mysql

C:\Users\hp>
```

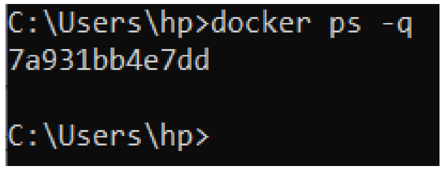
```
docker ps -l
```

```
C:\Users\hp>docker ps -l
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS                 NAMES
7a931bb4e7dd   mysql     "docker-entrypoint.s…"   6 minutes ago   Up 6 minutes   3306/tcp, 33060/tcp   mysql

C:\Users\hp>
```

```
docker ps -a
```

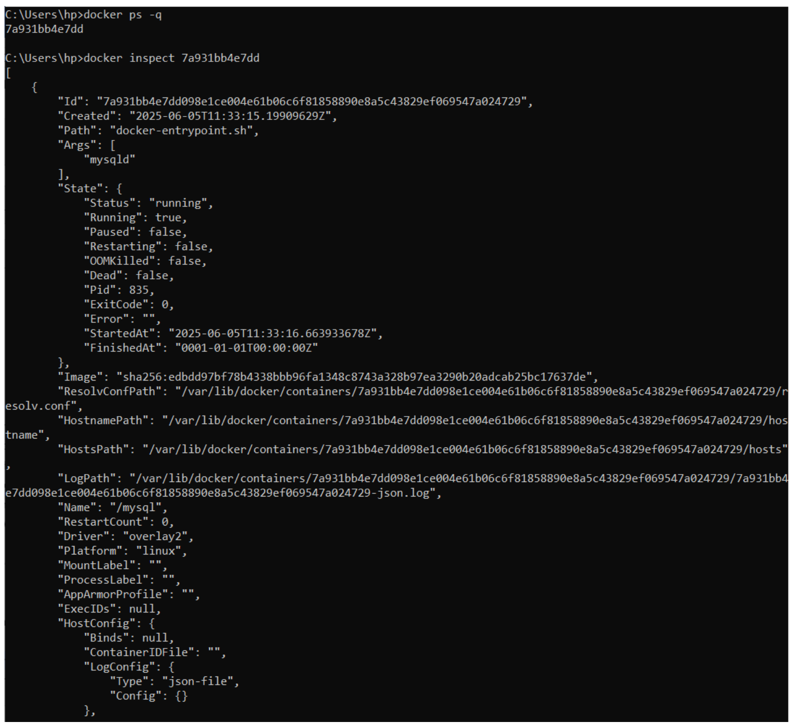
```
C:\Users\hp>docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS                     PORTS                 NAMES
7a931bb4e7dd   mysql     "docker-entrypoint.s…"   6 minutes ago   Up 6 minutes               3306/tcp, 33060/tcp   mysql
40348607f098   mysql     "docker-entrypoint.s…"   7 minutes ago   Exited (1) 7 minutes ago                         thirs
ty_kirch

C:\Users\hp>
```

```
docker ps -q
```



- `docker inspect`: It is used to inspect the Docker containers.

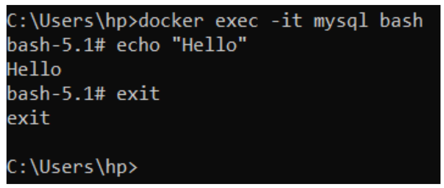
```
docker inspect <container_id or container_name>
```



- `docker exec`: It allows to execute the commands in the running containers. It accepts various options including:
  - `-d` to run command in the background (detached mode)
  - `-e` to set environment variables
  - `-i` to run in interactive mode which keeps STDIN open even when not attached
  - `-t` to allocate a pseudo-TTY
  - `-u` to specify username or UID with or without group or GID
  - `-w` to set the working directory inside the container

Run the following commands to enter into bash prompt inside `mysql` container:

```
docker exec -it mysql bash
echo "Hello"
exit
```



- `docker attach`: It is used to attach local host standard input, output, and error to the primary process of a running container so that it allows to interact with the process inside the container as if the process is running in our terminal.

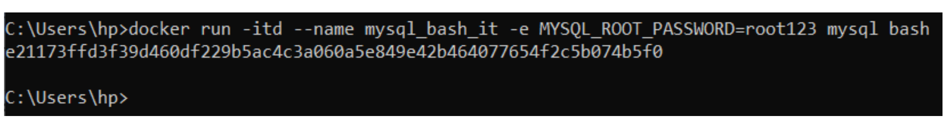
```
docker attach <container_id or name>
```

or

```
docker container attach <container_id or name>
```

First, run the following command to start a new `mysql` container in interactive and detached mode with `bash` process running inside it.

```
docker run -itd --name mysql_bash_it -e MYSQL_ROOT_PASSWORD=root123
mysql bash
```



Then, run the below command to attach the terminal to the new container named `mysql_bash_it` and start interacting with container:

```
docker attach mysql_bash_it
echo "Hello"
exit
```

```
C:\Users\hp>docker attach mysql_bash_it
bash-5.1# echo "Hello"
Hello
bash-5.1# exit
exit

C:\Users\hp>
```

- `docker cp`: It is used to copy files between a container and docker host.

```
docker cp <local_file_path> <container_name or id>:<container_path>
```

or

```
docker container cp <local_file_path> <container_name or
id>:<container_path>
```

Open a new command prompt and run the following command to copy `tmp.txt` file to `mysql` container:

```
docker cp .\tmp.txt mysql:/tmp
```

```
C:\Users\hp>docker cp .\tmp.txt mysql:/tmp
Successfully copied 1.54kB to mysql:/tmp

C:\Users\hp>
```

Run the following commands to connect to `mysql` container and verify if the copied file exists:

```
docker exec -it mysql bash
ls /tmp/tmp.txt
exit
```

```
C:\Users\hp>docker exec -it mysql bash
bash-5.1# ls /tmp/tmp.txt
/tmp/tmp.txt
bash-5.1# exit
exit

C:\Users\hp>
```

- `docker diff`: It allows to find the differences between the container's filesystem and it's base image.

```
docker diff <container_id_or_name>
```
or
```
docker container diff <container_id_or_name>
```

Run the following command to see the changes in `mysql` container filesystem:

```
docker diff mysql
```

```
C:\Users\hp>docker diff mysql
C /tmp
A /tmp/tmp.txt
C /root
A /root/.bash_history
C /run
C /run/mysqld
A /run/mysqld/mysqld.sock
A /run/mysqld/mysqld.sock.lock
A /run/mysqld/mysqlx.sock
A /run/mysqld/mysqlx.sock.lock
A /run/mysqld/mysqld.pid

C:\Users\hp>
```

- `docker commit`: It is used to take an image of a running container. When the image is taken, all the processes are paused so that the image does not encounter any problems once it is used to create one more container.

```
docker commit <container_id_or_name> <image_name_and_or_tag>
```
or
```
docker container commit <container_id_or_name>
<image_name_and_or_tag>
```

Run the following command to create a new image based on `mysql` container filesystem:

```
docker commit mysql mysql:v1
```

```
C:\Users\hp>docker commit mysql mysql:v1
sha256:d7384f5213d70d73458f5193af8cd8a6570de97c20f66e0ce0d6214ba8482f2c

C:\Users\hp>docker images
REPOSITORY     TAG       IMAGE ID       CREATED          SIZE
mysql          v1        d7384f5213d7   6 seconds ago    859MB
mysql_import   latest    a62c8d5940ff   19 minutes ago   794MB
mysql          8.4       8f360cd2e6e4   7 weeks ago      777MB
mysql          latest    edbdd97bf78b   7 weeks ago      859MB

C:\Users\hp>
```

- `docker export`: It is used to export the container's file system as a tar archive which can be shared across different hosts. Specify the output file name using `–o` option.

```
docker export -o <output_file_name> <container_id_or_name>
```
or
```
docker container export -o <output_file_name>
<container_id_or_name>
```

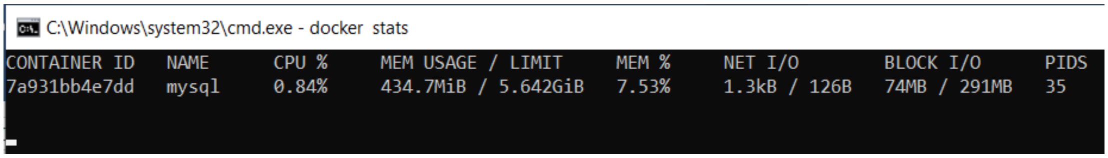
Run the following command to export `mysql` container filesystem as `mysql_container_export.tar` in the current directory:

```
docker export -o mysql_container_export.tar mysql
```

```
C:\Users\hp>docker export -o mysql_container_export.tar mysql

C:\Users\hp>dir mysql_container_export.tar
 Volume in drive C is SSD
 Volume Serial Number is 764D-D602

 Directory of C:\Users\hp

06/05/2025  05:19 PM       849,101,312 mysql_container_export.tar
               1 File(s)    849,101,312 bytes
               0 Dir(s)  26,348,359,680 bytes free

C:\Users\hp>
```

- `docker stats`: It is used to display the live stream of running containers resource usage statistics. It accepts various options including:
  - `-a` to show all containers live stream
  - `-- no-stream` to disable streaming stats and only pull the first result
  - `--no-trunc` to not truncate output

```
docker stats
```
or
```
docker container stats
```



```
C:\Windows\system32\cmd.exe - docker  stats
CONTAINER ID   NAME    CPU %   MEM USAGE / LIMIT     MEM %   NET I/O         BLOCK I/O        PIDS
7a931bb4e7dd   mysql   0.84%   434.7MiB / 5.642GiB   7.53%   1.3kB / 126B    74MB / 291MB     35
```

- `docker logs`: It is used to fetch the logs of a container. It accepts various options including:
  - `--details` to show extra details of logs
  - `-f` to continuously streaming the new output from container's STDOUT and STDERR
  - `-n` to display number of lines from the end of logs
  - `-t` to show timestamps of logs

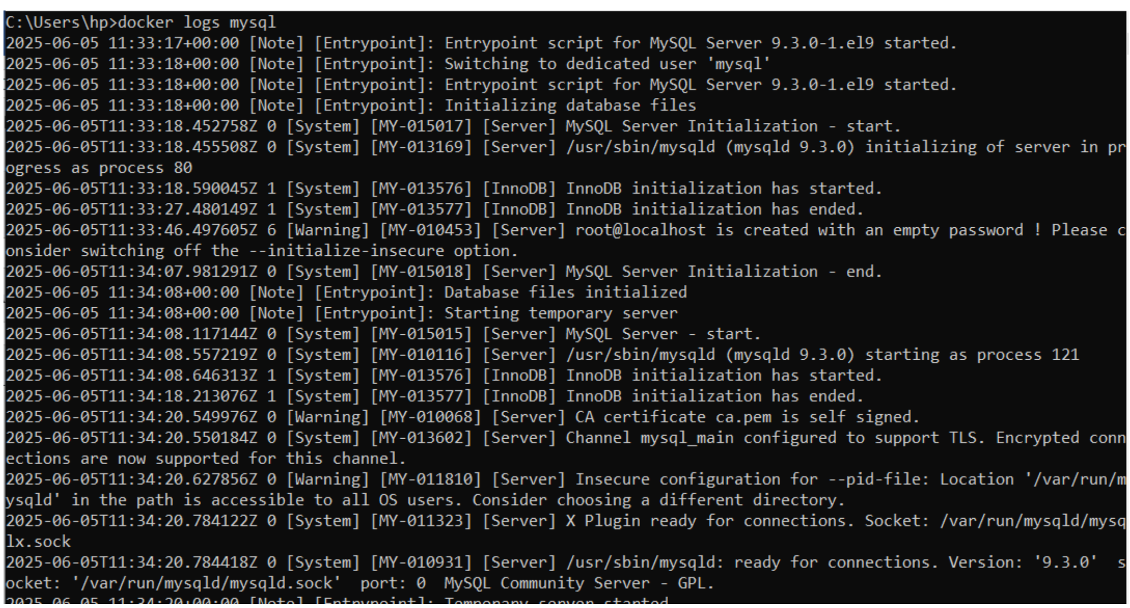
```
docker logs <container_id_or_name>
```

or

```
docker container logs <container_id_or_name>
```

Run the following command to get the `mysql` container logs until now:

```
docker logs mysql
```



```
C:\Users\hp>docker logs mysql
2025-06-05 11:33:17+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 9.3.0-1.el9 started.
2025-06-05 11:33:18+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
2025-06-05 11:33:18+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 9.3.0-1.el9 started.
2025-06-05 11:33:18+00:00 [Note] [Entrypoint]: Initializing database files
2025-06-05T11:33:18.452758Z 0 [System] [MY-015017] [Server] MySQL Server Initialization - start.
2025-06-05T11:33:18.455508Z 0 [System] [MY-013169] [Server] /usr/sbin/mysqld (mysqld 9.3.0) initializing of server in pr
ogress as process 80
2025-06-05T11:33:18.590045Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
2025-06-05T11:33:27.480149Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
2025-06-05T11:33:46.497605Z 6 [Warning] [MY-010453] [Server] root@localhost is created with an empty password ! Please c
onsider switching off the --initialize-insecure option.
2025-06-05T11:34:07.981291Z 0 [System] [MY-015018] [Server] MySQL Server Initialization - end.
2025-06-05 11:34:08+00:00 [Note] [Entrypoint]: Database files initialized
2025-06-05 11:34:08+00:00 [Note] [Entrypoint]: Starting temporary server
2025-06-05T11:34:08.117144Z 0 [System] [MY-015015] [Server] MySQL Server - start.
2025-06-05T11:34:08.557219Z 0 [System] [MY-010116] [Server] /usr/sbin/mysqld (mysqld 9.3.0) starting as process 121
2025-06-05T11:34:08.646313Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
2025-06-05T11:34:18.213076Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
2025-06-05T11:34:20.549976Z 0 [Warning] [MY-010068] [Server] CA certificate ca.pem is self signed.
2025-06-05T11:34:20.550184Z 0 [System] [MY-013602] [Server] Channel mysql_main configured to support TLS. Encrypted conn
ections are now supported for this channel.
2025-06-05T11:34:20.627856Z 0 [Warning] [MY-011810] [Server] Insecure configuration for --pid-file: Location '/var/run/m
ysqld' in the path is accessible to all OS users. Consider choosing a different directory.
2025-06-05T11:34:20.784122Z 0 [System] [MY-011323] [Server] X Plugin ready for connections. Socket: /var/run/mysqld/mysq
lx.sock
2025-06-05T11:34:20.784418Z 0 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections. Version: '9.3.0'  s
ocket: '/var/run/mysqld/mysqld.sock'  port: 0  MySQL Community Server - GPL.
```

- `docker kill`: It is used to kill one or more running containers by killing all processes abruptly inside the container.

```
docker kill <container_id or name>
```
or
```
docker container kill <container_id or name>
```

Run the following command to kill the `mysql` container:

```
docker kill mysql
```

```
C:\Users\hp>docker kill mysql
mysql

C:\Users\hp>docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES

C:\Users\hp>
```

- `docker start`: It is used to start one or more running containers.

```
docker start <container_id or name>
```
or
```
docker container start <container_id or name>
```

Run the following command to start the `mysql` container:

```
docker start mysql
```

```
C:\Users\hp>docker start mysql
mysql

C:\Users\hp>docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS         PORTS                 NAMES
7a931bb4e7dd   mysql     "docker-entrypoint.s…"   22 minutes ago   Up 2 seconds   3306/tcp, 33060/tcp   mysql

C:\Users\hp>
```

- `docker wait:` It is used to wait or halt the execution until one or more specified containers have stopped and print their exit codes the logs of a container.

```
docker wait <container_id_or_name>
```
or

```
docker container wait <container_id_or_name>
```

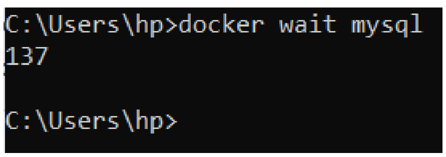
Run the following command to get the exit status of `mysql` container:

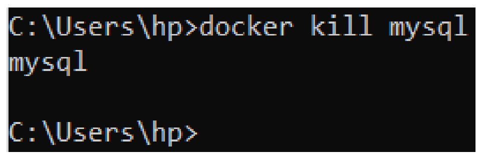
```
docker wait mysql
```

In another command prompt, run the following command to kill the container:

```
docker kill mysql
```

Then, you can see exit code `137` on `docker wait` terminal.



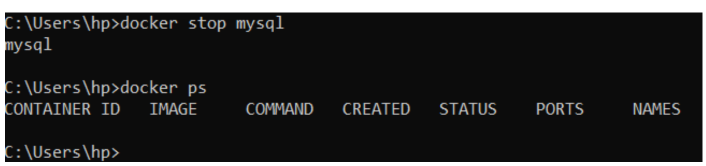


- `docker stop`: It is used to stop one or more running containers.

```
docker stop <container_id or name>
```
or
```
docker container stop <container_id or name>
```

Run the following command to stop the `mysql` container:

```
docker stop mysql
```

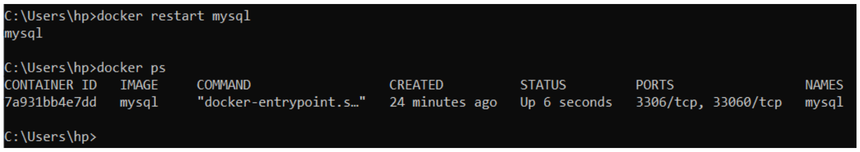


- `docker restart`: It is used to stop and start one or more containers.

```
docker restart <container_id or name>
```
or
```
docker container restart <container_id or name>
```

Run the following command to restart the `mysql` container:

```
docker restart mysql
```

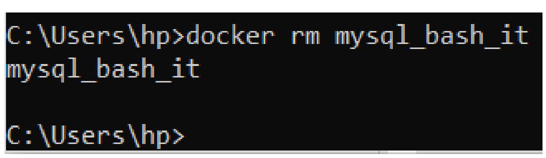


- `docker rm`: It is used to delete the specified container. We can mention either container Id or container name. By default, it deletes the container that is not currently running. It can also take some flags such as:
  - -f to remove the container forcefully even if it is running
  - -v to remove volumes
  - -l to remove specific link mentioned

```
docker rm <container_name or ID>
```
or
```
docker container rm <container_name or ID>
```
or
```
docker container remove <container_name or ID>
```

Run the following command to remove the container named `mysql_bash_it` which is in **Exited** state. If the container is removed successfully, it displays the container name that got deleted.

```
docker rm mysql_bash_it
```



- `docker container prune`: It removes all stopped containers on the Docker host.

```
docker container prune
```

```
C:\Users\hp>docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
40348607f0989f54f2c88cc143fd88cb6ca5164d55bb1b6f473d87b5408c49eb

Total reclaimed space: 0B

C:\Users\hp>
```

6.4 Docker Volume Commands

Docker volume commands are used for data persistence and sharing between containers. They allow to store data independently of container lifecycles. Some of these commands are listed below:

- `docker volume create`: It is used to create a new Docker volume.

```
docker volume create <volume_name>
```

Run the following command to create a new volume named `mysql_volume`:

```
docker volume create mysql_volume
```

```
C:\Users\hp>docker volume create mysql_volume
mysql_volume

C:\Users\hp>
```

- `docker volume ls`: It is used to list all volumes available on the Docker host.

```
docker volume ls
```

or

```
docker volume list
```

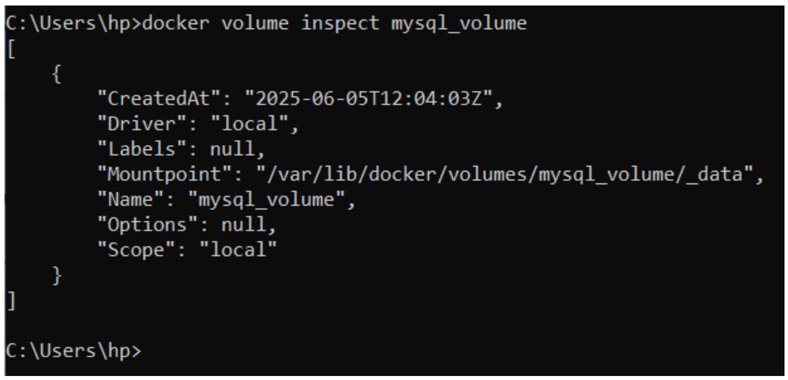
```
C:\Users\hp>docker volume ls
DRIVER    VOLUME NAME
local     b75cde356a3f12d83385c23be939ae31cf4469380701a64b7579a6cdc4f09e79
local     c34ace6b8bf943c30833e989876f1441dfa02aeb6387aeb3c284adf885941320
local     cc1a6a41897191331036ae4d9e63a3511809fca1af3637591ed6c71e361b5d45
local     mysql_volume

C:\Users\hp>
```

- `docker volume inspect`: It is used to get the details of a specified volume.

```
docker volume inspect <volume_name>
```

Run the following command to inspect `mysql_volume`:

```
docker volume inspect mysql_volume
```

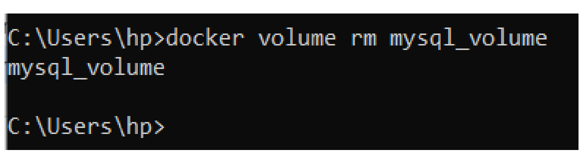


- `docker volume rm`: It allows to remove the specified volume from the Docker host.

```
docker volume rm <volume_name>
```
or
```
docker volume remove <volume_name>
```

Run the following command to remove the volume named `mysql_volume`:

```
docker volume rm mysql_volume
```



- `docker volume prune`: It is used to remove all unused volumes from the Docker host to free up space.

```
docker volume prune
```

```
C:\Users\hp>docker volume prune
WARNING! This will remove anonymous local volumes not used by at least one container.
Are you sure you want to continue? [y/N] y
Deleted Volumes:
b75cde356a3f12d83385c23be939ae31cf4469380701a64b7579a6cdc4f09e79
c34ace6b8bf943c30833e989876f1441dfa02aeb6387aeb3c284adf885941320

Total reclaimed space: 0B

C:\Users\hp>
```

6.5 Docker Network Commands

Docker network commands are used to attach containers to a network. Some of these commands are listed below:

- `docker network create`: It is used to create a new Docker network.

```
docker network create <network_name>
```

Run the following command to create a new network named `mynetwork`:

```
docker network create mynetwork
```

```
C:\Users\hp>docker network create mynetwork
af199d3995039e1f8d853dc56a629c8c1c868bda2868d85fe625b007ca94a5f3

C:\Users\hp>
```

- `docker network ls`: It is used to list all networks created on the Docker host.

```
docker network ls
```

or

```
docker network list
```

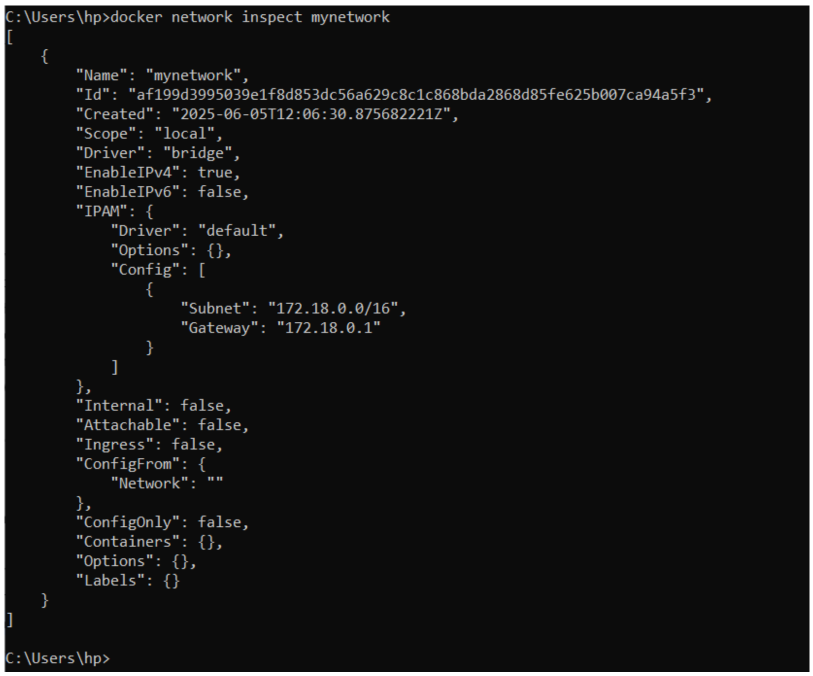
```
C:\Users\hp>docker network ls
NETWORK ID     NAME        DRIVER    SCOPE
e3fd1b94cb33   bridge      bridge    local
6bc62a19d6bf   host        host      local
af199d399503   mynetwork   bridge    local
202439de8a5a   none        null      local

C:\Users\hp>
```

- `docker network inspect`: It is used to get the details of a specified volume.

```
docker network inspect <network_name>
```

Run the following command to inspect `mynetwork`:

```
docker network inspect mynetwork
```



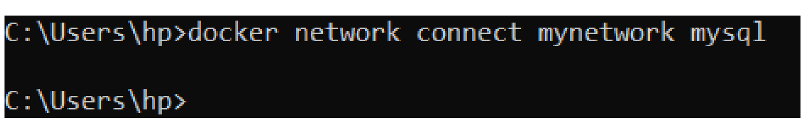
```
C:\Users\hp>docker network inspect mynetwork
[
    {
        "Name": "mynetwork",
        "Id": "af199d3995039e1f8d853dc56a629c8c1c868bda2868d85fe625b007ca94a5f3",
        "Created": "2025-06-05T12:06:30.875682221Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv4": true,
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]

C:\Users\hp>
```

- `docker network connect`: It allows to connect the specified container to a Docker network.

```
docker network connect <network_name> <container_id or name>
```

Run the following command to connect `mysql` container to `mynetwork` network:

```
docker network connect mynetwork mysql
```



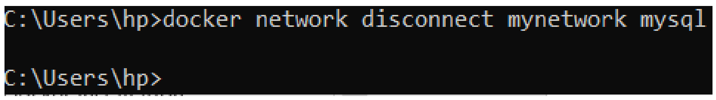
```
C:\Users\hp>docker network connect mynetwork mysql

C:\Users\hp>
```

- `docker network disconnect`: It allows to disconnect the specified container from the given Docker network.

```
docker network disconnect <network_name> <container_id or name>
```

Run the following command to disconnect `mysql` container from `mynetwork` network:

```
docker network disconnect mynetwork mysql
```



```
C:\Users\hp>docker network disconnect mynetwork mysql

C:\Users\hp>
```

- `docker network rm`: It allows to remove the specified network from the Docker host.

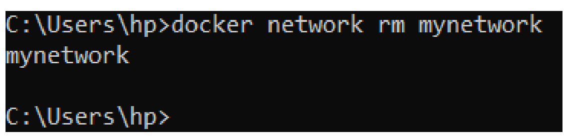
```
docker network rm <volume_name>
```

or

```
docker network remove <volume_name>
```

Run the following command to remove the network named `mynetwork`:

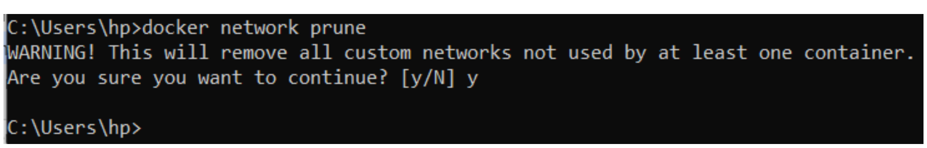
```
docker network rm mynetwork
```



```
C:\Users\hp>docker network rm mynetwork
mynetwork

C:\Users\hp>
```

- `docker network prune`: It is used to remove all unused networks from the Docker host to free up space.

```
docker network prune
```



```
C:\Users\hp>docker network prune
WARNING! This will remove all custom networks not used by at least one container.
Are you sure you want to continue? [y/N] y

C:\Users\hp>
```

6.6 Docker Registry Commands

Docker registry commands are used to work with the configured Docker registry. Some of these commands are listed below:

- `docker login`: It is used to authenticate to a Docker registry. By default, it authenticates to Docker Hub using a device code flow which lets to authenticate to Docker Hub without entering the password by just visiting a URL and enter the confirmation code and authenticate.

```
docker login
```

To authenticate with a specific user name and password use `-u` and `-p` options:

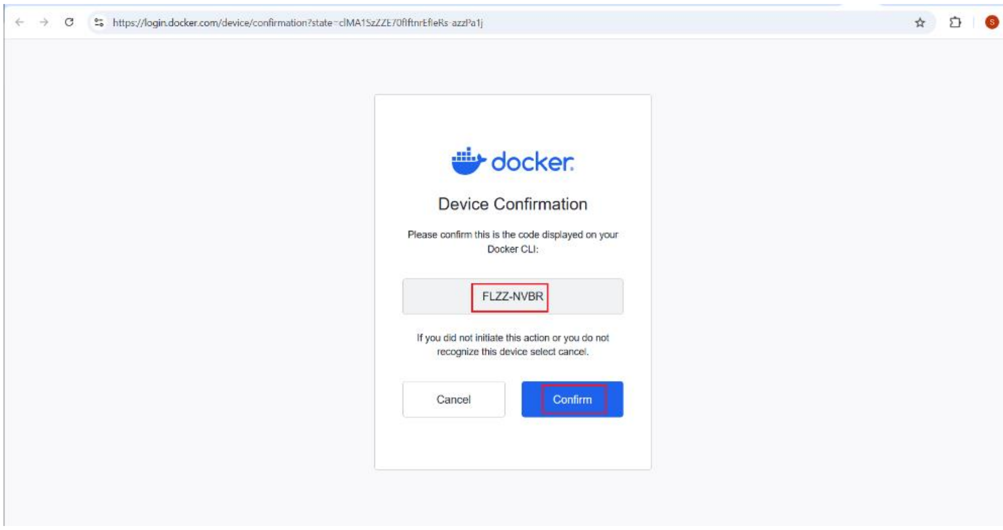
```
docker login -u <username> -p <password>
```

Run the following command to get authenticated to Docker Hub without providing credentials.

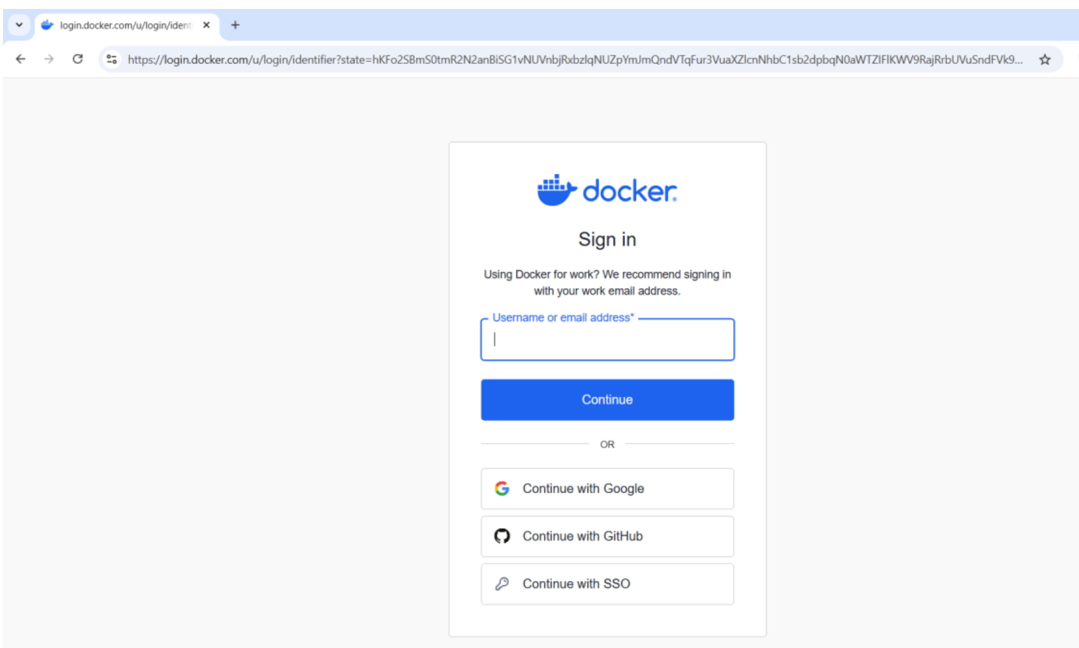
```
docker login
```

It provides a confirmation code and press **Enter** to open the login URL in the browser.

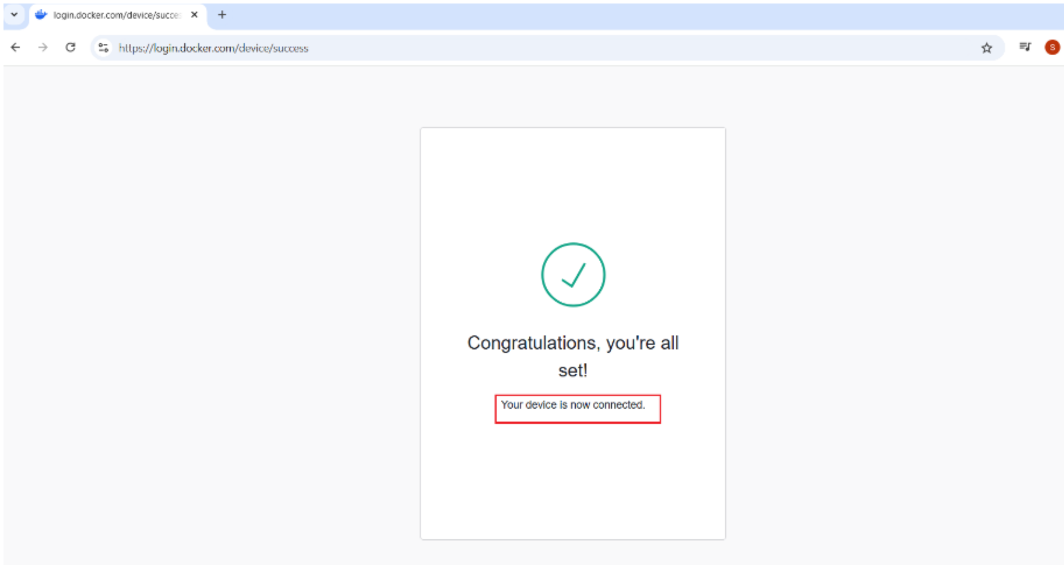
On the browser, verify the confirmation code and click on **Confirm** button:



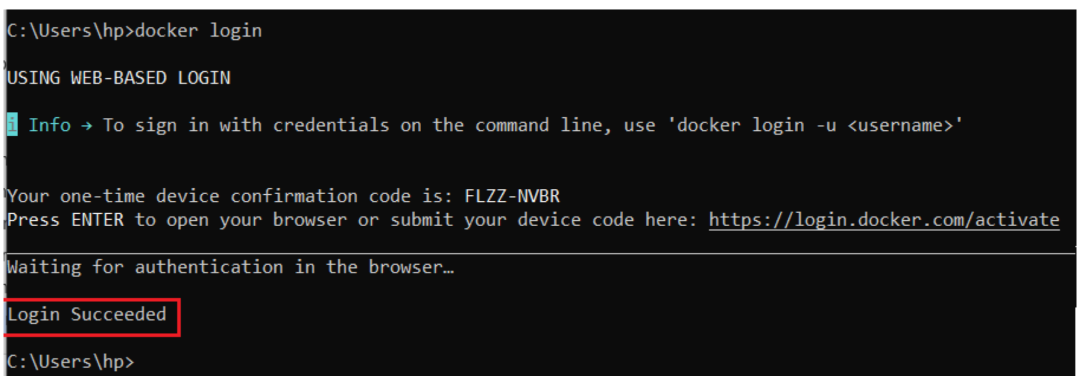
You can enter your Docker Hub username or directly login with **Google** credentials.



It then confirms that device is successfully connected.



On the command prompt, it displays **Login Succeeded** message.



- `docker search`: It is used to search the specific image in the Docker Hub.

```
docker search <image_name>
```

Run the following command to search `mysql` image:

```
docker search mysql
```

```
C:\Users\hp>docker search mysql
NAME                    DESCRIPTION                                     STARS     OFFICIAL
mysql                   MySQL is a widely used, open-source relation…   15813     [OK]
bitnami/mysql           Bitnami container image for MySQL               135
circleci/mysql          MySQL is a widely used, open-source relation…   31
bitnamicharts/mysql     Bitnami Helm chart for MySQL                    0
cimg/mysql                                                              3
ubuntu/mysql            MySQL open source fast, stable, multi-thread…   68
google/mysql            MySQL server for Google Compute Engine          26
linuxserver/mysql       A Mysql container, brought to you by LinuxSe…   44
elestio/mysql           Mysql, verified and packaged by Elestio         1
docksal/mysql           MySQL service images for Docksal - https://d…   0
alpine/mysql            mysql client                                    4
datajoint/mysql         MySQL image pre-configured to work smoothly …   2
ilios/mysql             Mysql configured for running Ilios              1
ddev/mysql              ARM64 base images for ddev-dbserver-mysql-8.…   1
mirantis/mysql                                                          0
corpusops/mysql         https://github.com/corpusops/docker-images/     0
mysql/mysql-server      Optimized MySQL Server Docker images. Create…   1030
vulhub/mysql                                                            1
cbioportal/mysql        This repository hosts MySQL database images …   1
vitess/mysql            Lightweight image to run MySQL with Vitess      1
mysql/mysql-router      MySQL Router provides transparent routing be…   28
mysql/mysql-cluster     Experimental MySQL Cluster Docker images. Cr…   100
nasqueron/mysql                                                         1
mysql/mysql-operator    MySQL Operator for Kubernetes                   1
openeuler/mysql                                                         0

C:\Users\hp>
```

- `docker logout`: It is used to log out from the Docker registry which is Docker Hub by default.

```
docker logout
```

```
C:\Users\hp>docker logout
Removing login credentials for https://index.docker.io/v1/

C:\Users\hp>
```

7 DOCKERFILE

A **`Dockerfile`** is a text file that contains various instructions *(or commands that can be called on the command-line as well)* to build a Docker image. Each instruction in the `Dockerfile` creates a new layer in the image. Docker reads instructions one by one in the `Dockerfile` and executes them to create an image so that the resulting image can be pushed to a Docker registry and shared with others.

`Dockerfile` ensures that applications and dependencies are packaged and deployed consistently across different environments, without worrying about the underlying infrastructure.

`Dockerfile` typically includes the following:

- Starts with a base or parent image that provides operating system and runtime environment for the application.
- Commands to update the base image and install additional software.
- Commands to build artifacts or dependencies to include.
- Commands to expose services such as storage and network.
- Command to run when the container is launched.

The basic `Dockerfile` instructions which are commonly used:

- `FROM` - This is used for to set the base image that provides OS and runtime environment. This instruction gets executed first before any other instructions and hence, it is very important to mention this in the first line of docker file. A Dockerfile can have multiple `FROM` instructions to produce more than one image.

```
FROM <image_name>
```

For example:

```
FROM ubuntu:19.04
```

- `MAINTAINER` - This is a non-executable instruction and is generally used to specify the author of the Dockerfile.

```
MAINTAINER <author_name> <email> <other_details>
```

For example:

```
MAINTAINER Firstname Lastname example@google.com
```

- LABEL - This is used to provide metadata in the form of key-value pairs for the Docker image, such as name, version, description, etc. It is always good to use few LABEL instructions wherever needed as labels are helpful for organizing and managing images, automating processes, and providing governance and ownership information. All labels can be inspected using the `docker inspect` command.

```
LABEL <key1>="<value1>" <key2>="<value2>"
```

For example:

```
LABEL name="example"
LABEL email="user1@example.com"
LABEL tutorial1="Docker" tutorial2="LABEL INSTRUCTION"
```

- ENV - This is used to set the environment variables in the Docker file while building the image which are persisted when a container is run. This instruction can be used within Dockerfile and also can be used by any scripts that Dockerfile calls.

```
ENV <variable>=<value>
```

or

```
ENV <variable> <value>
```

For example:

```
ENV URL_POST=production.example-gfg.com
```

- COPY - This is used to copy the files and directories from the host machine into the container image while building the image.

```
COPY <source> <destination>
```

For example:

```
COPY /target/java-web-app.war /usr/local/webapps/java-web-app.war
```

- ADD - This is similar to COPY instruction which is used to copy files, directories from host to Docker image but ADD instruction has additional features of auto-decompressing archives and fetching files from remote URLs to the file system of the image.

```
ADD <url>
```

For example:

```
ADD run.sh /run.sh
ADD project.tar.gz /install/
ADD https://project.example.com/testingproject.rpm/test
```

- RUN - This allows to execute scripts and commands on top of the existing image or layer and create a new layer with the result of command execution. It basically tells which process will be running inside the container at the run time.

```
RUN <command> <arguments>
```

or

```
RUN ["<command>", "<arguments>"]
```

For example:

```
RUN unzip install.zip /opt/install
RUN echo hello
```

- CMD – This is used to start the process inside the container but does not perform anything during the building of docker image. This instruction can be overridden and is mostly used to launch the software required in the container. There can only be one CMD instruction in a Dockerfile and if there are more than one CMD, then only the last CMD will take effect.

```
CMD ["<command>", "<arguments>"]
```

or

```
CMD command arguments
```

For example:

```
CMD ["program-foreground"]
CMD ["executable", "program1", "program2"]
```

- ENTRYPOINT – This is used to execute a command or script as soon as the docker container is started but does not perform anything during the building of docker image. This instruction cannot be overridden.

```
ENTRYPOINT [<command>, <arguments>]
```

For example:

```
ENTRYPOINT ["/start.sh"]
```

Note that both CMD and ENTRYPOINT instructions define which command is executed but there are some ground rules to be followed while using these two instructions:

  - At least one of the CMD or ENTRYPOINT commands should be specified in the `Dockerfile`.
  - When using the container as an executable, ENTRYPOINT must be defined.
  - CMD should be used to specify default arguments for an ENTRYPOINT command or to run an ad hoc command in a container.
  - When running the container with alternative arguments, CMD will be overridden.

- EXPOSE - This is used by container to listen on a specific network port over TCP (default) or UDP protocol as required by application servers at runtime. It actually exposes the port of the application running inside the container to the outside of container.

```
EXPOSE <port_number>
```

```
EXPOSE <port_number>/<protocol>
```

For example:

```
EXPOSE 3030
EXPOSE 80/udp
```

- VOLUME - This is used to either create a mount point and assign a given name volume to hold externally mounted volumes or define a shared volume by mapping host path to the container path depending on number of arguments provided.

```
VOLUME [<new_volume_path>]
```

or

```
VOLUME [<host_path>] [<container_path>]
```

For example:

```
VOLUME ["/app/data"]
VOLUME ["/tmp/data" "/app/data/"]
```

- `WORKDIR` - This is used to set the working directory for other Dockerfile instructions. If the working directory does not exist already, this instruction will create it automatically.

```
WORKDIR <directory_path>
```

For example:

```
WORKDIR /app
```

- USER - This is used to set the user name (or UID) and optionally the group (or GID) to be used by the container when running the image. This is helpful when shared network directories are involved which need a fixed username or user ID.

```
USER <user_name or user_id>
```

For example:

```
USER example
USER 4000
```

- ARG - This is used to define a variable that can be used at build time using the `--build-arg` flag on the `docker build` command. This can be specified before FROM instruction and multiple ARG instructions are supported.

```
ARG <variable>=<value>
```

For example:

```
ARG image_name=latest
FROM centos:$image_name
docker build -t <image-name>:<tag> --build-arg image_name=centos8 .
```

- SHELL – This instruction overrides the default shell used for the shell form of commands. On **Linux**, the default shell is `[“/bin/sh”, “-c”]`, and on Windows, it is `[“cmd”, “/S”, “/C”]`. There can be multiple SHELL instructions but each SHELL instruction overrides all preceding SHELL instructions and has an impact on all subsequent instructions.

```
SHELL ["<executable>", "<parameters>"]
```

For example:

```
SHELL ["powershell", "-command"]
```

- HEALTHCHECK – This instructs Docker on how to test a container to ensure that it is still operational. This can detect cases such as a web server that is stuck in an infinite loop and unable to handle new connections, despite the server process still being active, etc. When a container is marked for health check, it becomes either healthy when a health check passes or unhealthy after a certain number of consecutive failures. Only one HEALTHCHECK instruction is allowed in a Dockerfile but subsequent HEALTHCHECK instructions will override the previous ones.

```
HEALTHCHECK <options> CMD <command>
```

Various HEALTHCHECK options are:

  - `--interval=<duration>` (default: 30s)
  - `--timeout=<duration>` (default: 30s)
  - `--start-period=<duration>` (default: 0s)
  - `--start-interval=<duration>` (default: 5s)
  - `--retries=<number>` (default: 3)

For example:

```
HEALTHCHECK --interval=5m --timeout=3s CMD curl -f
http://localhost/ || exit 1
```

When a container with a HEALTHCHECK instruction starts, it has an initial status `starting`. Then Docker runs the specified `command` at the given `interval` and if the `command` exits with 0, the container's status becomes healthy or remains healthy.

If the `command` runs for more than specified `timeout` seconds and exits with a non-zero status, the failure counter increments and if the failure counter reaches the specified `retries` value, the container's status becomes unhealthy. If a health check passes after a failure, the counter is reset to 0.

8 DOCKER IMAGE FROM DOCKERFILE

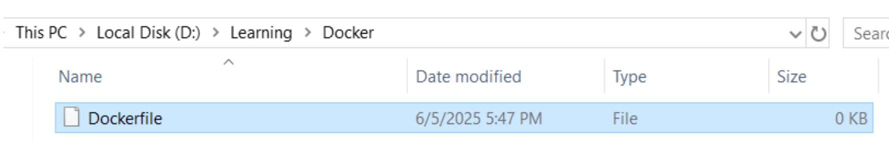
A Docker image can be created from a Dockerfile starting with the base image available in the Docker registry and the resulting image is structured in the form of layers.

Following are the four main stages of creating a Docker image from the Dockerfile:

1. Create a file named `Dockerfile`.
2. Add instructions in `Dockerfile`.
3. Build `Dockerfile` to create an image.
4. Run the image to create a container.

8.1 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.

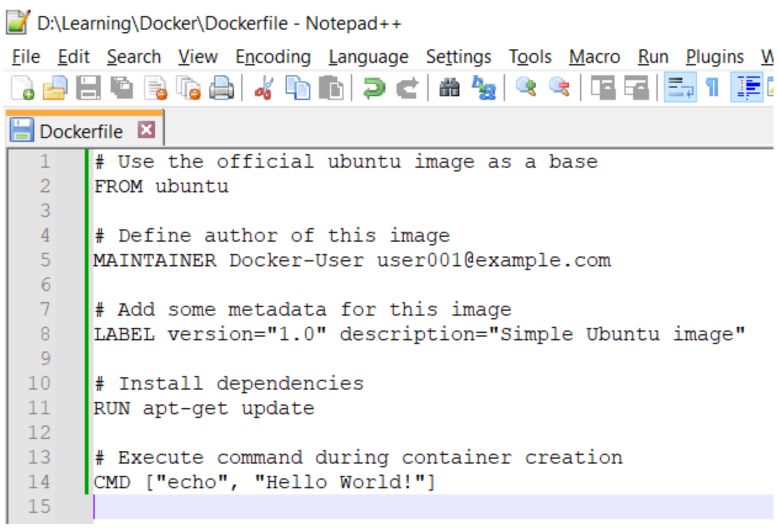
8.2 Add Instructions

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.



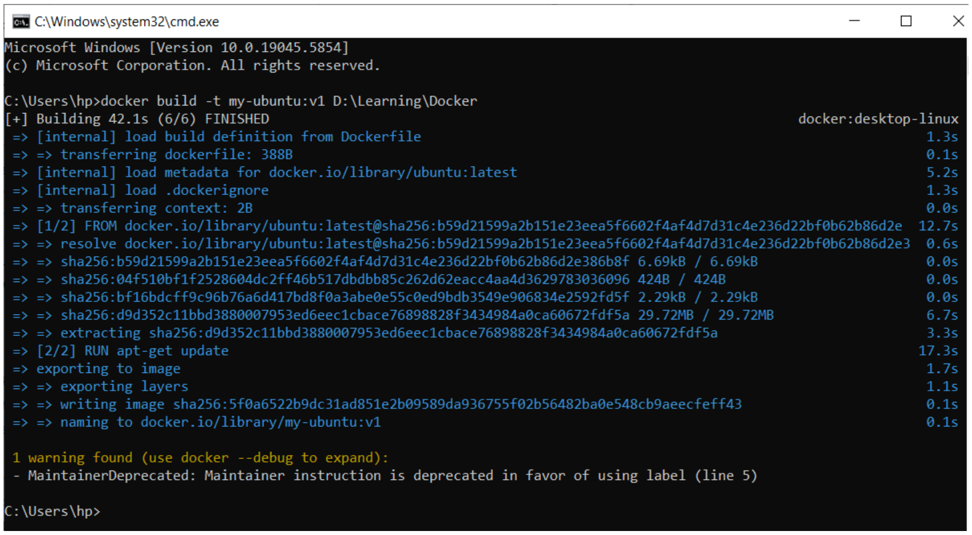
8.3 Build Docker Image

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.

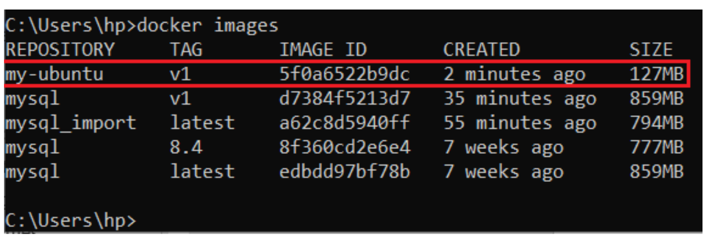
```
docker build -t my-ubuntu:v1 . -f Dockerfile.txt
```



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.19045.5854]
(c) Microsoft Corporation. All rights reserved.

C:\Users\hp>docker build -t my-ubuntu:v1 D:\Learning\Docker
[+] Building 42.1s (6/6) FINISHED                                                          docker:desktop-linux
 => [internal] load build definition from Dockerfile                                                       1.3s
 => => transferring dockerfile: 388B                                                                       0.1s
 => [internal] load metadata for docker.io/library/ubuntu:latest                                           5.2s
 => [internal] load .dockerignore                                                                          1.3s
 => => transferring context: 2B                                                                            0.0s
 => [1/2] FROM docker.io/library/ubuntu:latest@sha256:b59d21599a2b151e23eea5f6602f4af4d7d31c4e236d22bf0b62b86d2e  12.7s
 => => resolve docker.io/library/ubuntu:latest@sha256:b59d21599a2b151e23eea5f6602f4af4d7d31c4e236d22bf0b62b86d2e3  0.6s
 => => sha256:b59d21599a2b151e23eea5f6602f4af4d7d31c4e236d22bf0b62b86d2e386b8f 6.69kB / 6.69kB             0.0s
 => => sha256:04f510bf1f2528604dc2ff46b517dbdbb85c262d62eacc4aa4d3629783036096 424B / 424B                 0.0s
 => => sha256:bf16bdcff9c96b76a6d417bd8f0a3abe0e55c0ed9bdb3549e906834e2592fd5f 2.29kB / 2.29kB             0.0s
 => => sha256:d9d352c11bbd3880007953ed6eec1cbace76898828f3434984a0ca60672fdf5a 29.72MB / 29.72MB           6.7s
 => => extracting sha256:d9d352c11bbd3880007953ed6eec1cbace76898828f3434984a0ca60672fdf5a                  3.3s
 => [2/2] RUN apt-get update                                                                              17.3s
 => exporting to image                                                                                     1.7s
 => => exporting layers                                                                                    1.1s
 => => writing image sha256:5f0a6522b9dc31ad851e2b09589da936755f02b56482ba0e548cb9aeecfeff43               0.1s
 => => naming to docker.io/library/my-ubuntu:v1                                                            0.1s

 1 warning found (use docker --debug to expand):
 - MaintainerDeprecated: Maintainer instruction is deprecated in favor of using label (line 5)

C:\Users\hp>
```

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

```
docker images
```



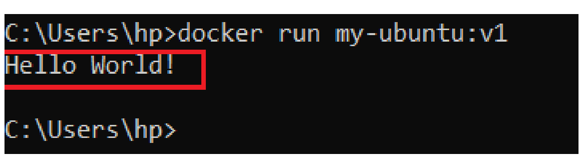
```
C:\Users\hp>docker images
REPOSITORY     TAG       IMAGE ID       CREATED          SIZE
my-ubuntu      v1        5f0a6522b9dc   2 minutes ago    127MB
mysql          v1        d7384f5213d7   35 minutes ago   859MB
mysql_import   latest    a62c8d5940ff   55 minutes ago   794MB
mysql          8.4       8f360cd2e6e4   7 weeks ago      777MB
mysql          latest    edbdd97bf78b   7 weeks ago      859MB

C:\Users\hp>
```

8.4 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
```



```
C:\Users\hp>docker run my-ubuntu:v1
Hello World!

C:\Users\hp>
```

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>
```

9 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.

Key features of Docker Desktop:

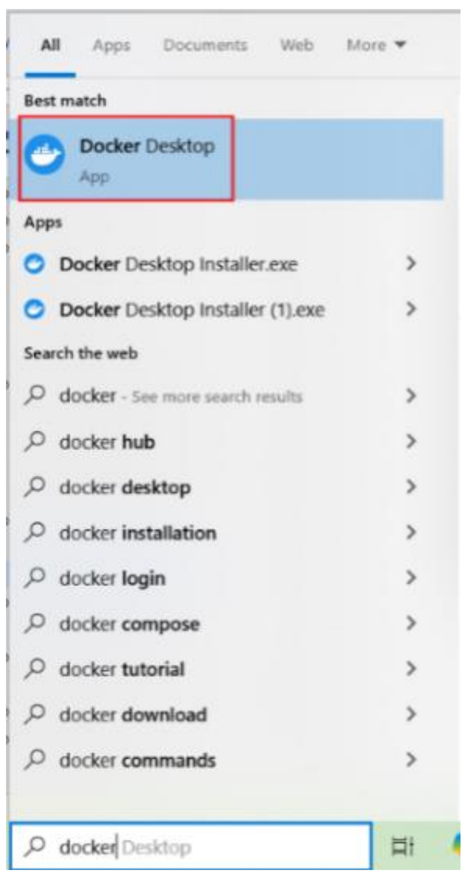
- Ability to containerize and share any application on any cloud platform in multiple languages and frameworks.
- Quick installation and setup of a complete Docker development environment.
- Includes the latest version of Kubernetes.
- Supports both Windows and macOS operating systems, leveraging native virtualization technologies (Hyper-V on Windows, HyperKit on macOS) to run Linux containers.
- On Windows, the ability to toggle between Linux and Windows containers to build applications.
- Fast and reliable performance with native Windows Hyper-V virtualization.
- Ability to work natively on Linux through WSL 2 on Windows machines.
- Volume mounting for code and data, including file change notifications and easy access to running containers on the localhost network.

- Seamless integration with existing development workflows and tools including IDEs, version control systems and CI/CD pipelines.
- Offers configurable options for managing resources such as CPU, memory, and disk space allocated to Docker containers and virtual machines.

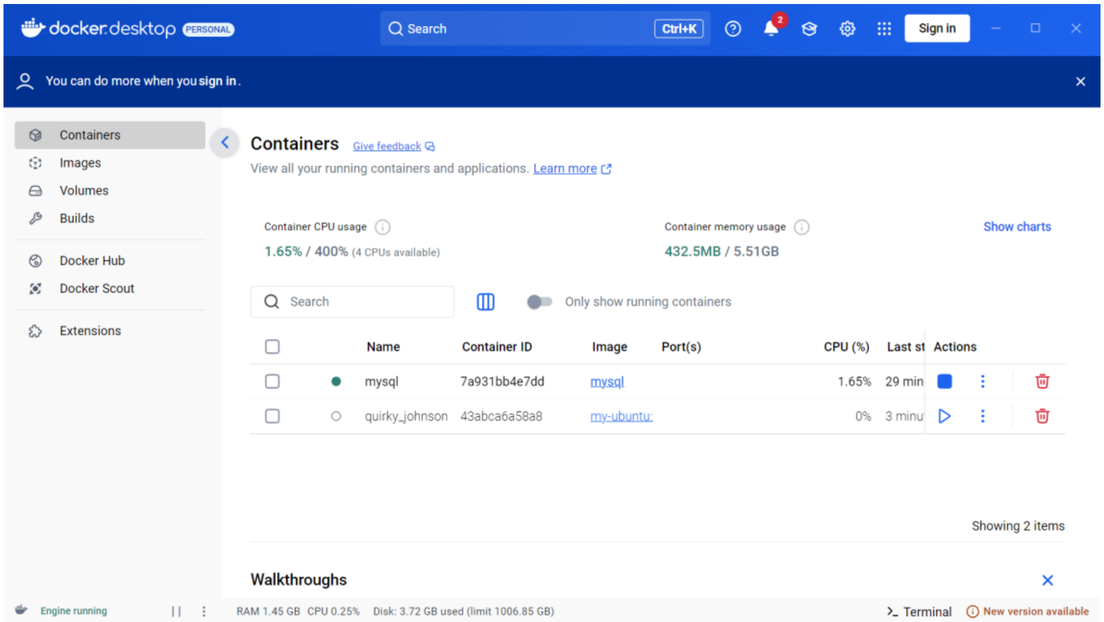
Docker Desktop includes components such as **Docker Engine**, **Docker CLI**, **Docker Build**, **Docker Compose**, **Docker Extensions**, **Docker Content Trust**, **Kubernetes**, **Credentials Helper** and **Ask Gordon** *(a personal AI assistant)*.

9.1 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:

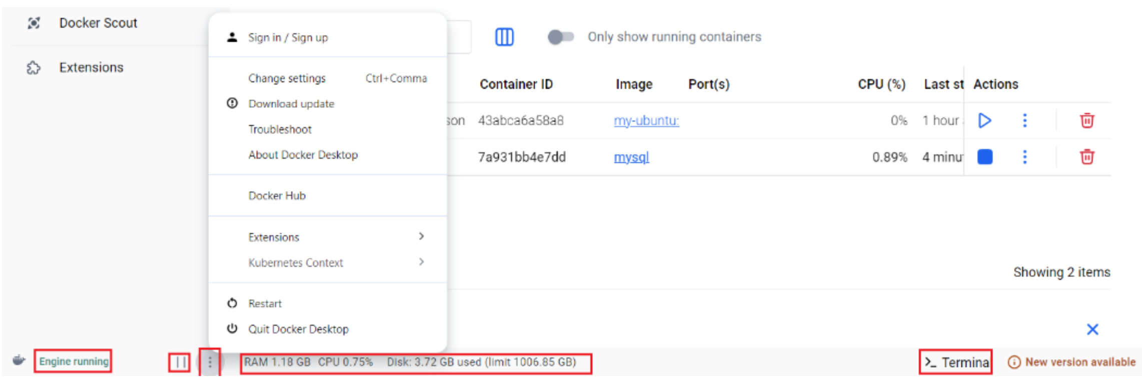
- **Quick Search** icon to search for local containers, local images, public Docker Hub images, images from remote repositories, extensions to install or open, volumes and official Docker documentation.
- **Notifications** icon to get notified of new releases and installation progress updates,
- **Troubleshoot** 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.

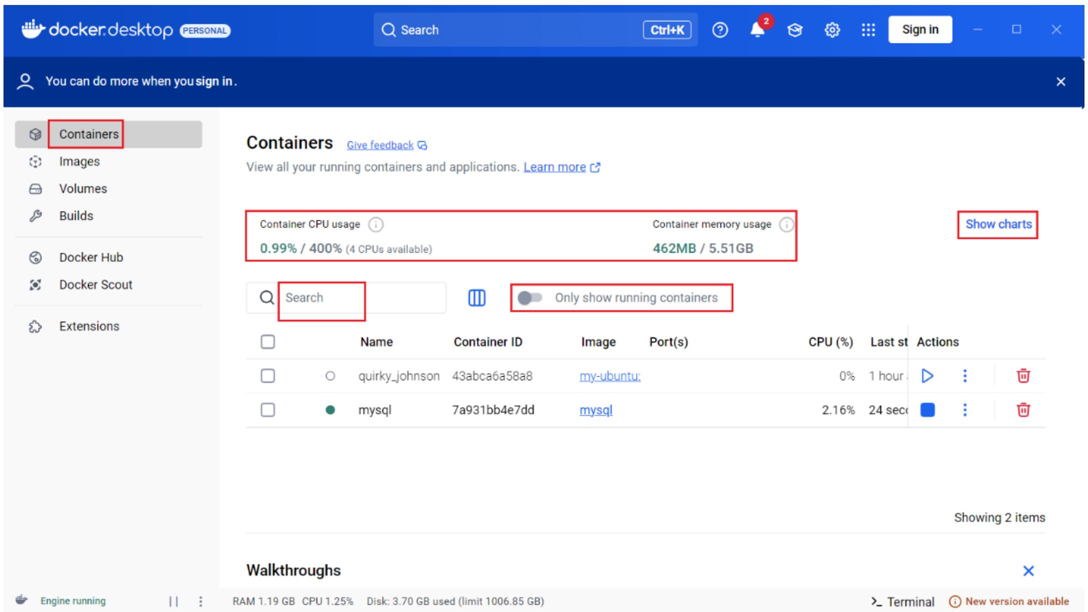


9.2 Containers View

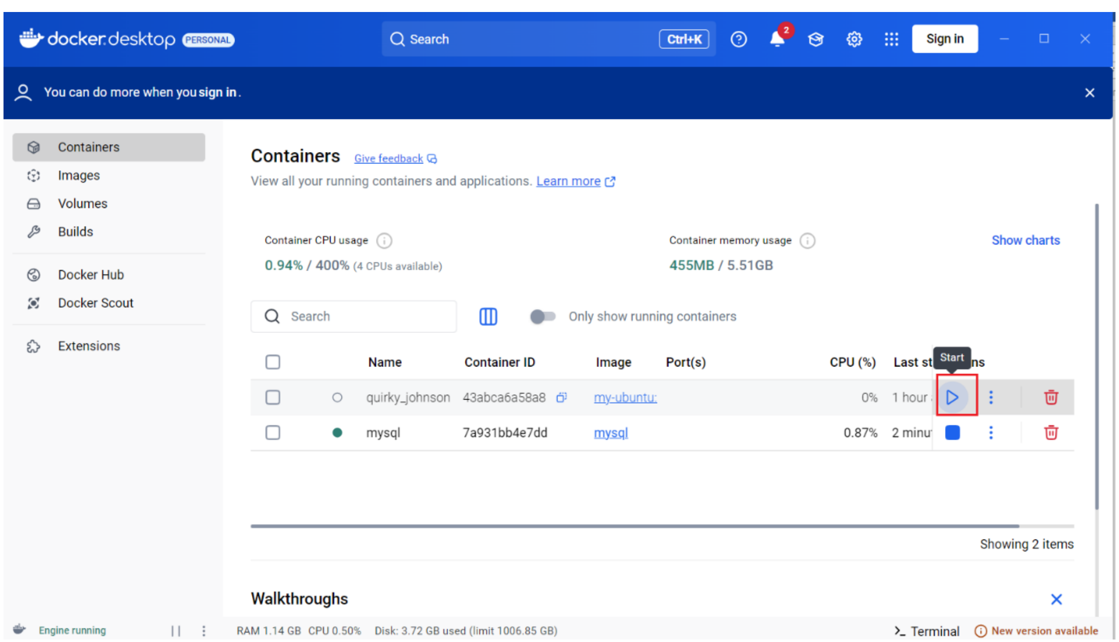
The **Containers** view in Docker Desktop provides a graphical interface for managing the life cycle of containers. It lists all running and stopped containers and provides a runtime view of all containers and applications. It allows to inspect, interact with and manage Docker objects including containers 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

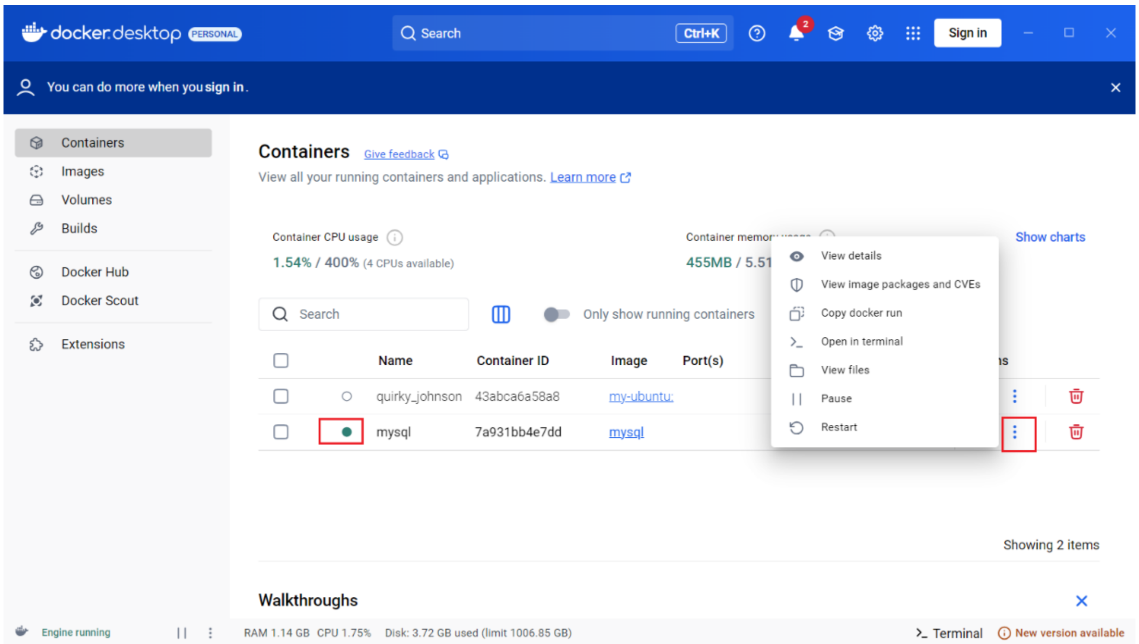
Containers status bar displays the CPU and memory usage of all running containers which can also be viewed in graphical format by choosing **Show charts** option and the **Search** field allows to search for a specific container.



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.

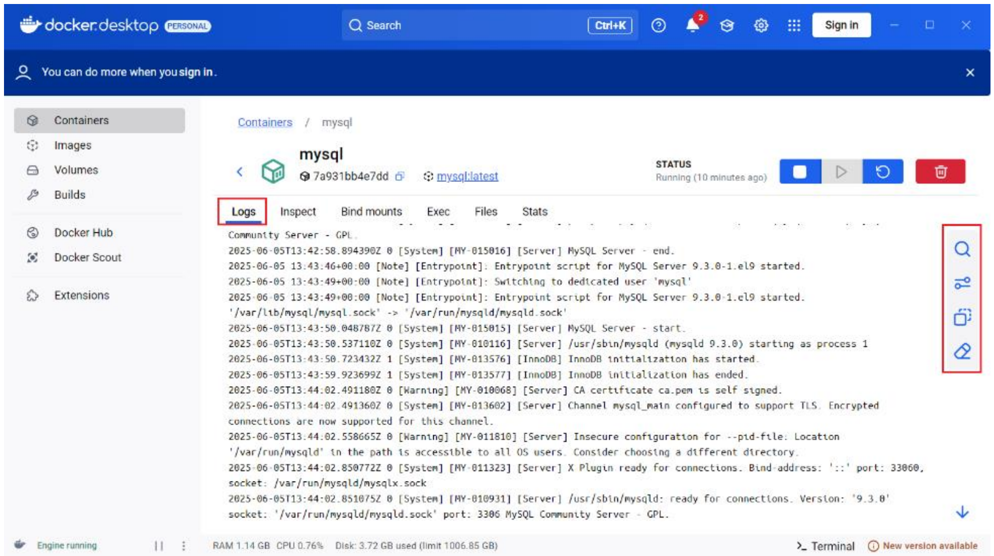


Click on `mysql` container that is running or select **View details** option under **Show Containers Actions** icon to inspect the container which displays various tabs:

- **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>
  ```

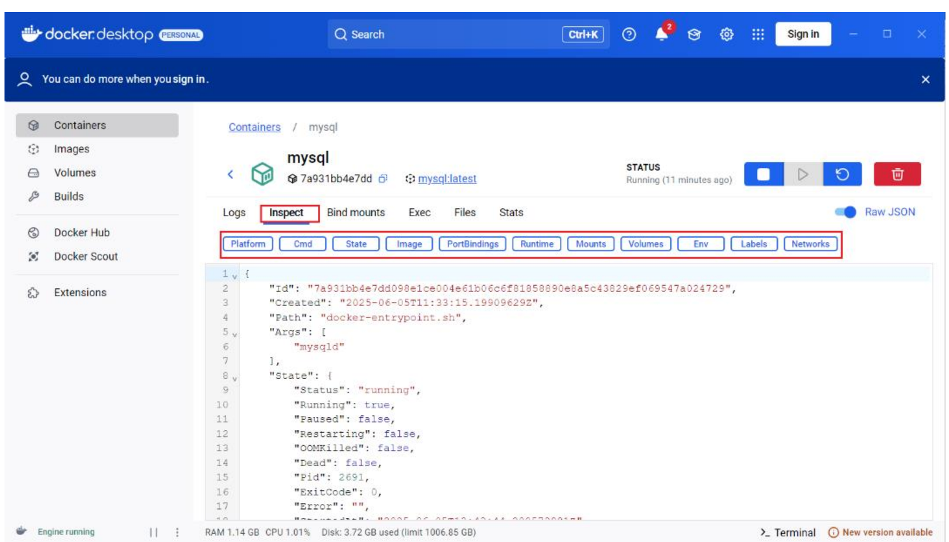
  The **Logs** tab provides various log options such as **Search terminal** to open specific log entries, **Settings** to define how log entries can be displayed, **Copy** to copy all logs to clipboard, **Clear terminal** to clear the logs terminal.



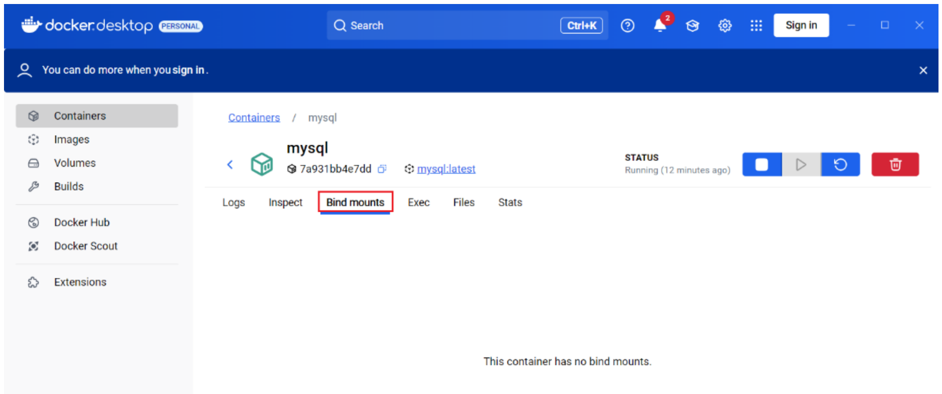
- **Inspect** tab allows to view the low-level information about the container including local path, version number of the image, port mapping, and other details. This tab is the same as running the following command:

```
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.



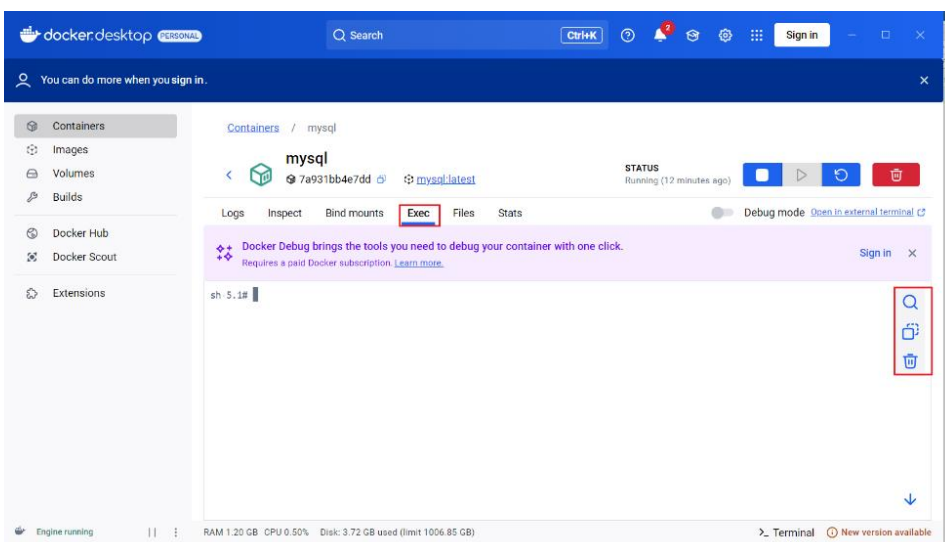
- **Exec** tab allows to quickly run commands within the running container. This tab is the same as running one of the following commands:

When accessing Linux containers:

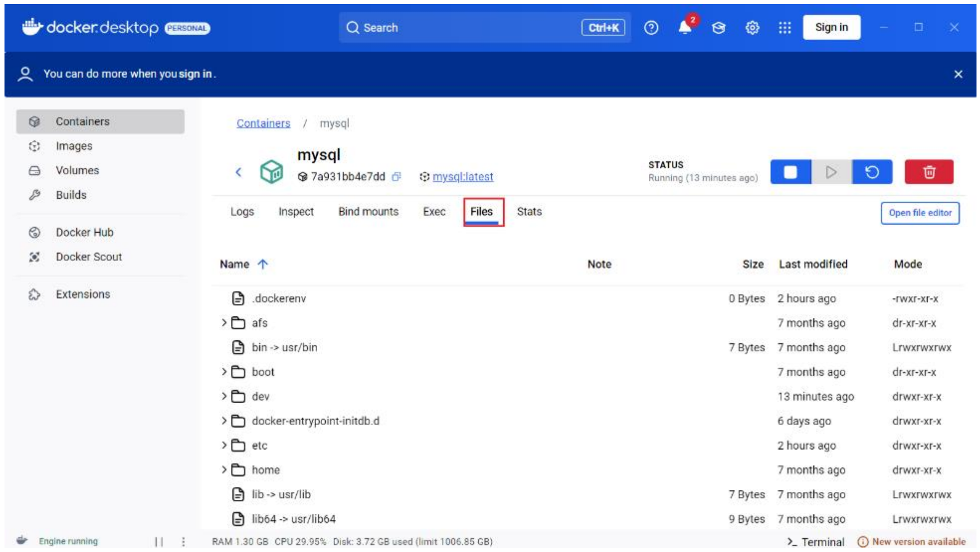
```
docker exec -it <container-id> /bin/sh
```

When accessing Windows containers:

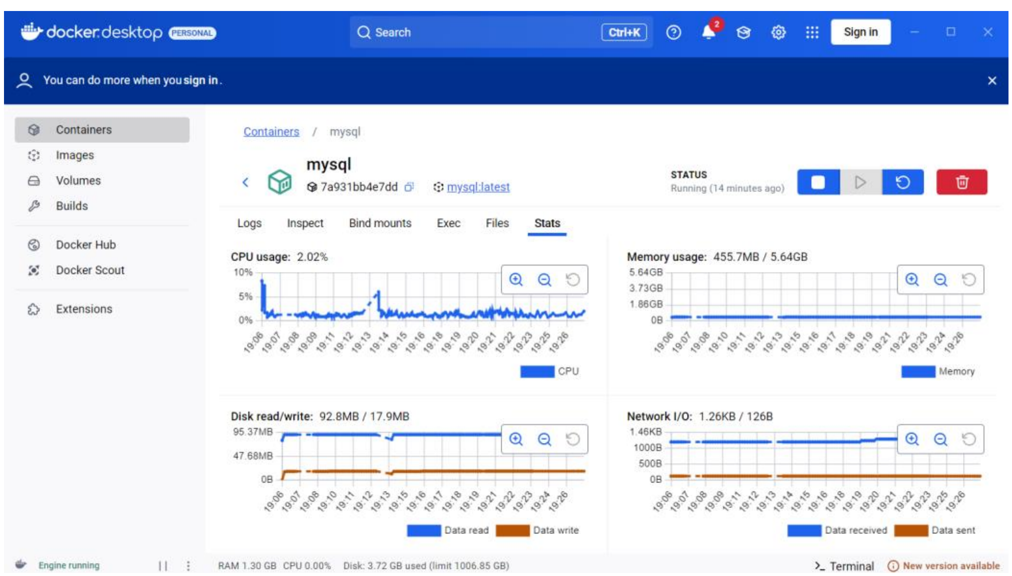
```
docker exec -it <container-id> cmd.exe
```



- **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.



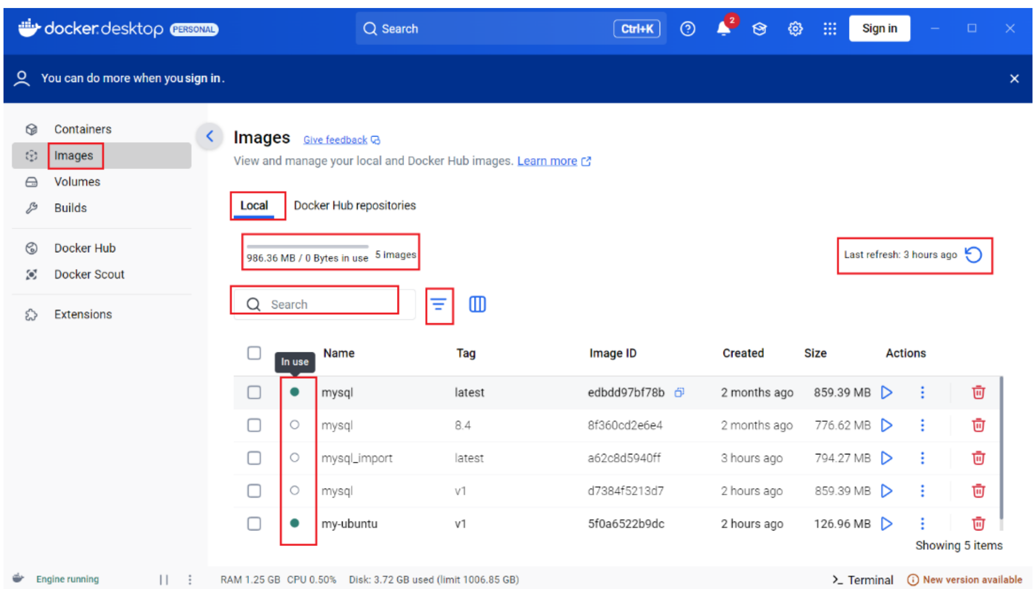
- **Stats** tab allows to display container's resource utilization and monitor container's CPU, memory, disk space and network usage over time in a graphical format.



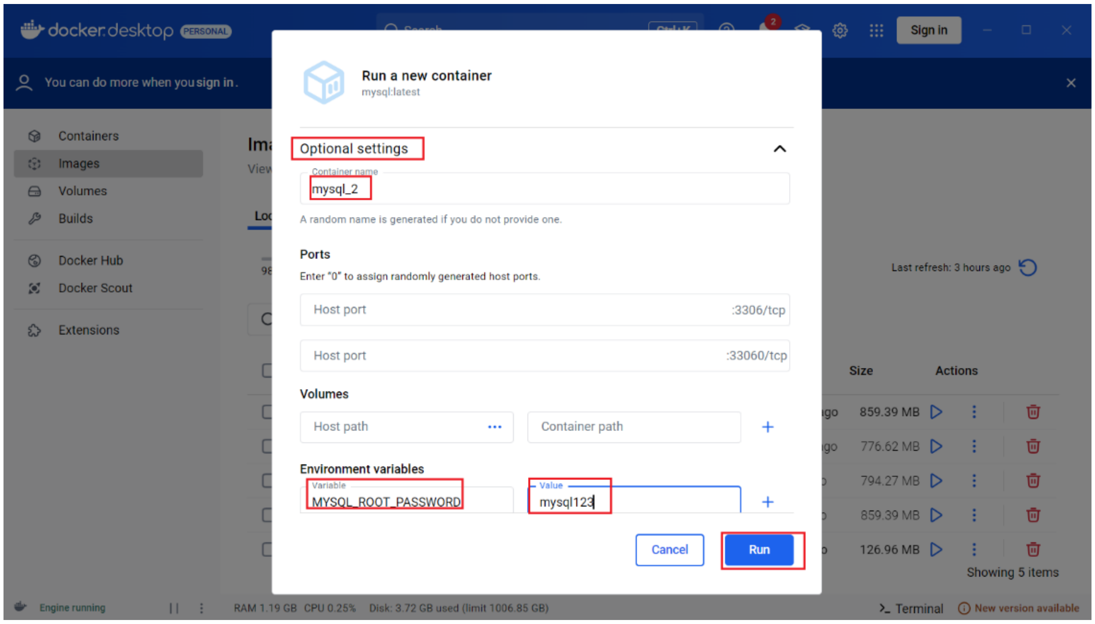
9.3 Images View

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.

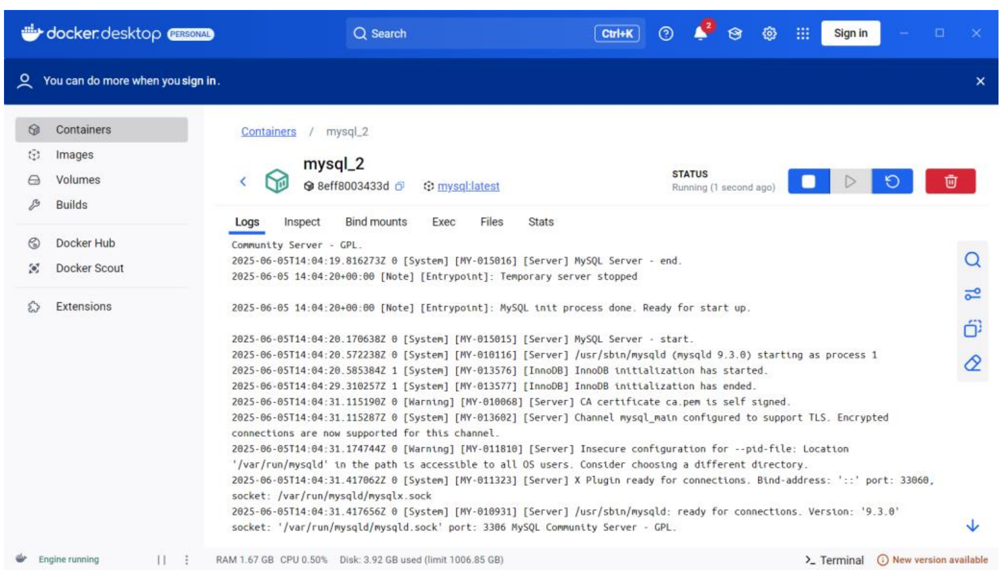
All docker images that are either built locally and pulled from the Docker repository are visible under **Local** tab. The **In Use** option next to the image name helps to identify if the image is in used by the container or unused. The **Images status** bar displays the number of images and total disk 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*.



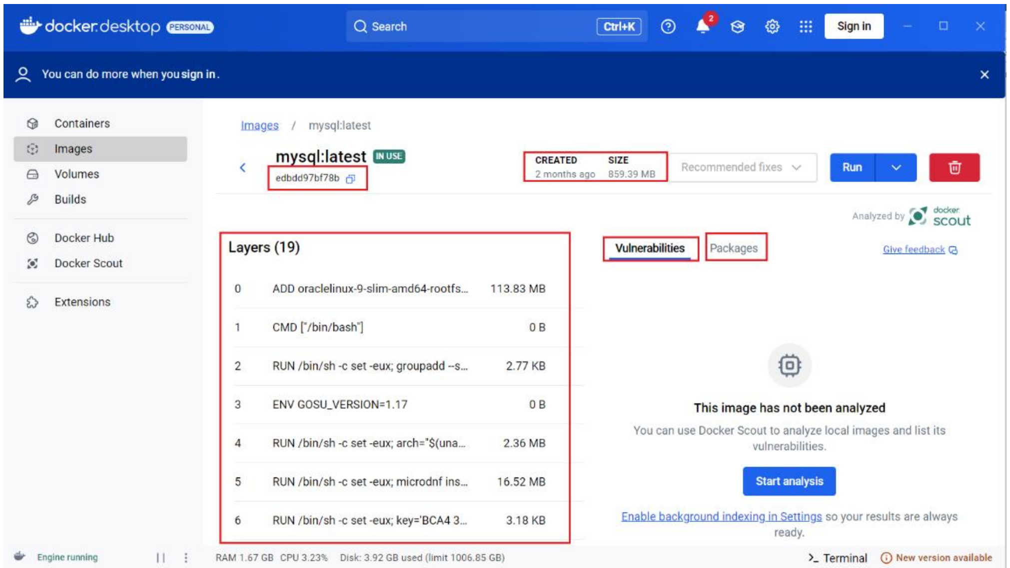
To run an image as a container, select the image, in this example it is `mysql`, and click on **Run** icon under **Actions** menu. When prompted, select the **Optional settings** drop-down and specify the container name as `mysql_2` and add the **Environment variable** as `MYSQL_ROOT_PASSWORD` and provide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



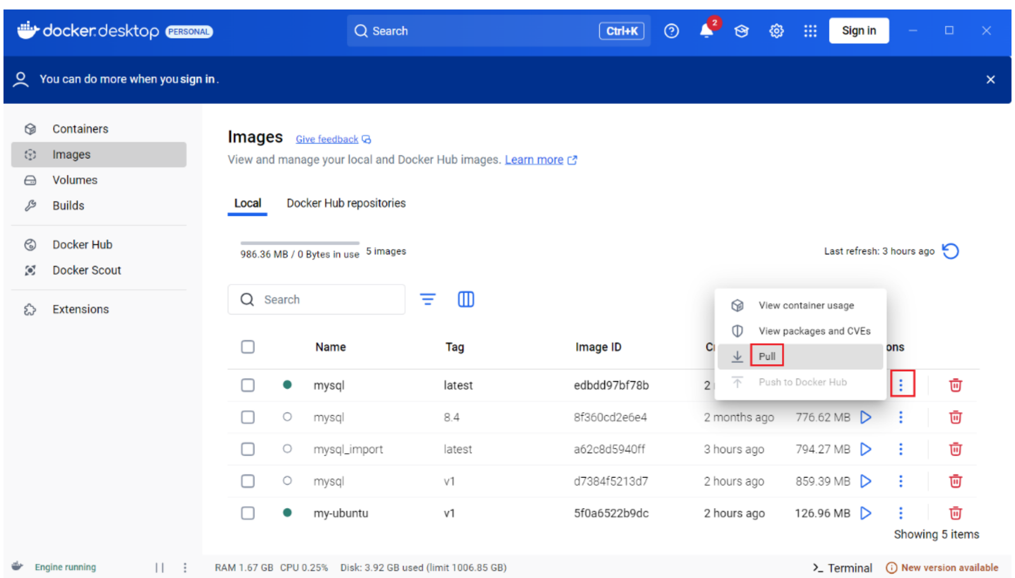
It then creates a new container `mysql_2` and displays log entries



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.

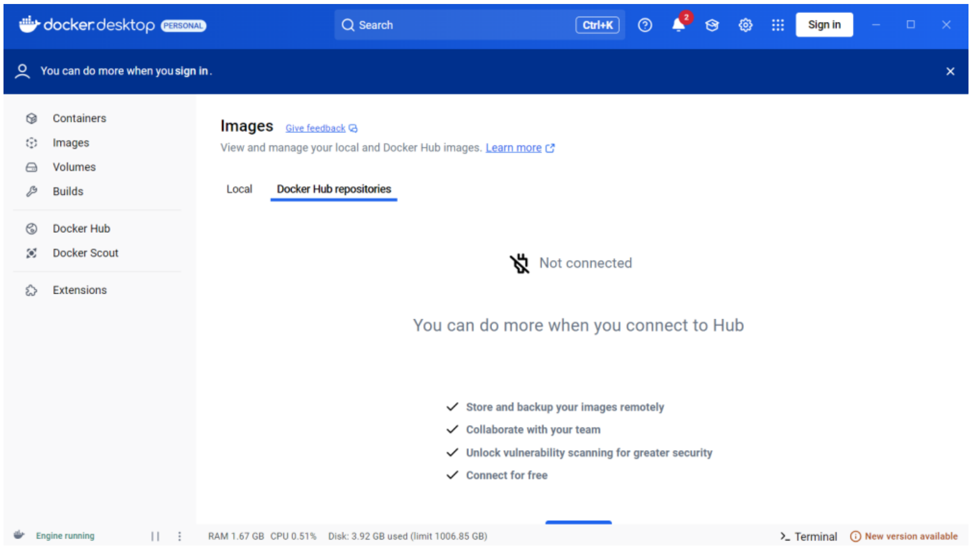


To pull the latest version of the image from Docker Hub, select the image and click on **Show image actions** icon and select **Pull**. Note that the repository must exist on Docker Hub in order to pull the latest version of an image.

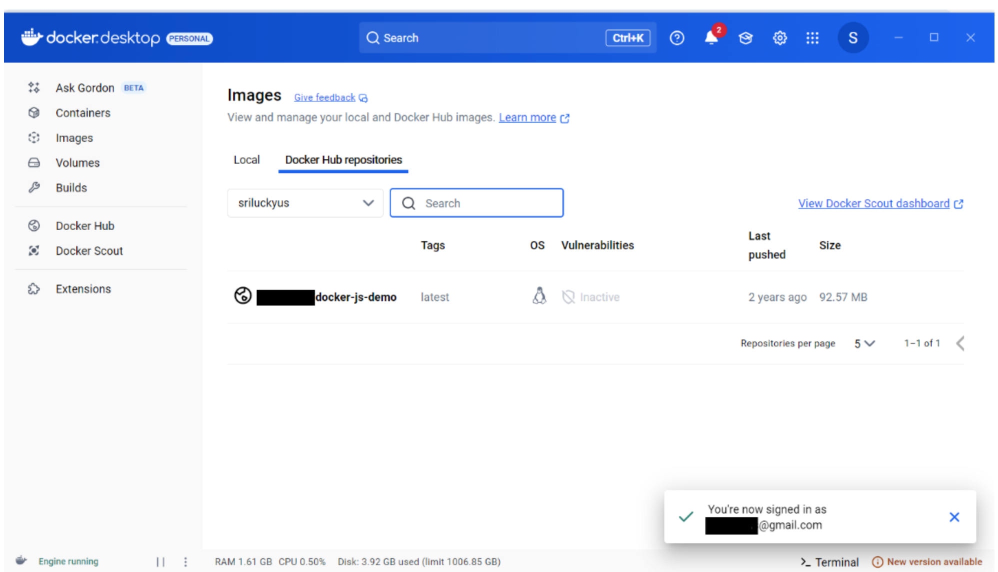


To push the image to Docker Hub, select the image and click on **Show image actions** icon and select **Push to Docker Hub**. Note that the Docker Hub has already been logged in and can push the image if it contains the correct username/organization in its tag.

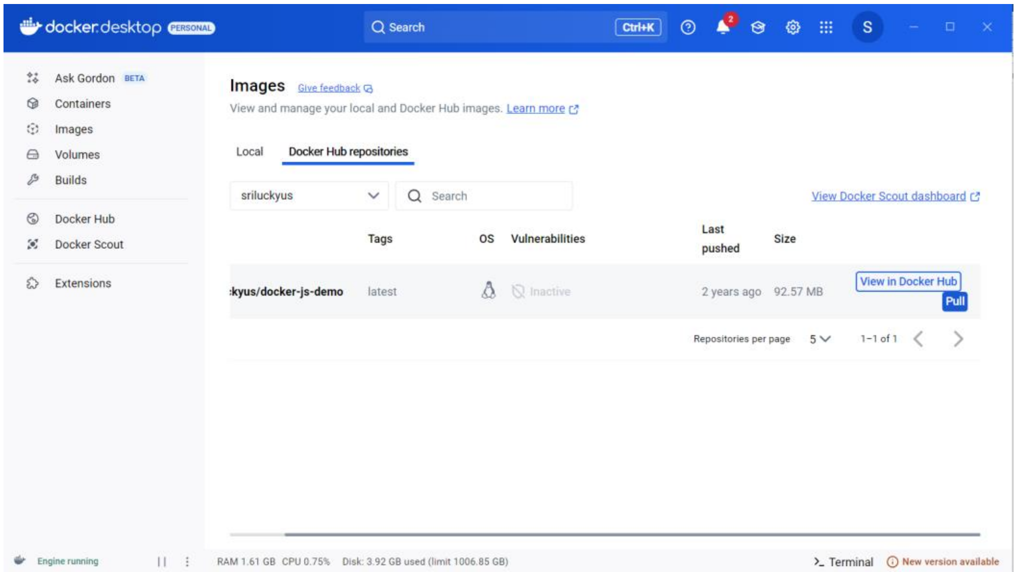
The **Images** view also allows to manage and interact with images in Docker Hub repositories. Click on **Docker Hub repositories** tab which prompts to sign in to Docker Hub account, if not already signed in.



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.



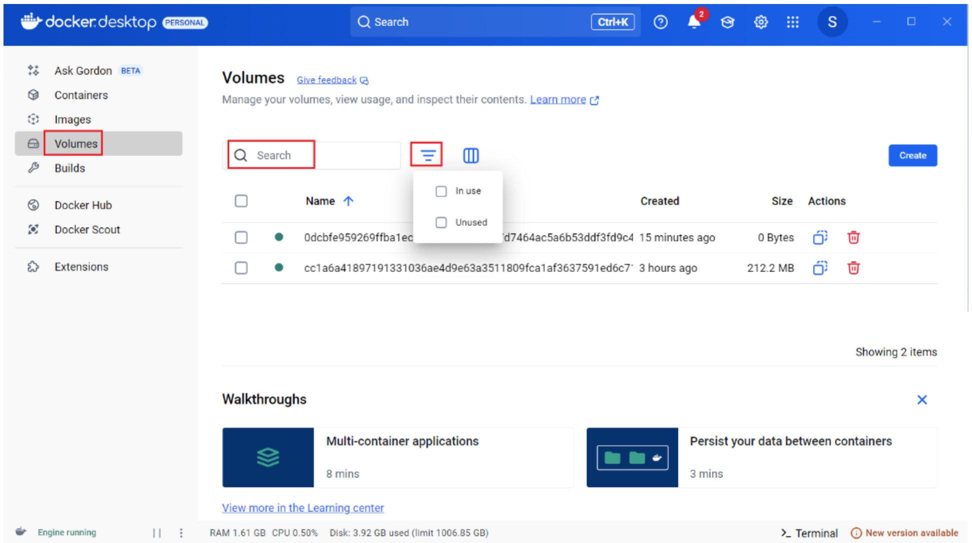
If **Docker Scout** is enabled on the repositories, it displays image analysis results and health scores next to the image tags. Hover over the image tag which displays **Pull** option pull the latest version of the image from Docker Hub and **View in Hub** to open the Docker Hub page and display detailed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



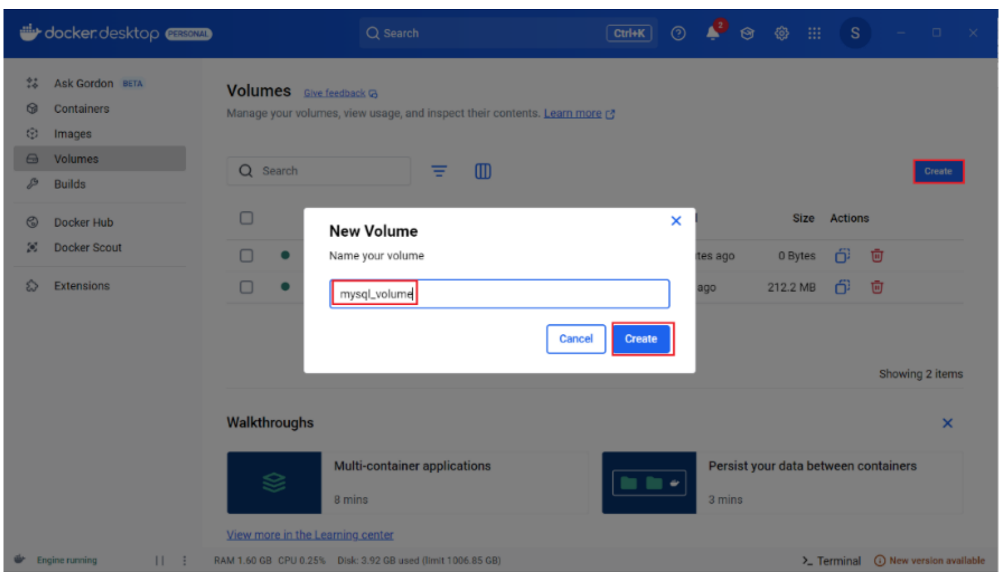
9.4 Volumes View

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*.

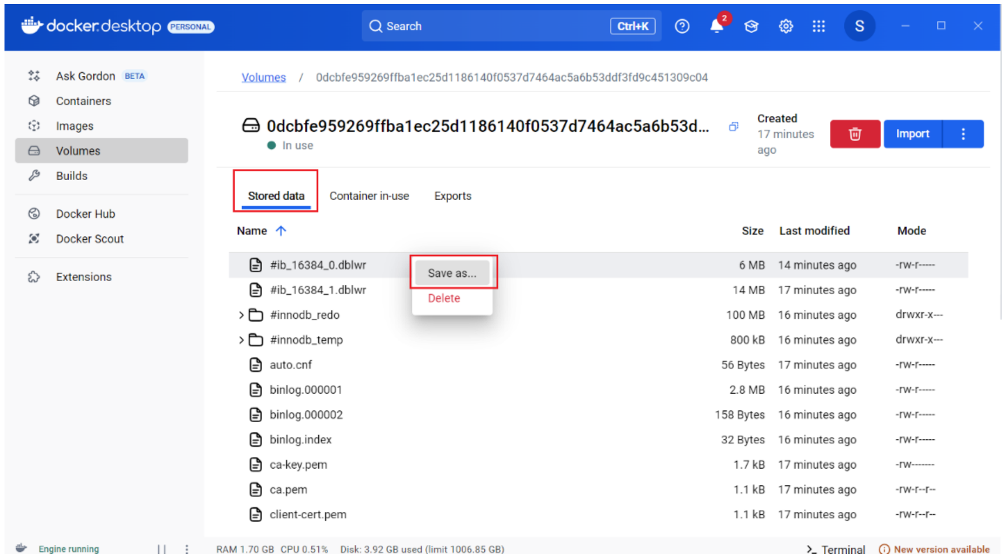


To create a volume, select **Create** button and specify the volume name (let's say `mysql_volume`) and click on **Create** in **New Volume** modal.

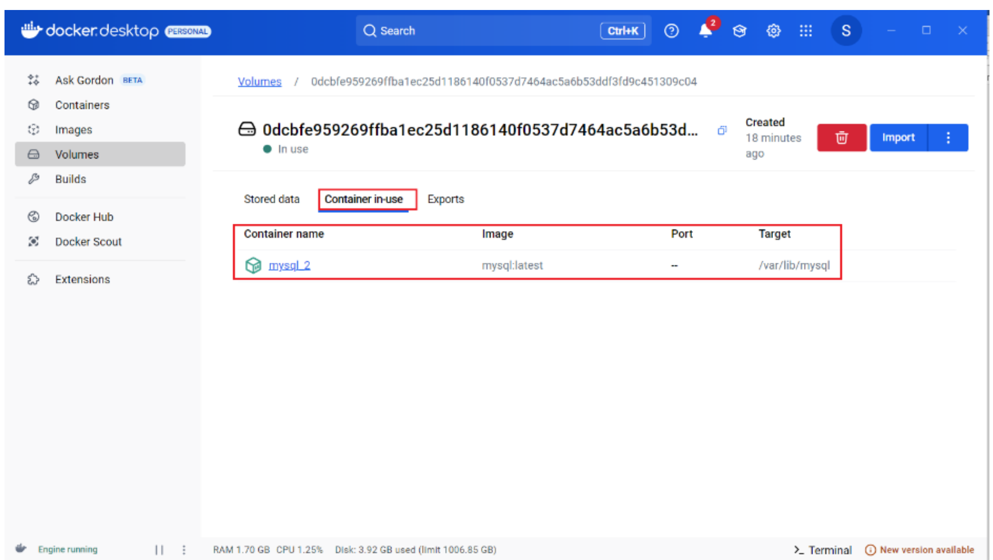


To inspect a volume, click on the volume to display detailed information about the volume such as:

- 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.

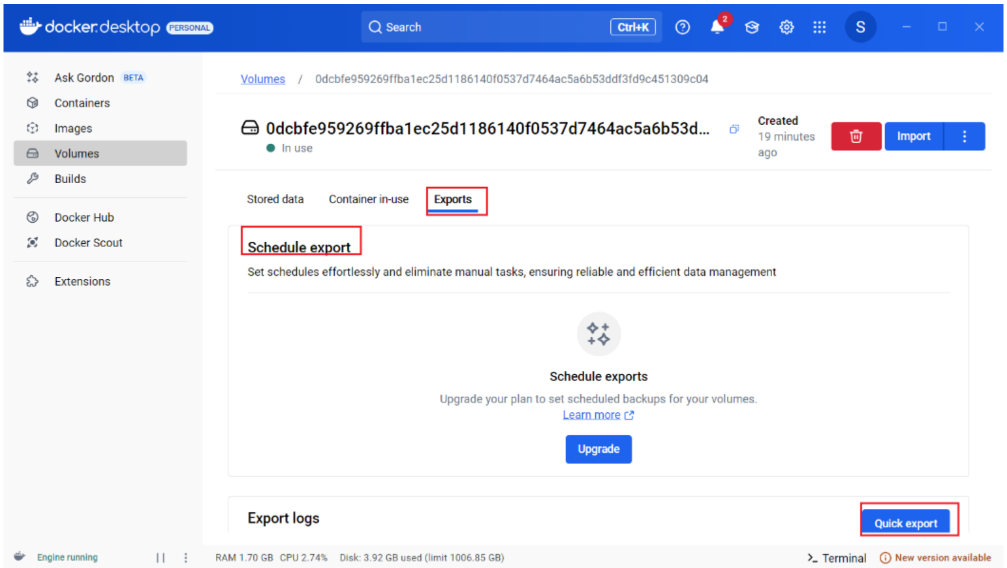


- **Container in-use** tab which displays the name of the container using the volume, the image name, the port number used by the container, and the target. A target is a path inside a container that gives access to the files in the volume.

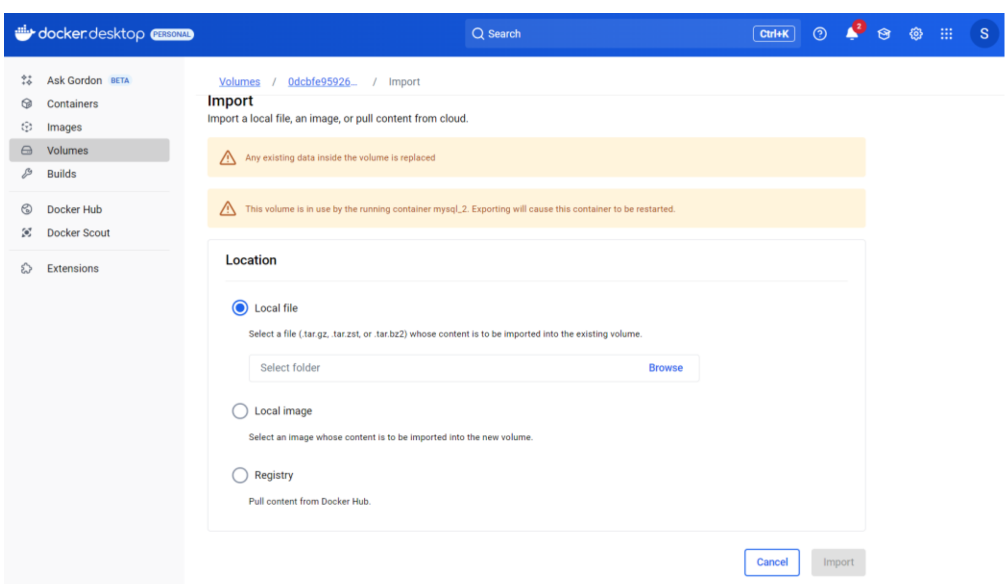


- **Export** tab which exports the content of a volume to a local file, a local image, and to an image in Docker Hub, or to a supported cloud provider. When exporting content from a volume used by one or more running containers, the containers are temporarily stopped while Docker exports the content, 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.

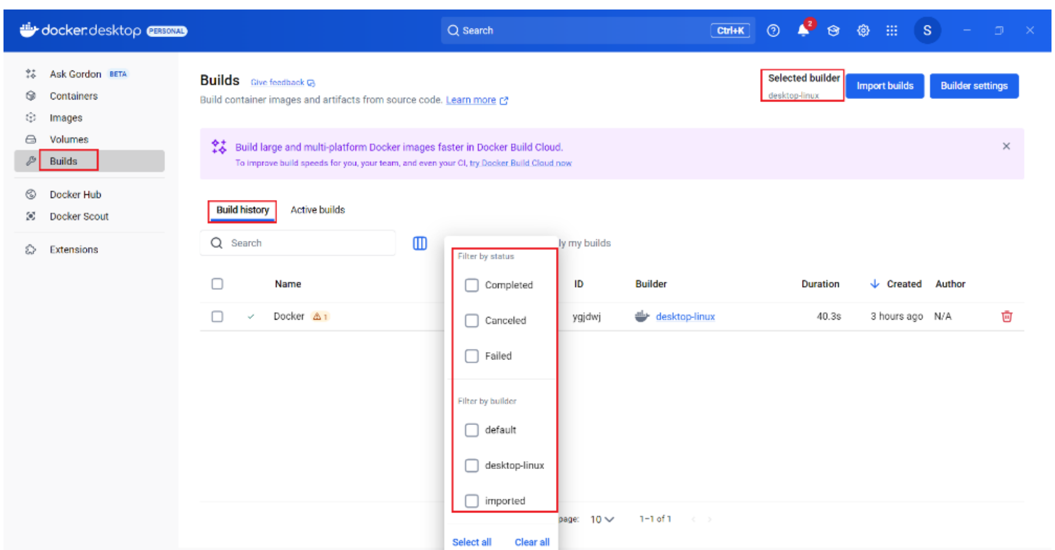


9.5 Builds View

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



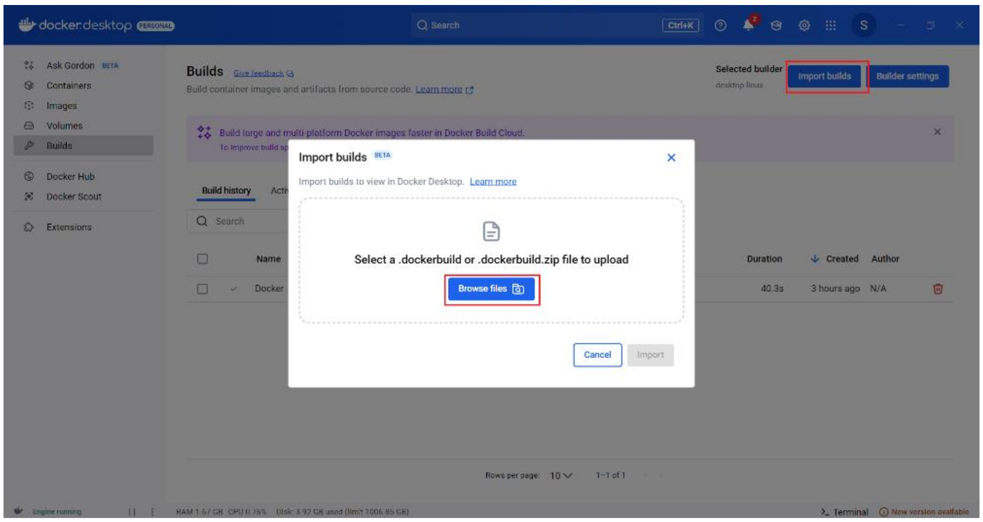
Note that when building Windows container images using the `docker build` command, the legacy builder is used which does not populate the Builds view. Instead, use either of the build commands to use **BuildKit** which populates the Builds view:

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

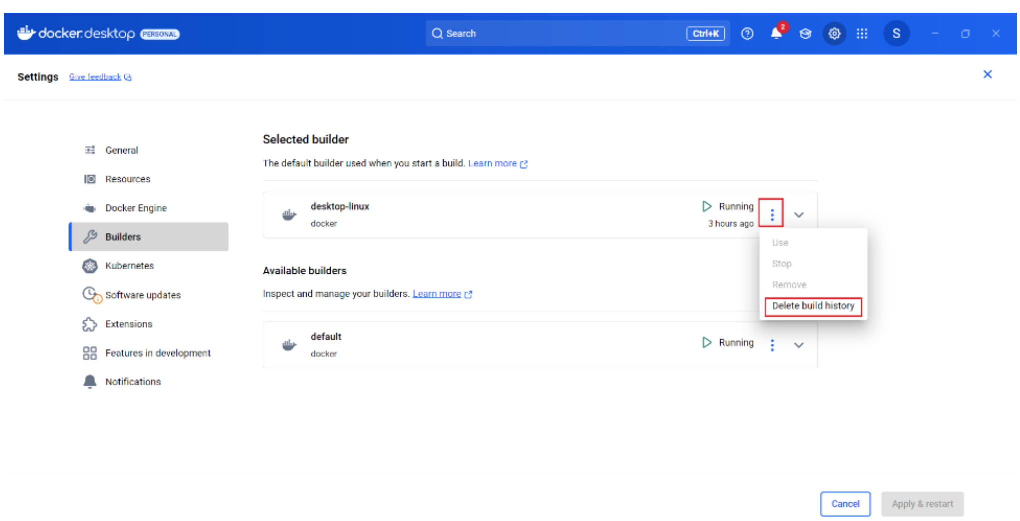
or

```
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.

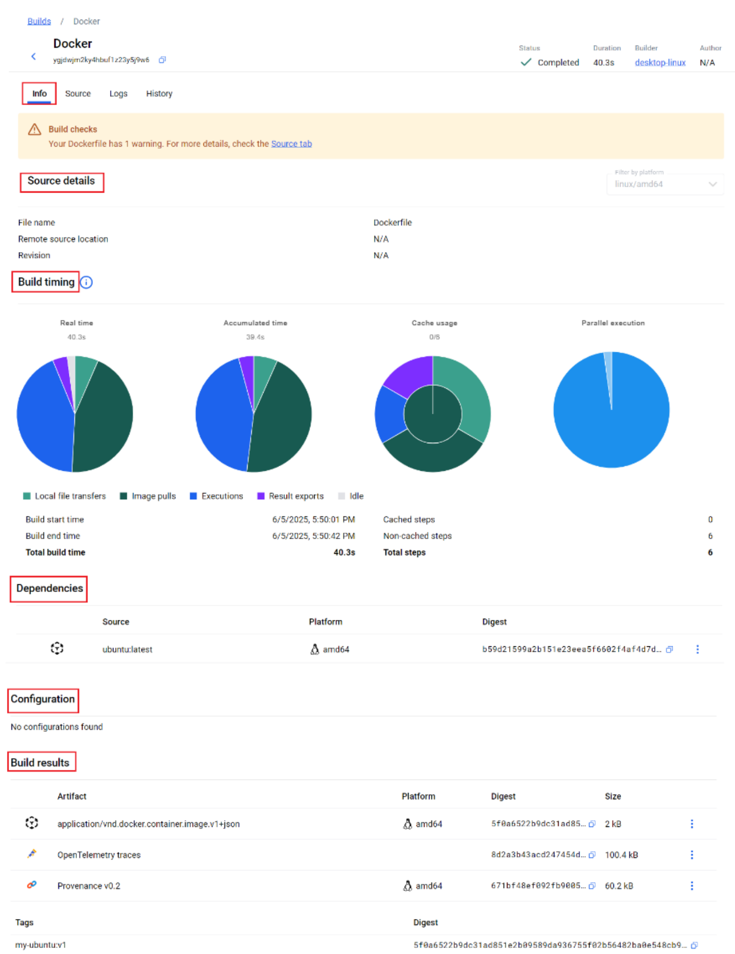


Click on **Builder settings** button to manage builders including inspecting the state and configuration of active builders, starting and stopping a builder, deleting the build history, adding or removing builders, connecting and disconnecting cloud builders directly from the Docker Desktop settings.

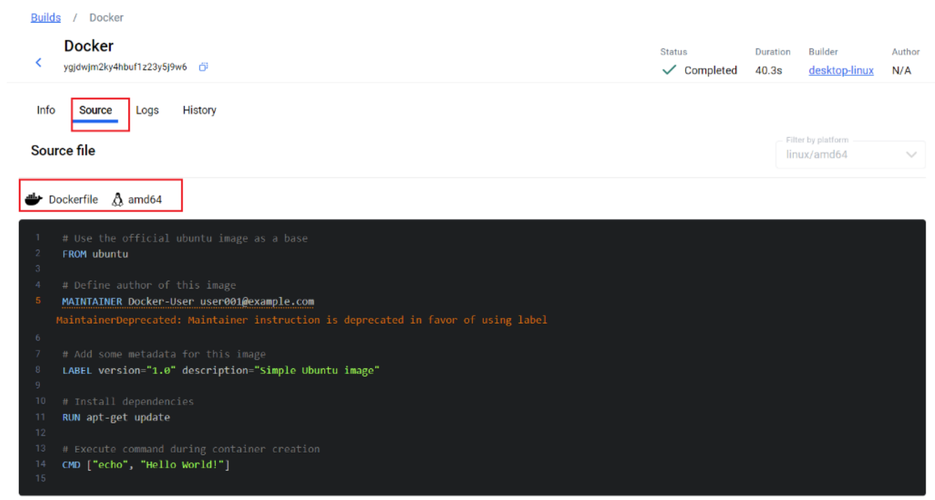


To inspect a build, click on the build name to display detailed information about the build such as:

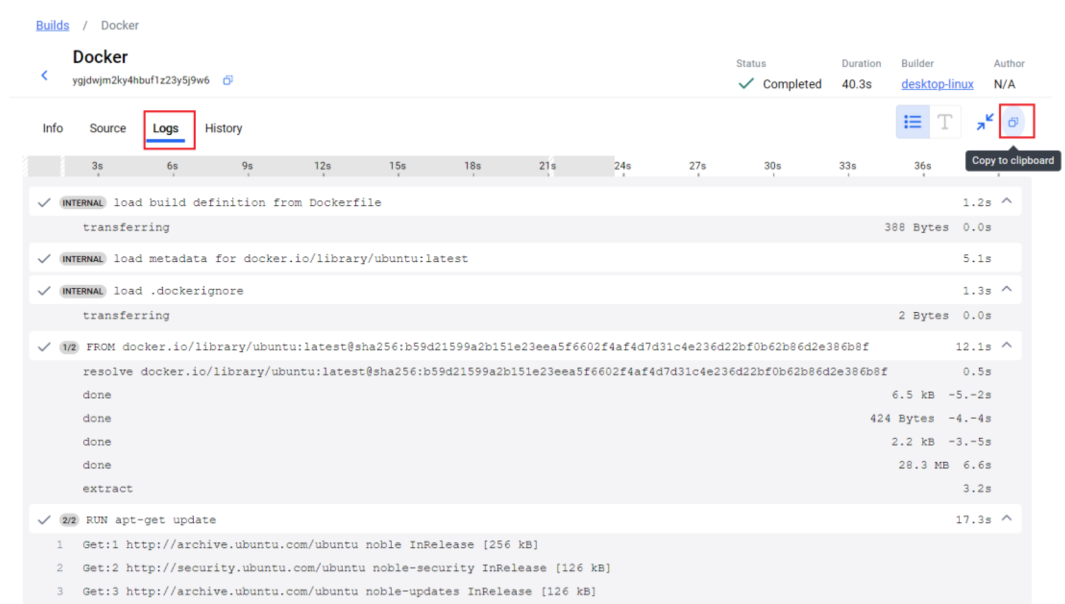
- **Info** tab which displays details about the build with various sections including
    - **Source Details** to display information about the frontend and, if available, the source code repository used for the build.
    - **Build timing** to displays charts showing a breakdown of the build execution 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.



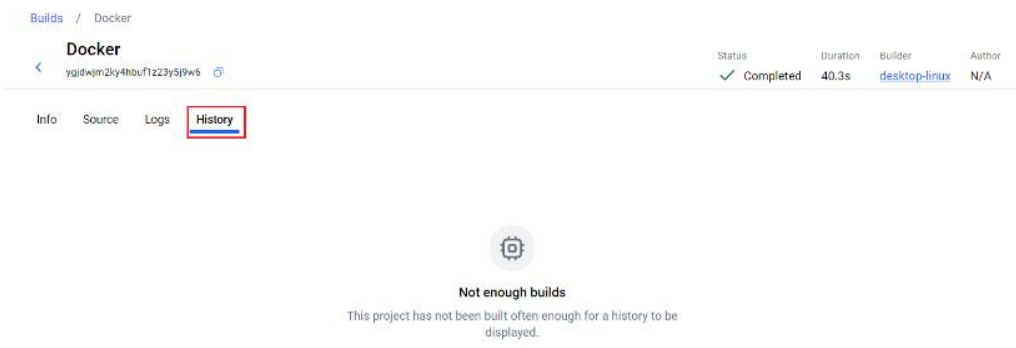
- **Sources** tab which displays the frontend used to create a build. If the build fails, an **Error** tab displays instead of the Source tab and the error message is inlined in the Dockerfile source, indicating where and why the failure happened.



- **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.

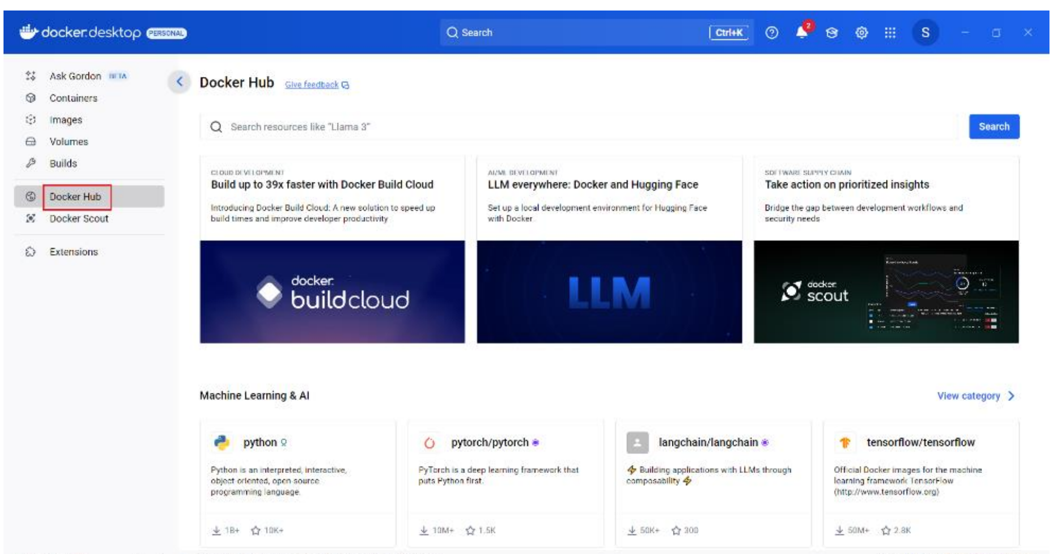


- **History** tab which displays statistics data about completed builds.



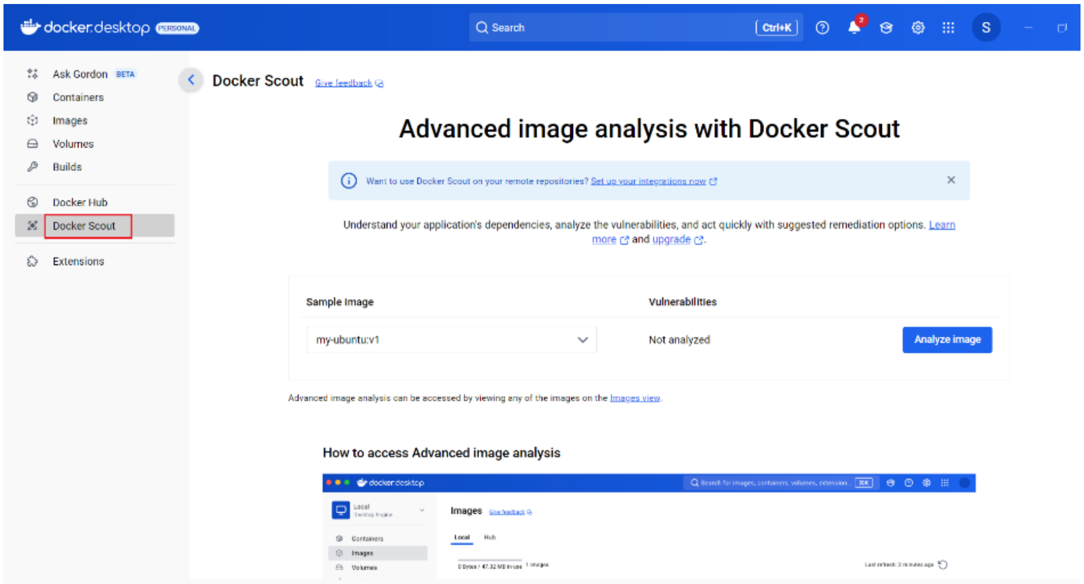
9.6 Docker Hub View

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



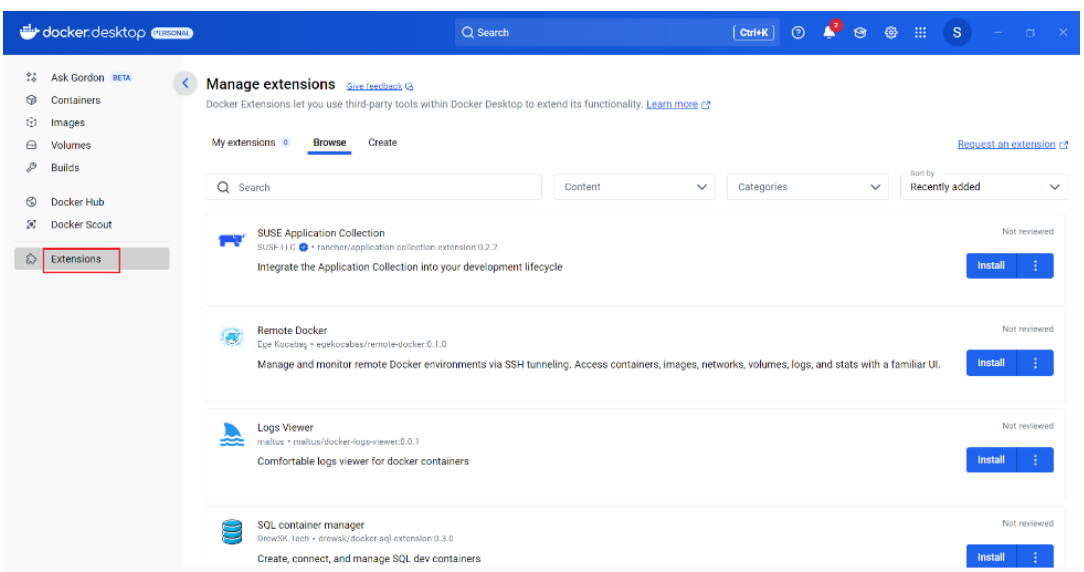
9.7 Docker Scout View

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.



9.8 Docker Extensions View

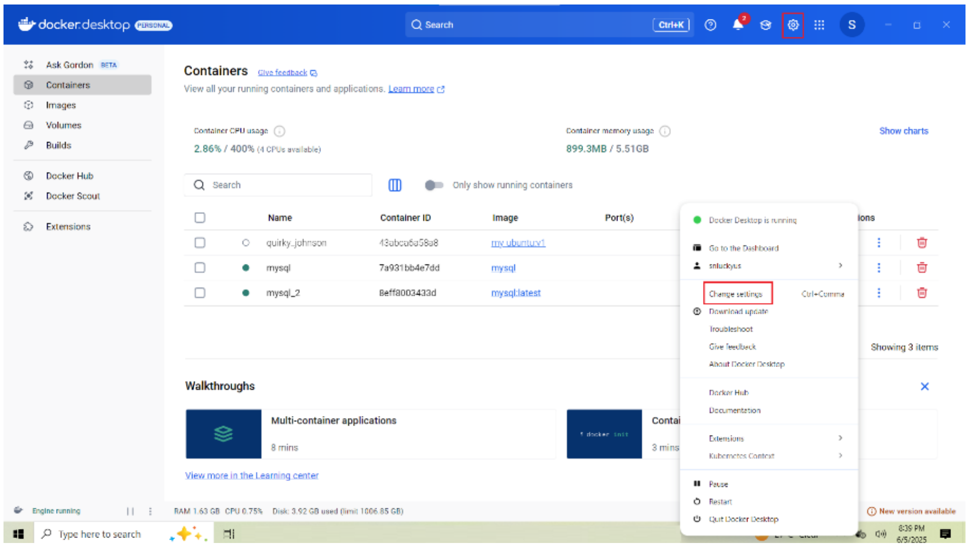
The **Extensions** view in Docker Desktop allows to use integrate third-party tools seamlessly in Docker Desktop for application development and deployment of workflows. It allows to explore the list of available extensions in Docker Hub or in the Extensions Marketplace within Docker Desktop.



9.9 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.

10 CREATE PYTHON DOCKER APPLICATION

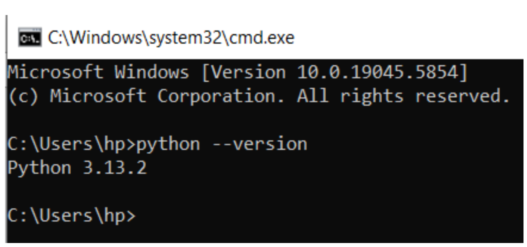
We will create a simple **Python Flask** containerized application to run from Docker. **Flask** (`flask` library) is a Python framework used to create a web application.

10.1 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
```



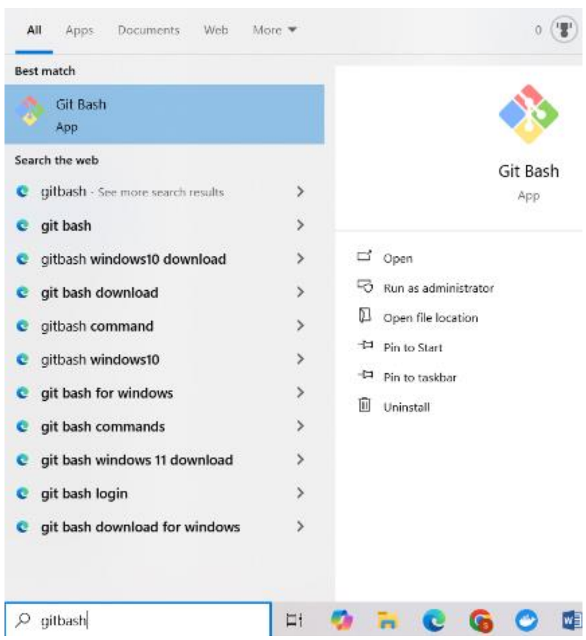
In my system, **Python 3.13.0** version is installed.

10.2 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.

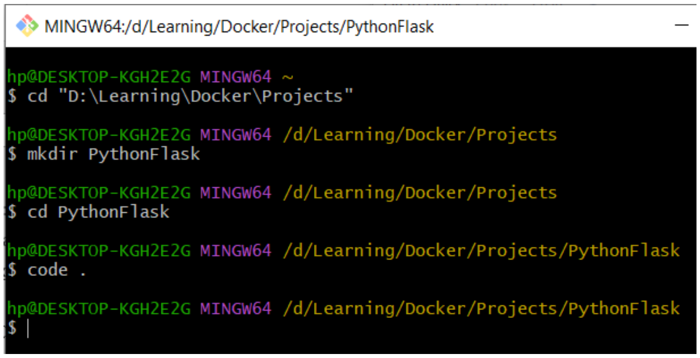
Though VS Code can be opened directly, we will use **Git** software to launch VS Code. Install **Git** software on Windows from the [Git website] if you do not have it.

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

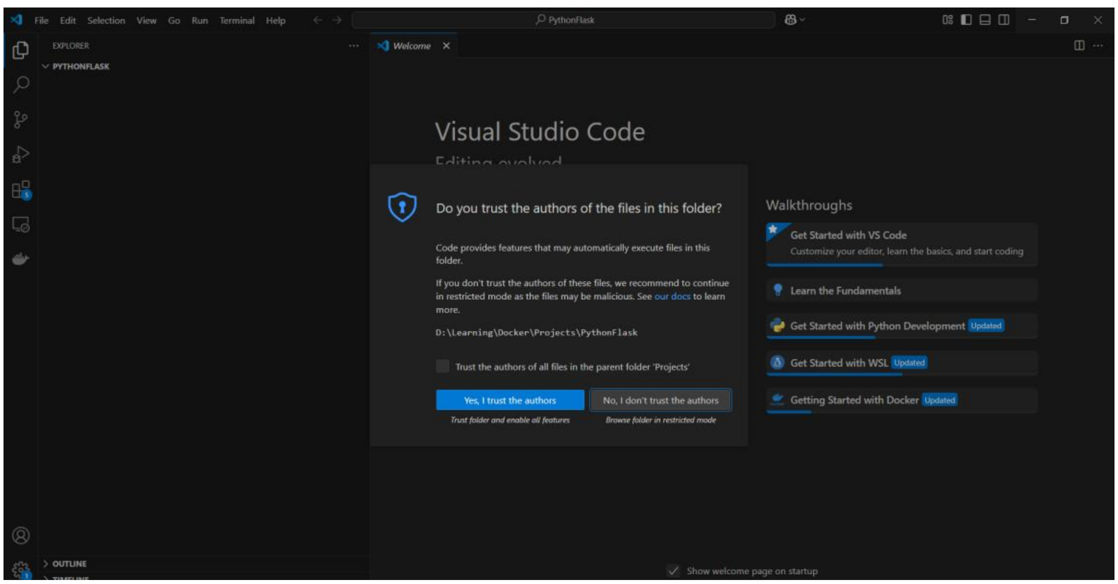


In the **Git Bash** command prompt, run the following commands to create a new directory for our Dockerized Python application at `D:\Learning\Docker\Projects` location and open **VS Code**. You can choose any location to create a new directory.

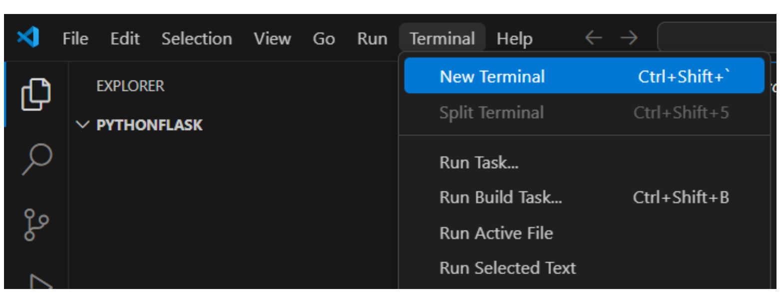
```
cd "D:\Learning\Docker\Projects"
mkdir PythonFlask
cd PythonFlask
code .
```



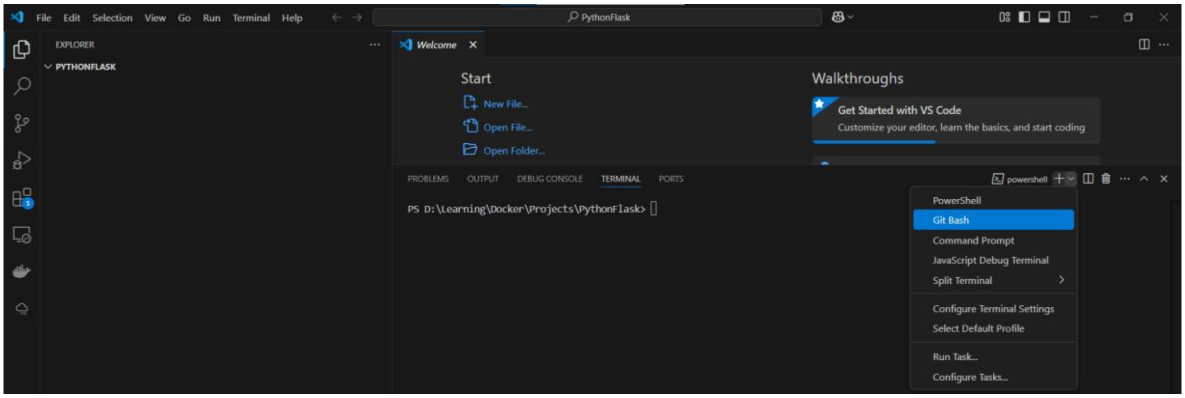
It opens **VS Code** application which might prompt you to confirm if you trust the authors of file in which case, click on **Yes, I trust the authors** button.



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.



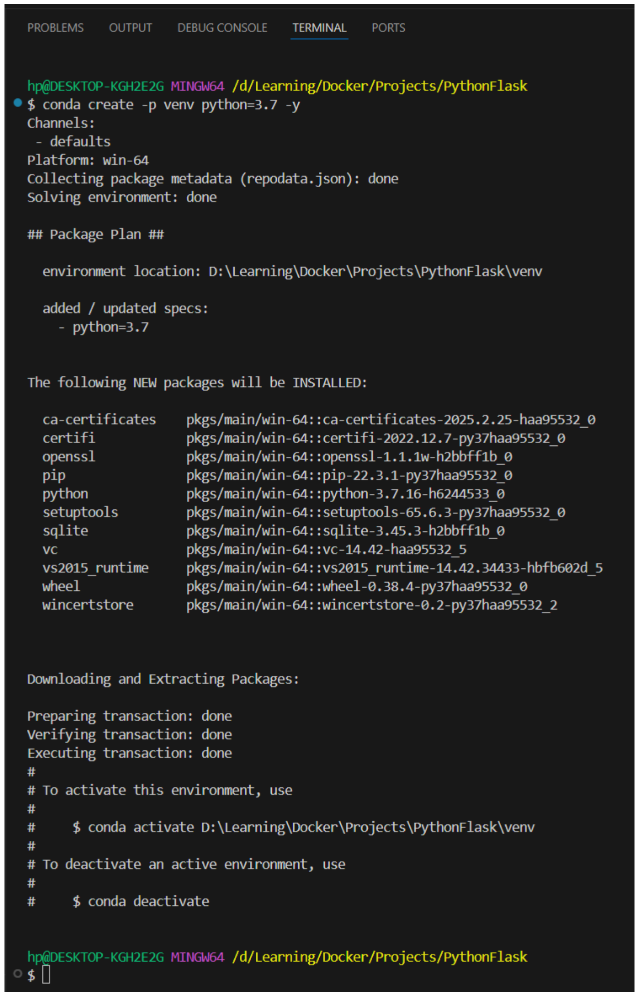
10.3 Create Virtual Environment

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.

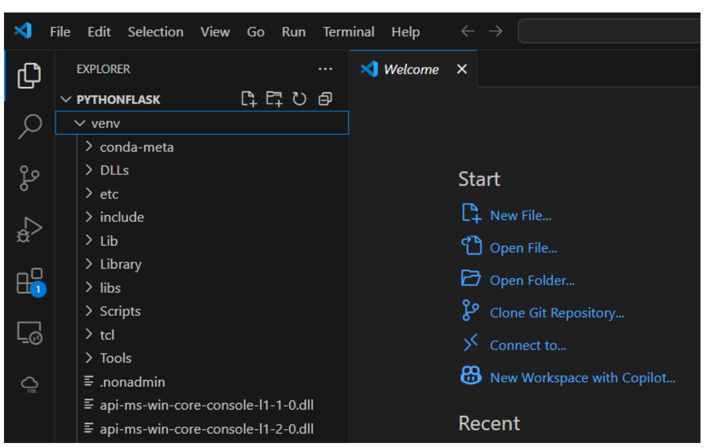
In the **Terminal** window, run the following command to create virtual environment with Python 3.7 version:

```
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.



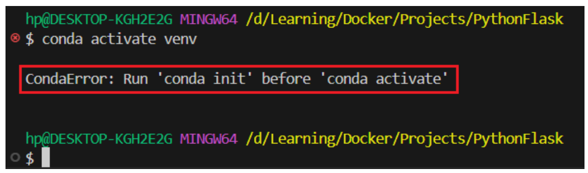
Once the Python virtual environment got created successfully, we can see that `venv` directory which contains Python related libraries and files got created under `PYTHONFLASK` project.



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

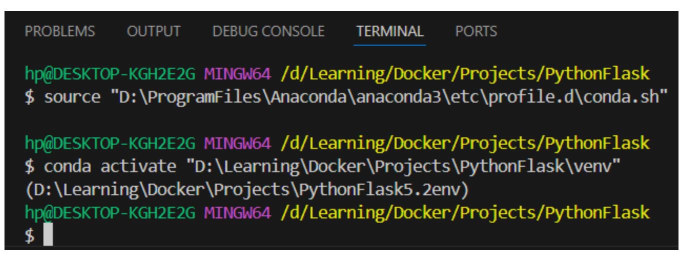
```
conda activate venv
```

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



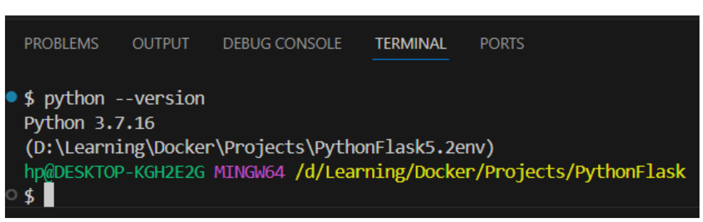
To fix this issue, source the location of `conda.sh` file from your Anaconda installation path and then activate the virtual environment using the following commands. In my case Anaconda was installed at `D:\ProgramFiles\Anaconda\anaconda3` and if your Anaconda installation path is different, replace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

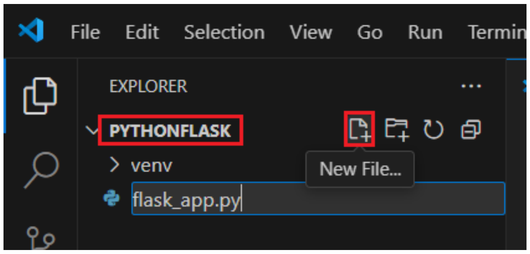
```
python --version
```



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

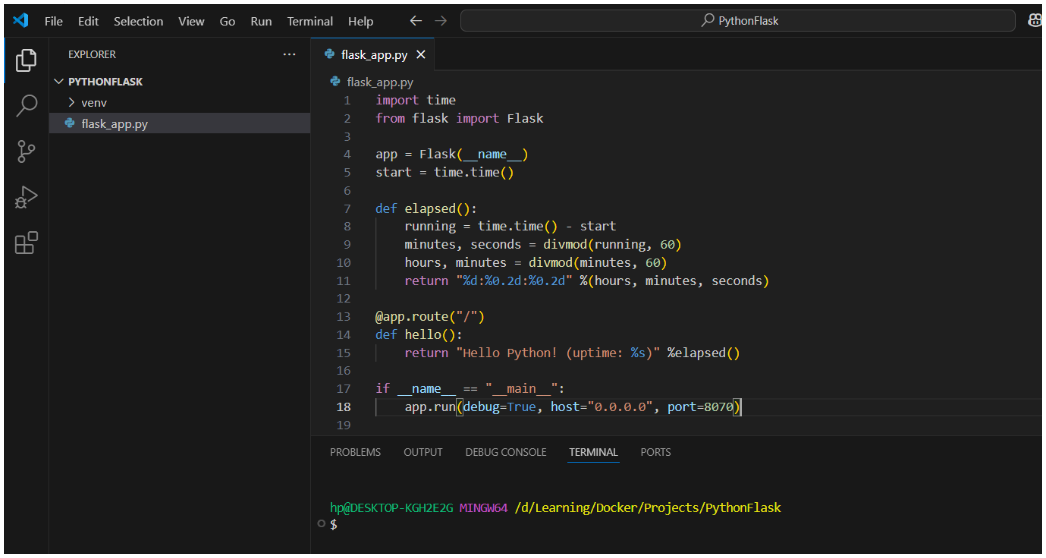
10.4 Create Python File

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.

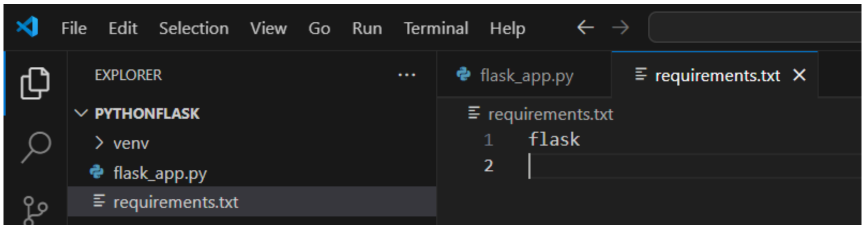
```
import time
from flask import Flask

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.

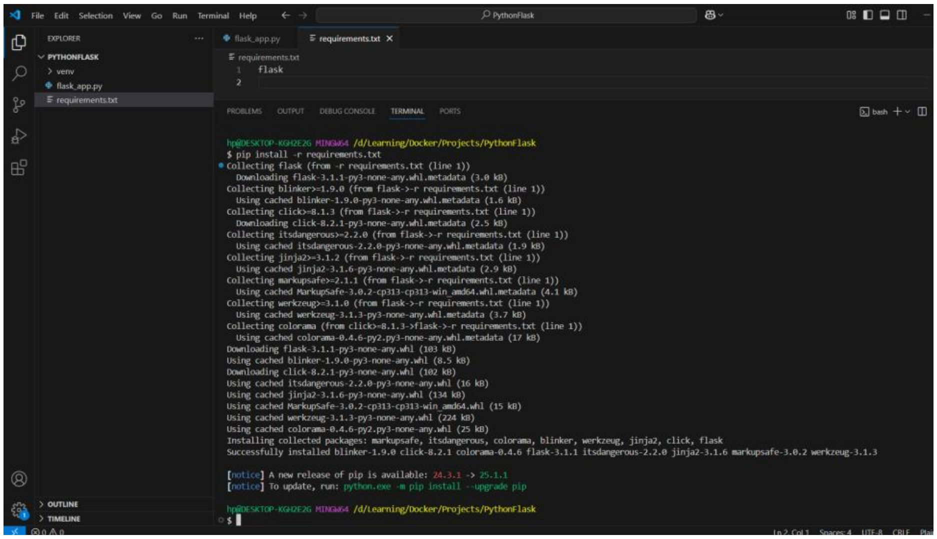
Enter the following line in `requirements.txt` file and save it.

```
flask
```



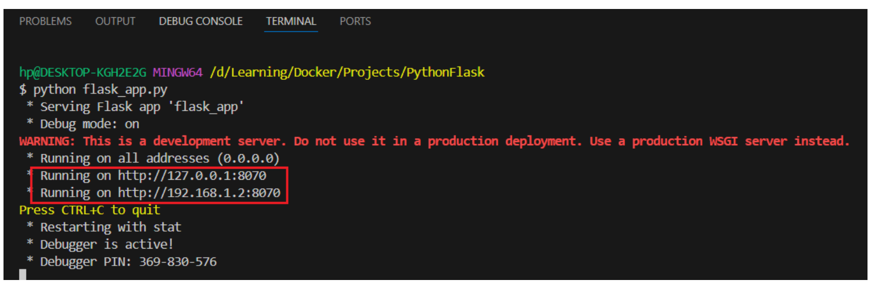
Next, run the following command in the VS Code Terminal window to install Python library requirements:

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

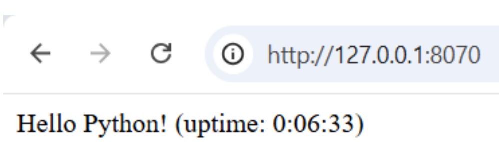


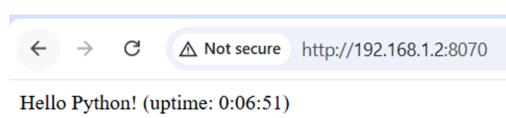
Now, execute the `flask_app.py` file using the following command:

```
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.

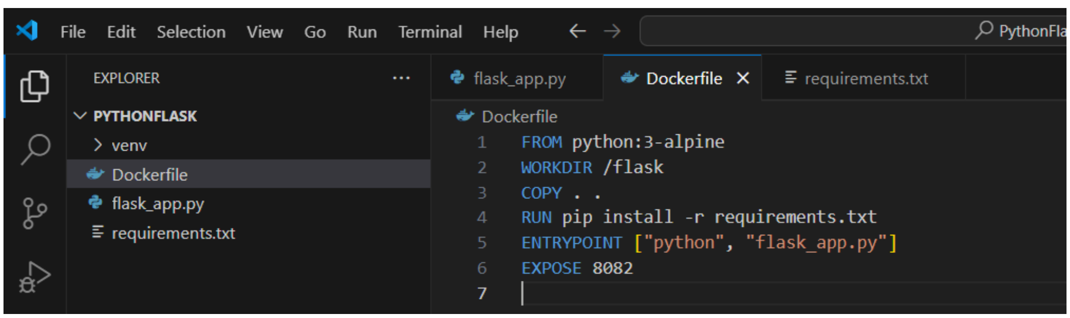
Press **Ctrl+C** on the VS Code terminal to exit out of Flask application.

10.5 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
```



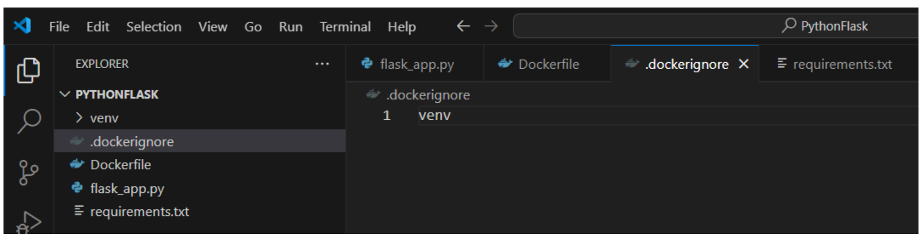
The above Docker instructions perform the following actions:

- `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
```



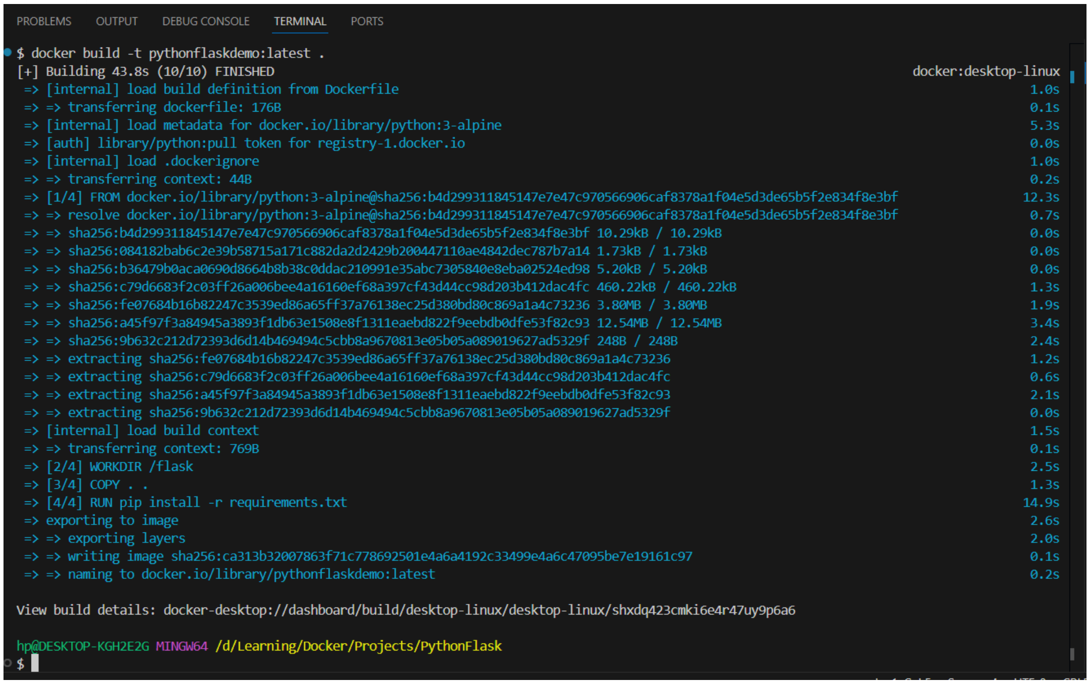
10.6 Build Docker Image

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.

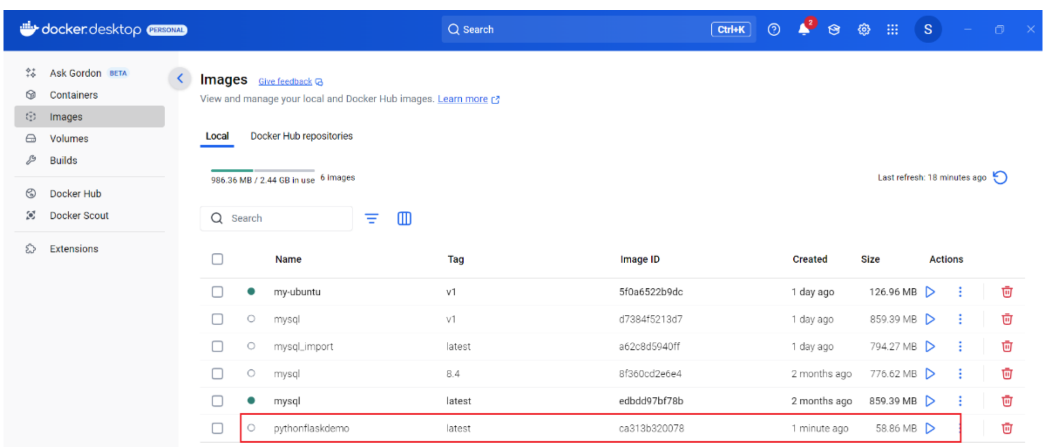
```
docker build -t pythonflaskdemo:latest .
```

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

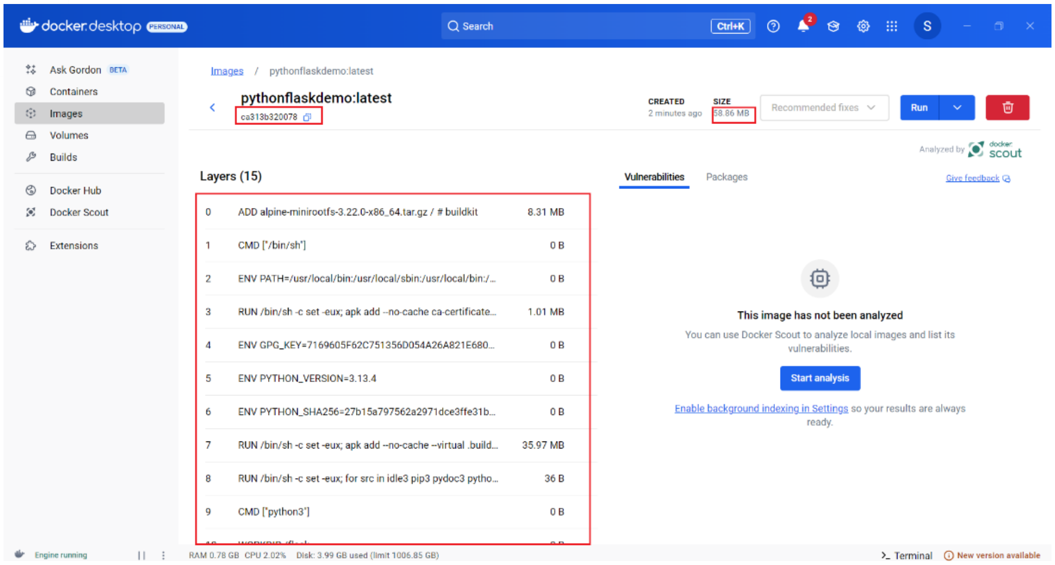
```
docker build -t pythonflaskdemo:latest . -f <docker_file>
```



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



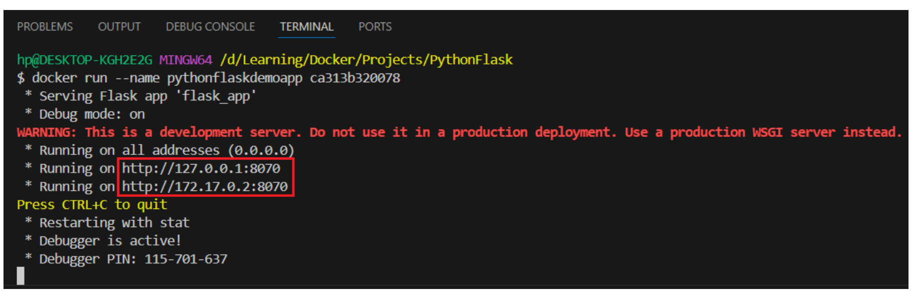
Click on `pythonfalskdemo` image and it displays image ID, image size and layers created to makeup this image.



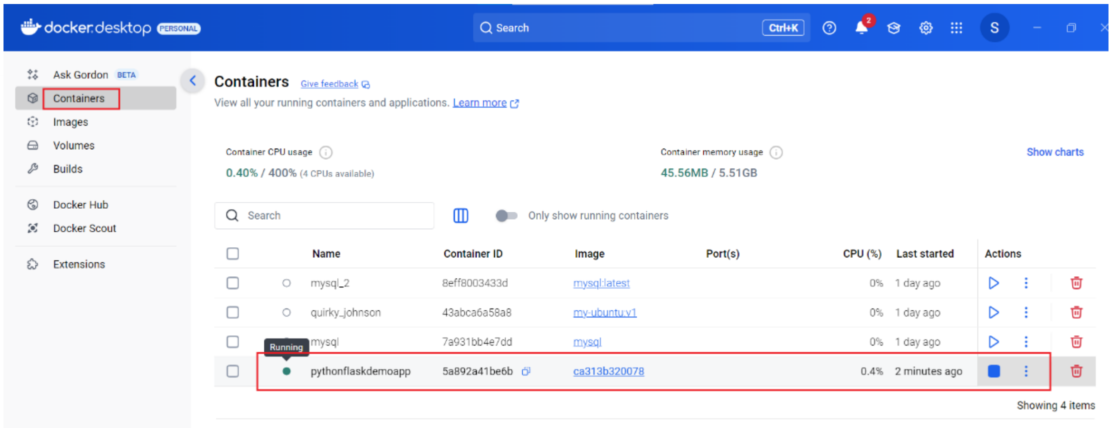
10.7 Run Docker Image

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.

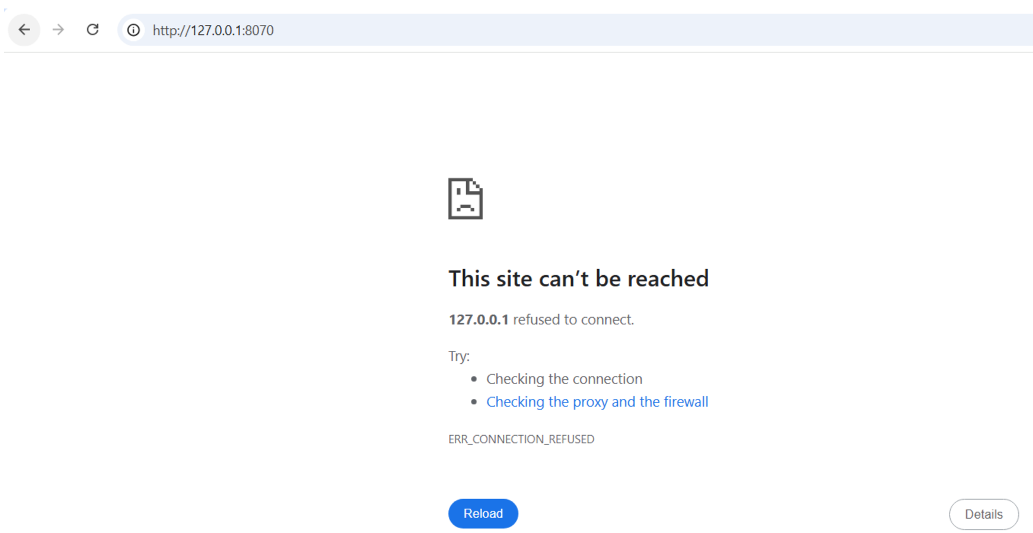
```
docker run --name pythonflaskdemoapp ca313b320078
```



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

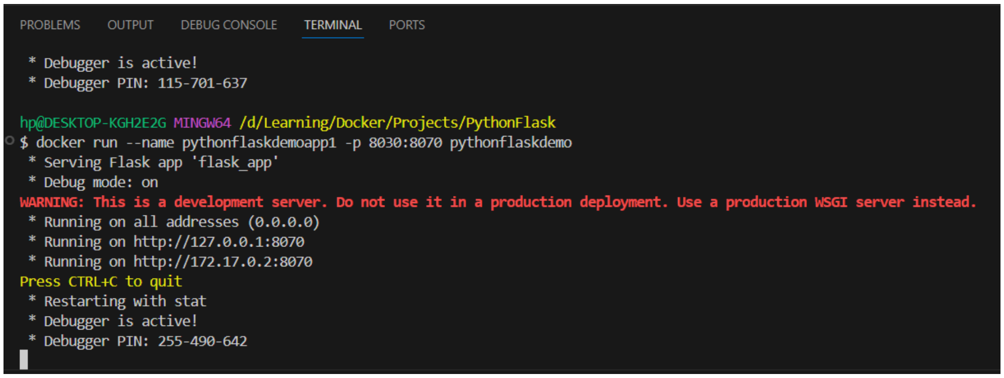


Now, hit the URL http://127.0.0.1:8070/ produced by the docker image upon running but it displays site cannot be reached error because the flask application port is not mapped to run outside the Docker container.

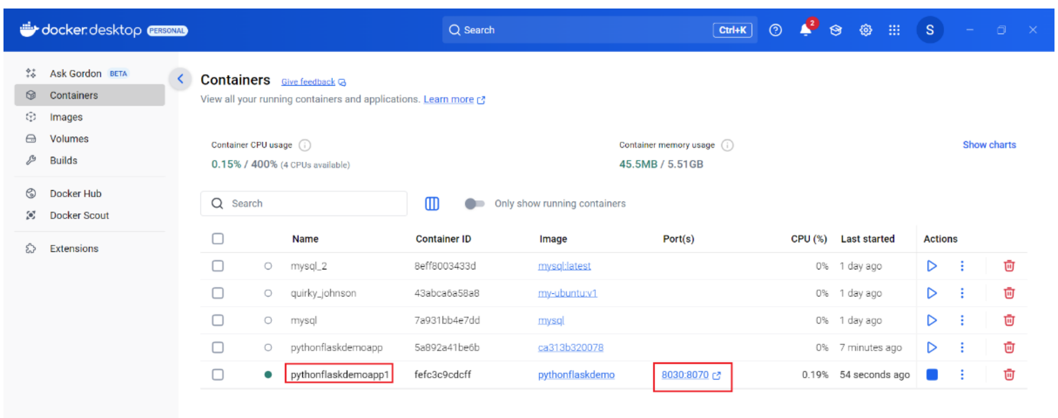


To overcome this problem, run the Docker container using the following command which maps the flask application port `8070` to port `8030` in local system. Press **Ctrl + C** to exit out of the previous flask execution before executing this command:

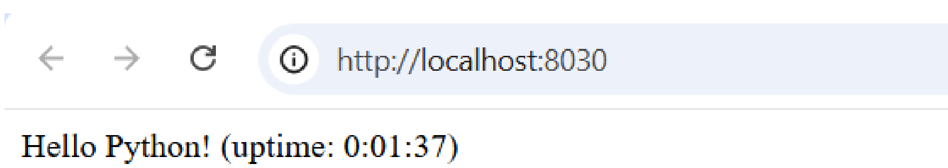
```
docker run --name pythonflaskdemoapp1 -p 8030:8070 pythonflaskdemo
```



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.



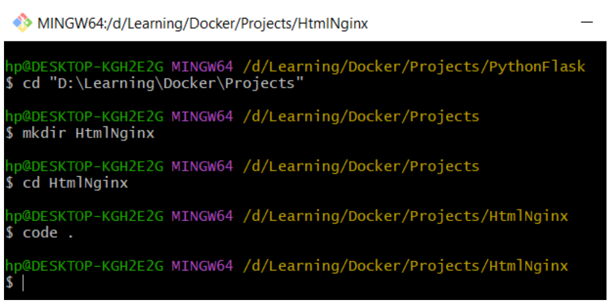
11 CREATE HTML DOCKER APPLICATION

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.

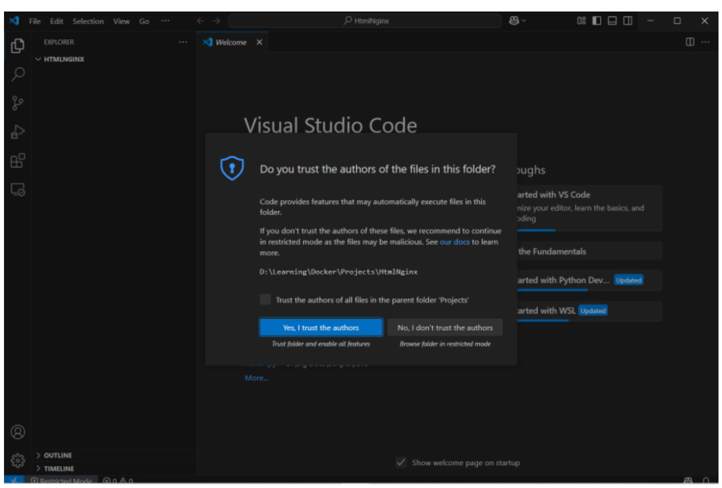
11.1 Launch VS Code

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.

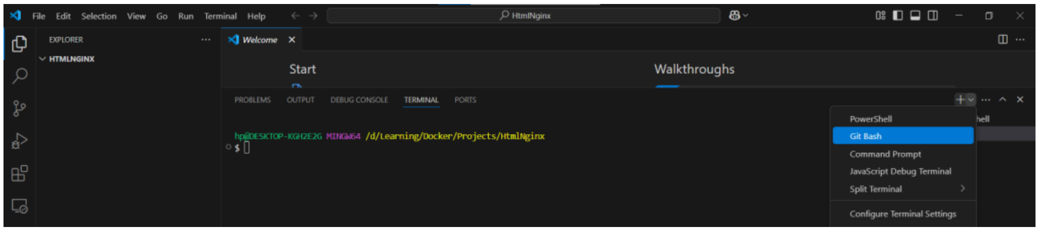
```
cd "D:\Learning\Docker\Projects"
mkdir HtmlNginx
cd HtmlNginx
code .
```



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.

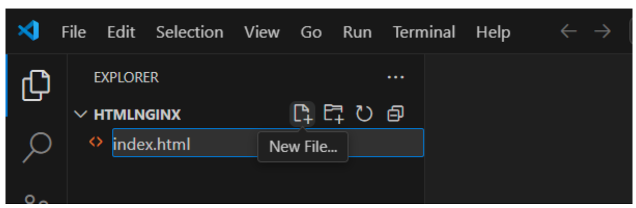


In VS Code, go to **Terminal** menu and select **New Terminal** and choose **Git Bash** terminal from the dropdown in the **Terminal** tab below.



11.2 Create HTML File

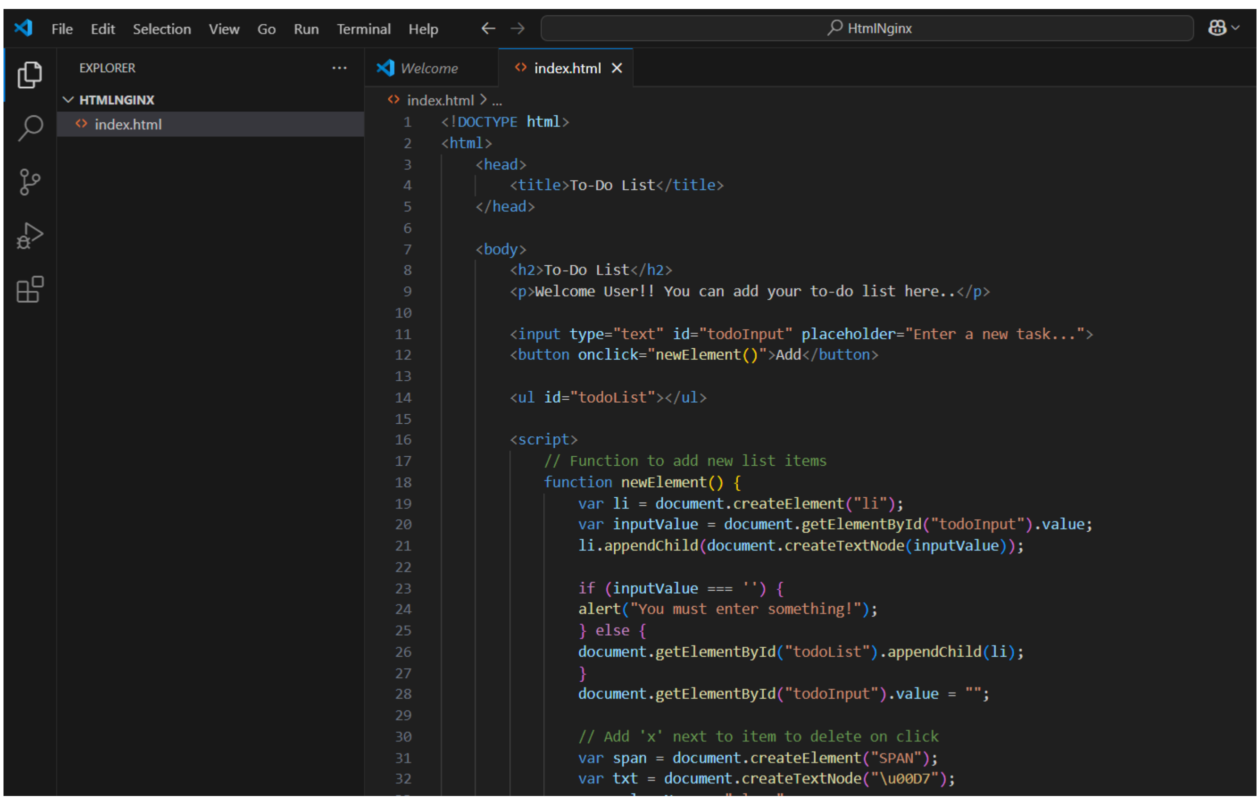
In **VS Code** application, click on **New File** icon next to `HTMLNGINX` project under **EXPLORER** and name the file as `index.html`



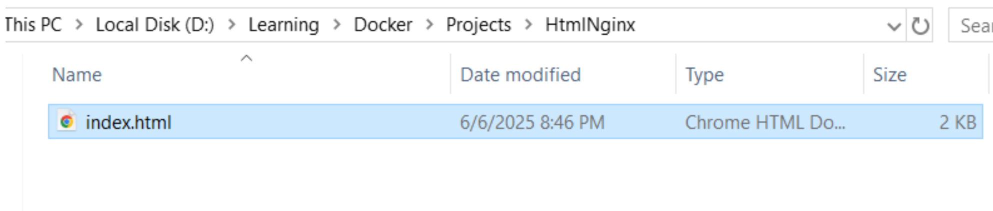
Enter the following code in `index.html` file and save it.

```
<!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;
```

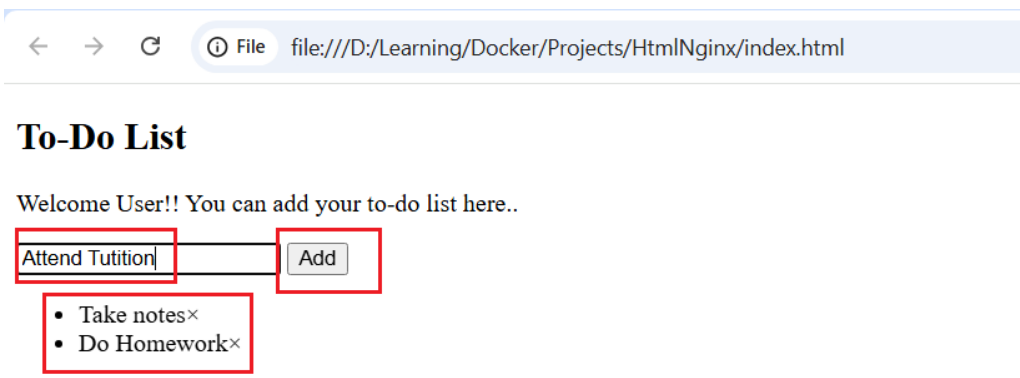
```
                        li.appendChild(document.createTextNode(inputValue));

                        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.

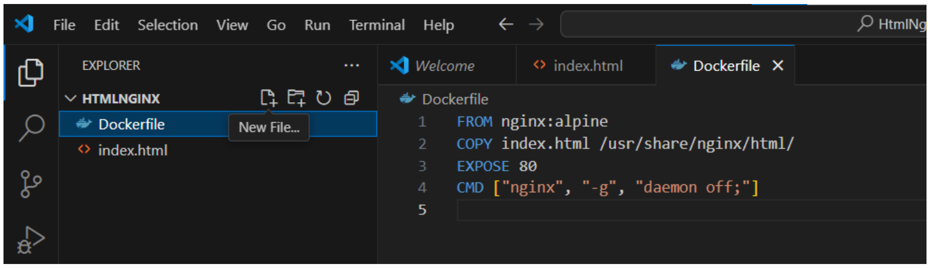


11.3 Create Docker File

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.

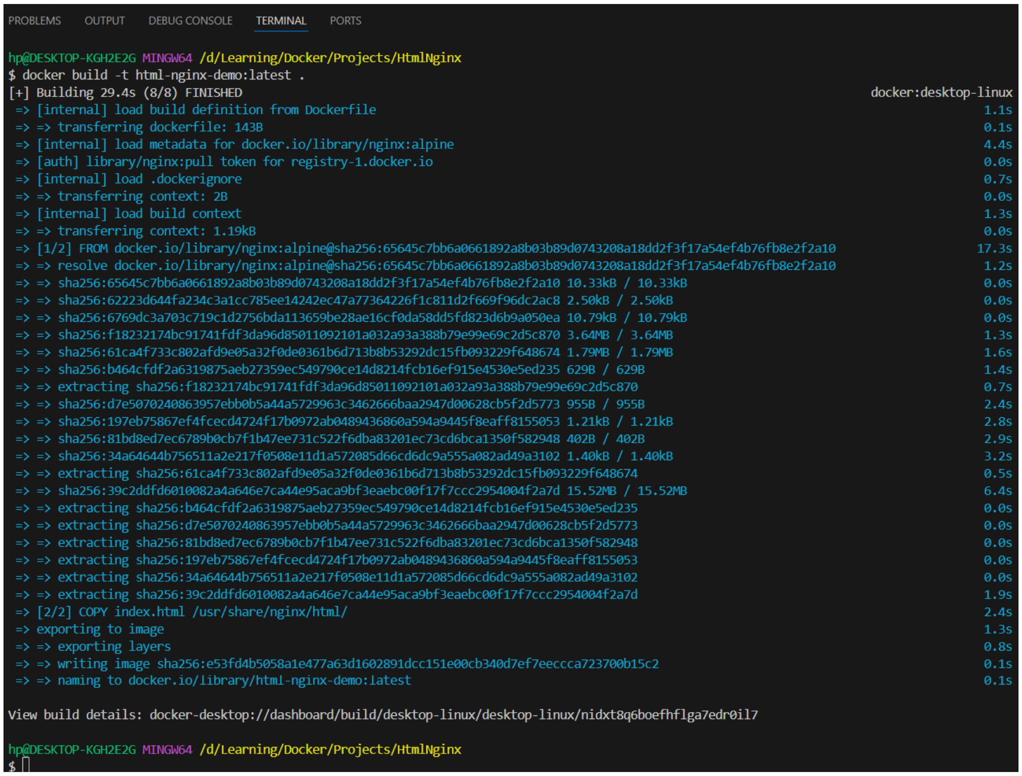
11.4 Build Docker Image

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.

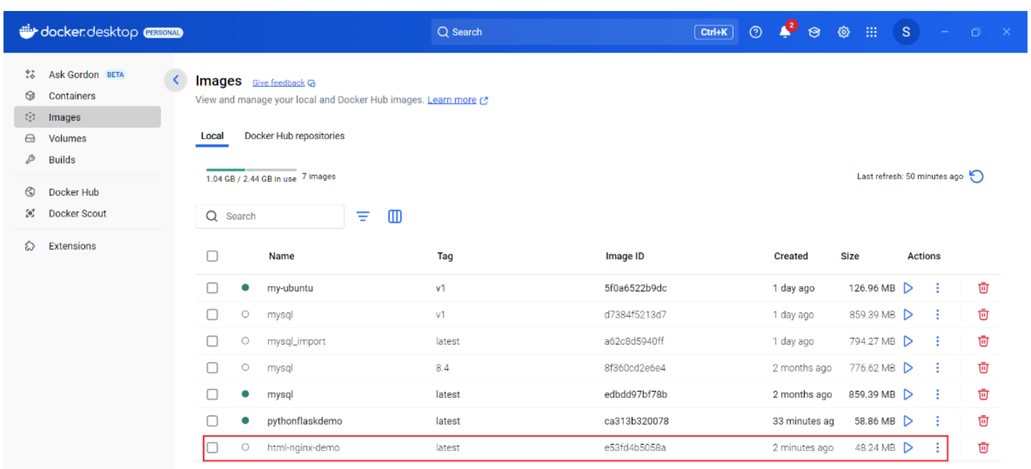
```
docker build -t html-nginx-demo:latest .
```

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

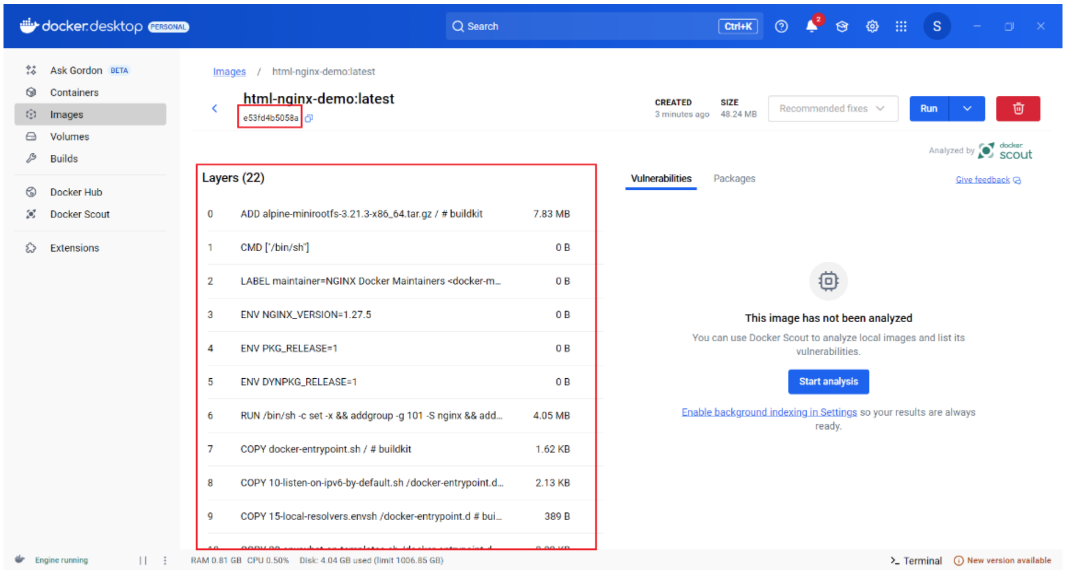
```
docker build -t html-nginx-demo:latest . -f <docker_file>
```



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



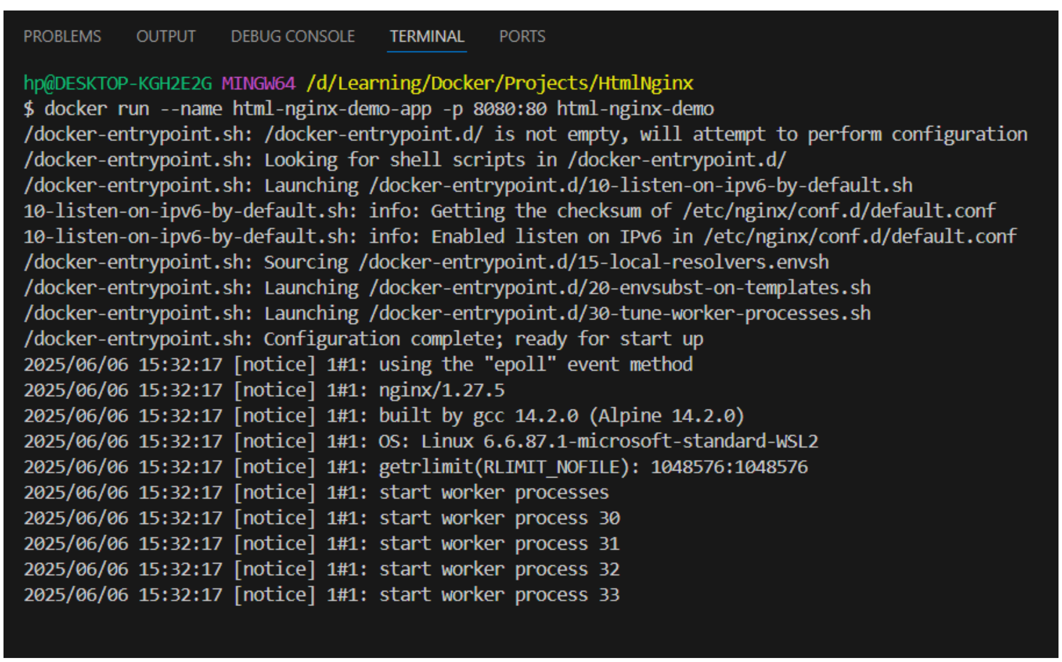
Click on `html-nginx-demo` image and it displays image ID, image size and layers created to makeup this image.



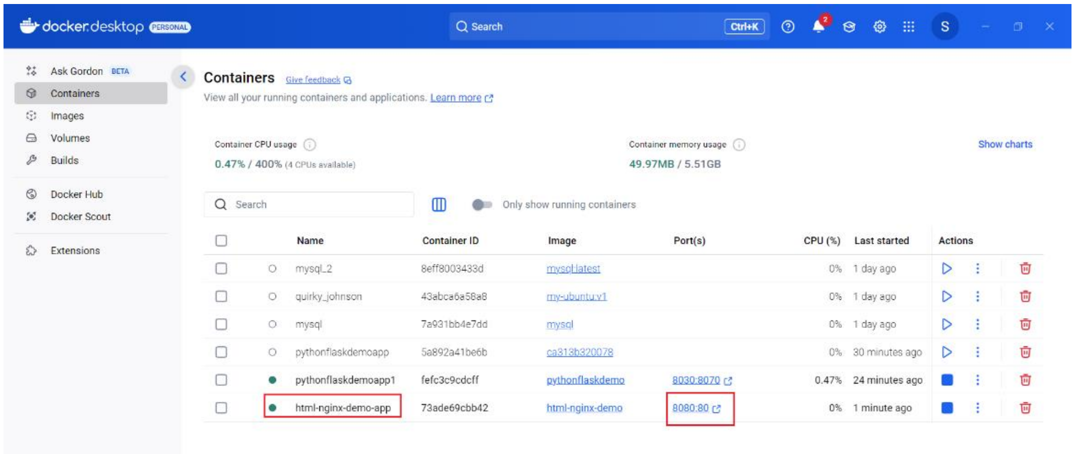
11.5 Run Docker Image

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

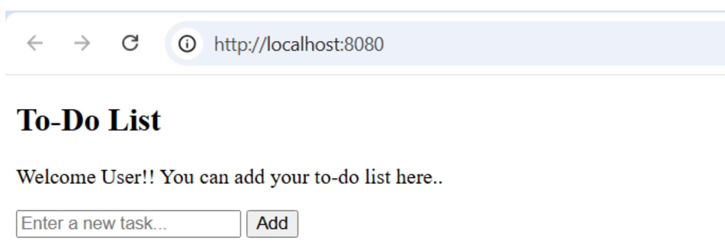
```
docker run --name html-nginx-demo-app -p 8080:80 html-nginx-demo
```



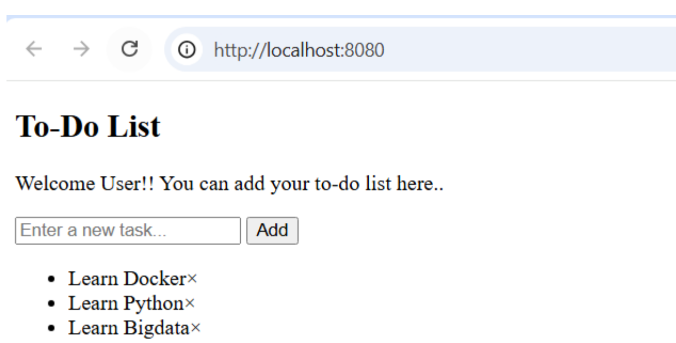
Go to **Docker Desktop** and see that a new container called `html-nginx-demo-app` has been created and running on port `8080` in **Containers** view.



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.



11.6 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.

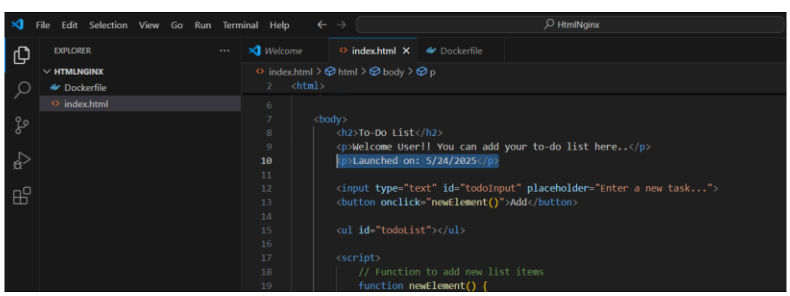
Change lines from:

```
        <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:

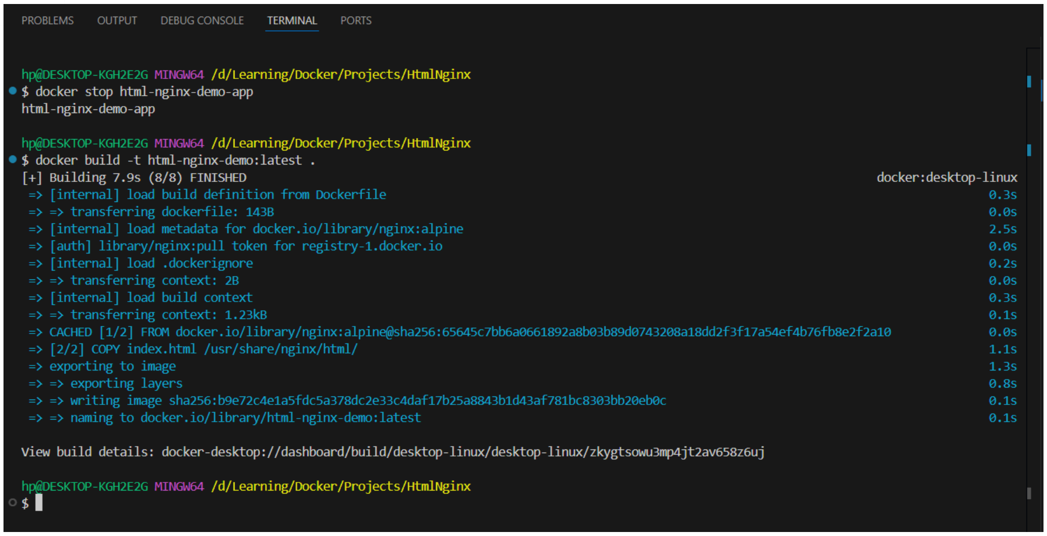
```
        <body>
                <h2>To-Do List</h2>
                <p>Welcome User!! You can add your to-do list here..</p>
        <p>Launched on: 5/24/2025</p>

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

Save changes in `index.html` file.

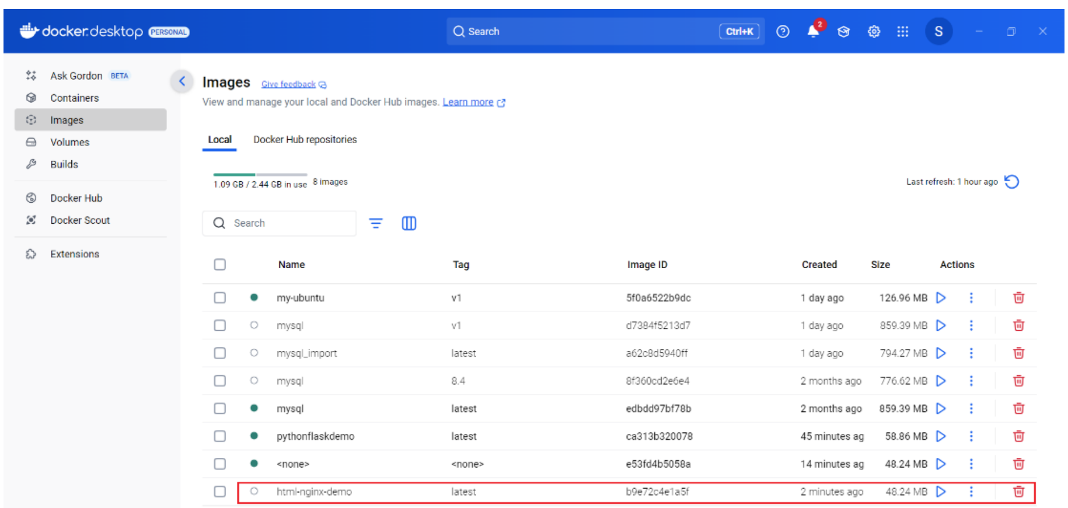


Now, open another terminal and run the following commands to stop the running container and rebuild the image file

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

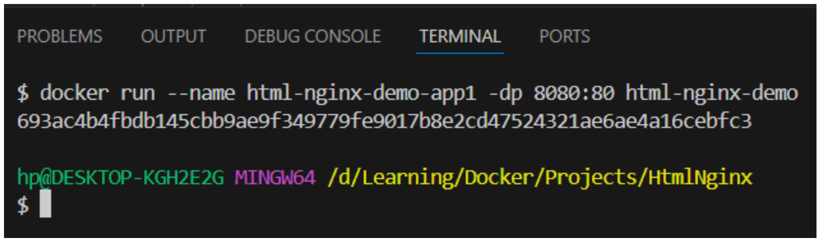


In **Docker Desktop**, you can see that the image was rebuilt.

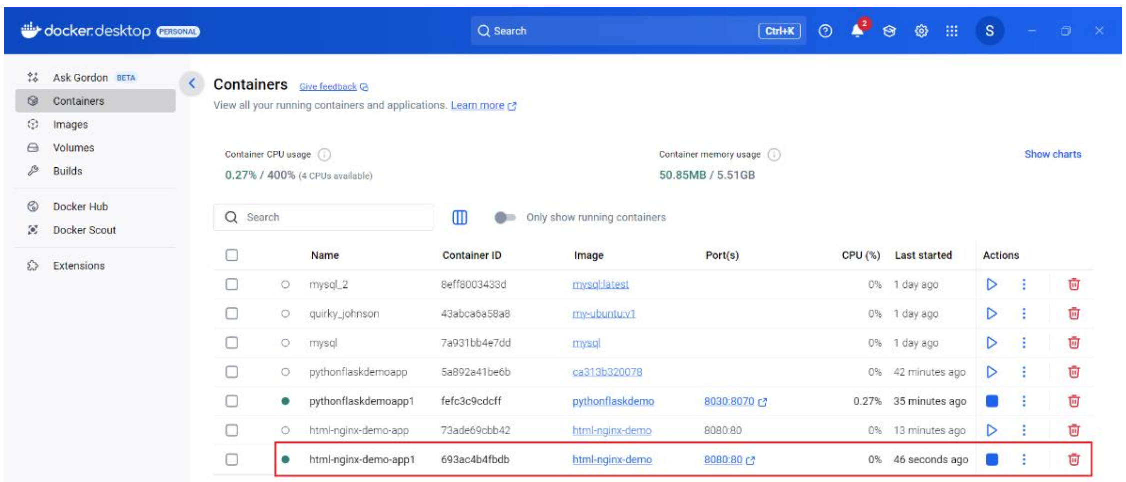


Now, let us run the container in the detached mode (to run container at background) with the new image by executing the following command:

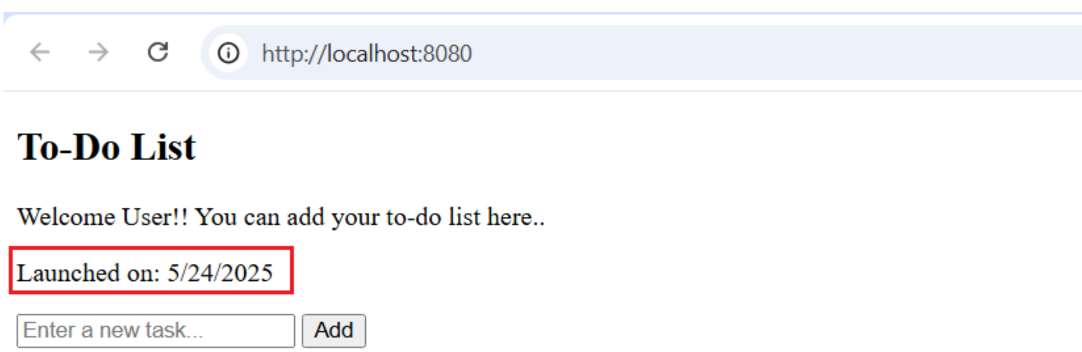
```
docker run --name html-nginx-demo-app1 -dp 8080:80 html-nginx-demo
```



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.



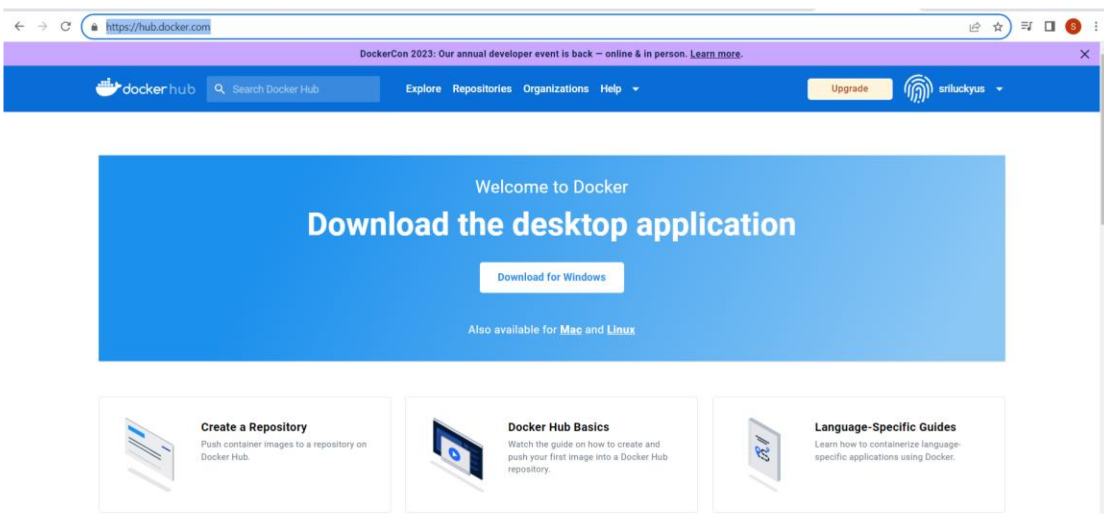
12 PUBLISH DOCKER IMAGE TO DOCKER HUB

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.

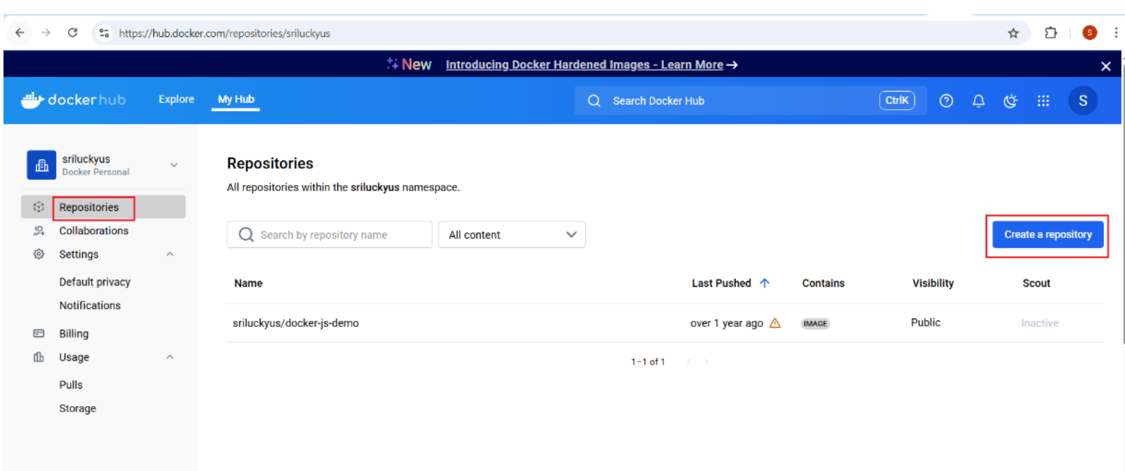
12.1 Create Repository

First, create a repository in Docker Hub.

Login to https://hub.docker.com/



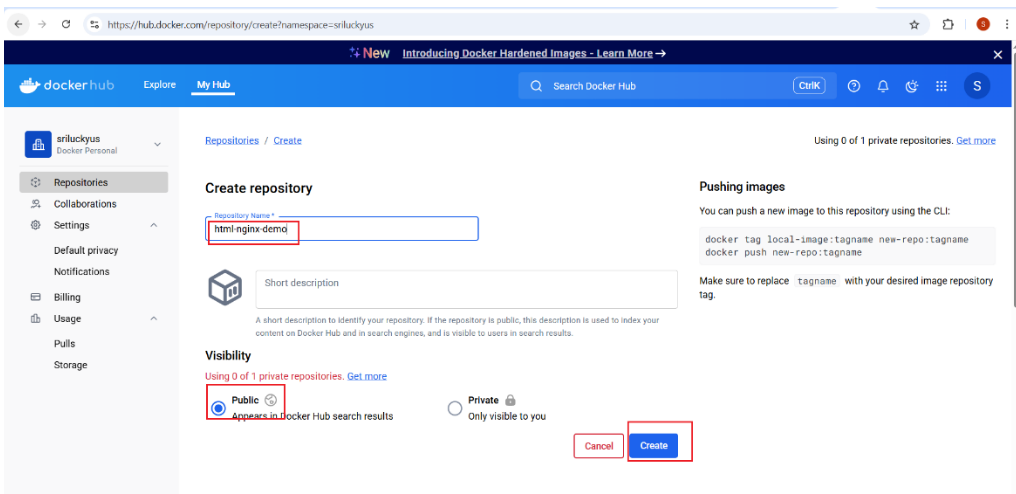
Select **Repositories** tab and click on **Create repository.**



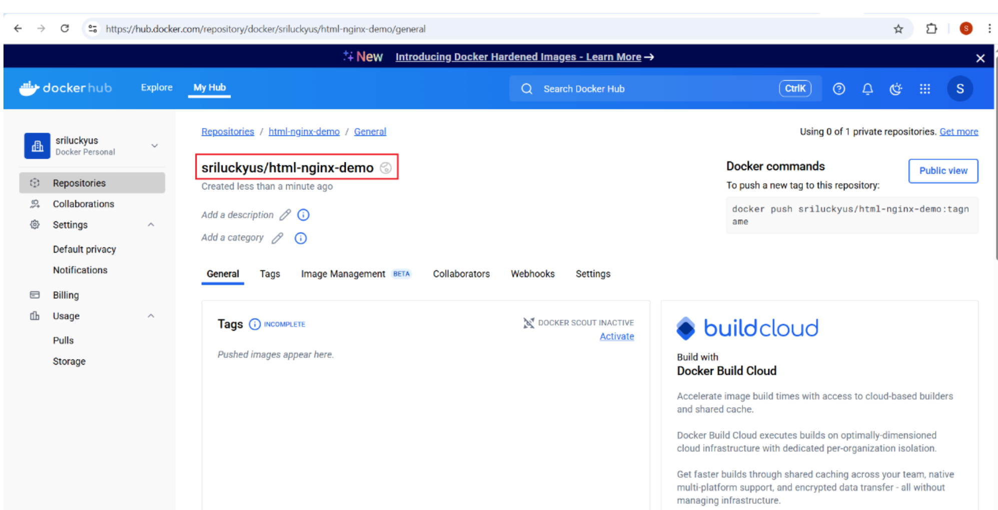
Under **Create repository** page:

- 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.

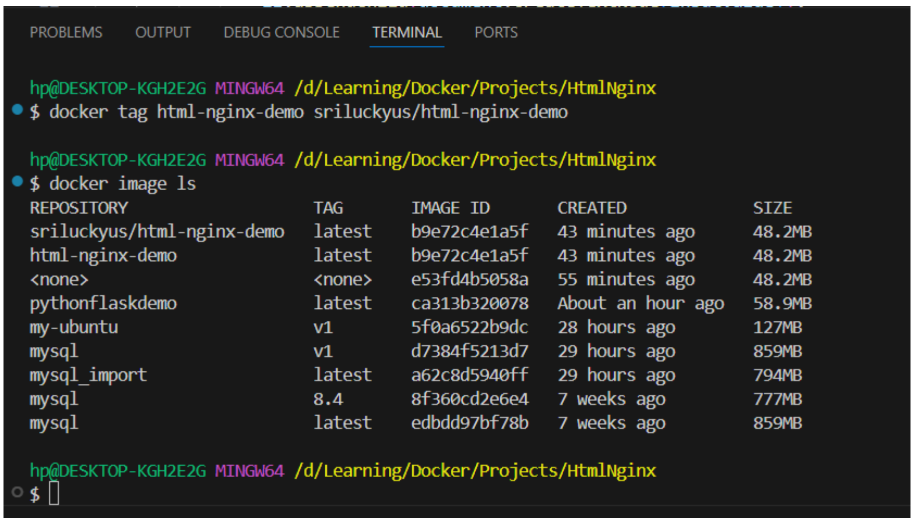


12.2 Tag Image to User Repository

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
```

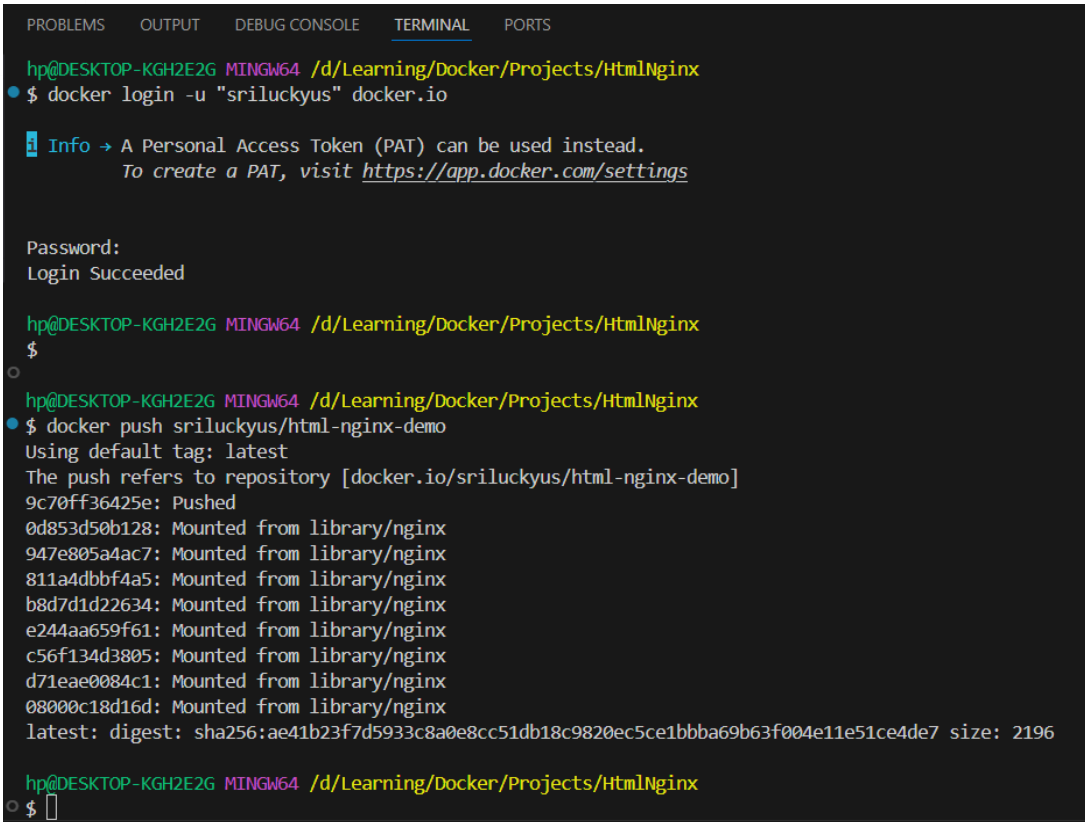


```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/HtmlNginx
$ docker tag html-nginx-demo sriluckyus/html-nginx-demo

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/HtmlNginx
$ docker image ls
REPOSITORY                   TAG       IMAGE ID       CREATED            SIZE
sriluckyus/html-nginx-demo   latest    b9e72c4e1a5f   43 minutes ago     48.2MB
html-nginx-demo              latest    b9e72c4e1a5f   43 minutes ago     48.2MB
<none>                       <none>    e53fd4b5058a   55 minutes ago     48.2MB
pythonflaskdemo              latest    ca313b320078   About an hour ago  58.9MB
my-ubuntu                    v1        5f0a6522b9dc   28 hours ago       127MB
mysql                        v1        d7384f5213d7   29 hours ago       859MB
mysql_import                 latest    a62c8d5940ff   29 hours ago       794MB
mysql                        8.4       8f360cd2e6e4   7 weeks ago        777MB
mysql                        latest    edbdd97bf78b   7 weeks ago        859MB

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

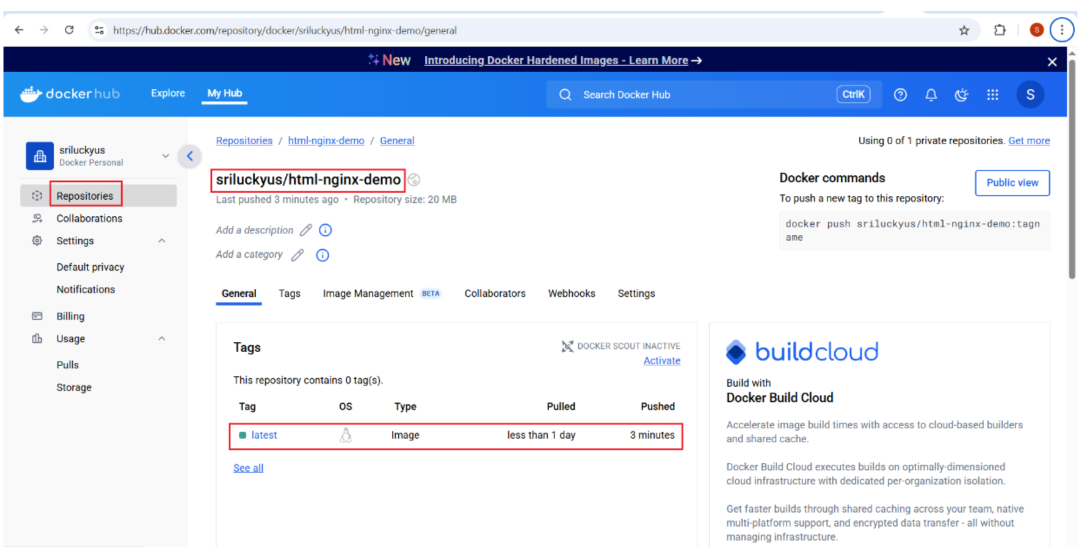
12.3 Login & Push User Repo Docker Image

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
```

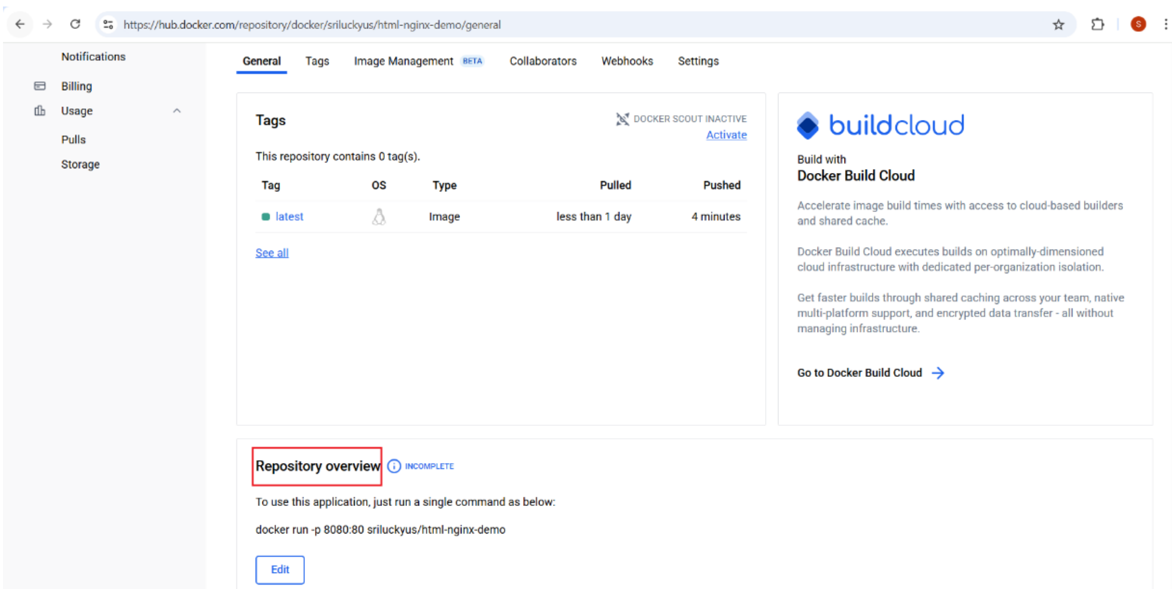


In Docker Hub, the image is visible under the new repository created



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
```

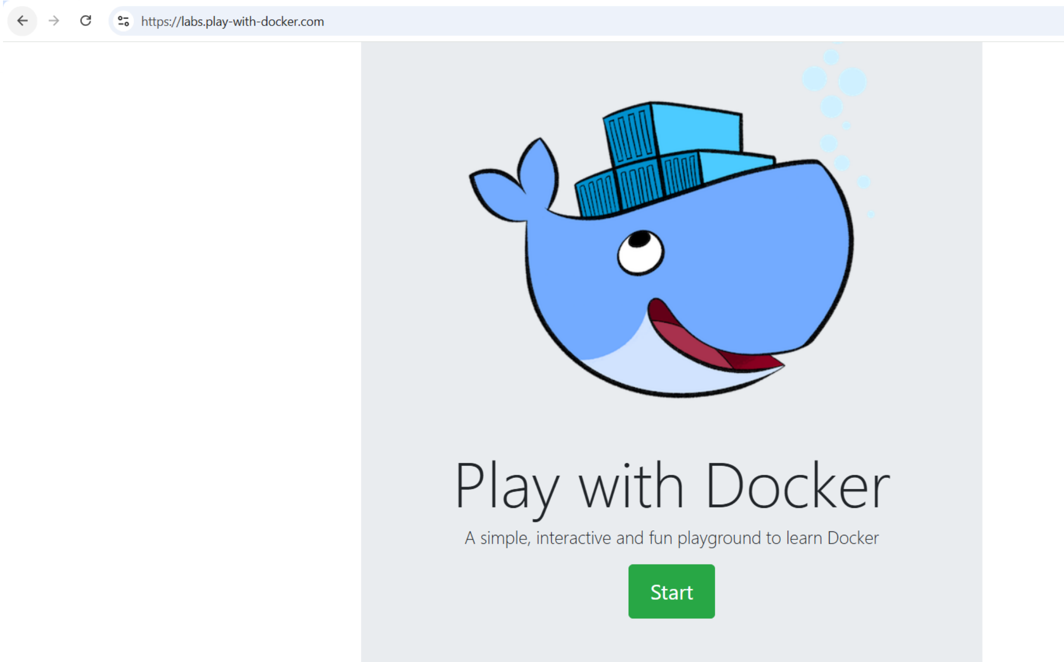


Now, anyone who has access to the above repository can get the docker container running locally.

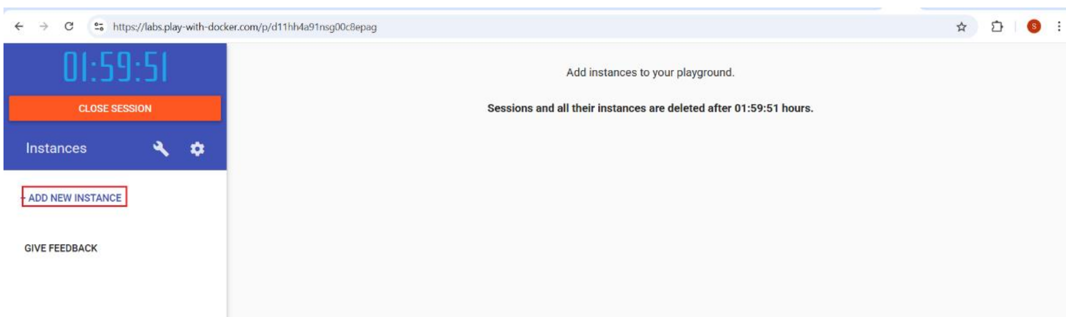
12.4 Run User Repo Docker Image

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.

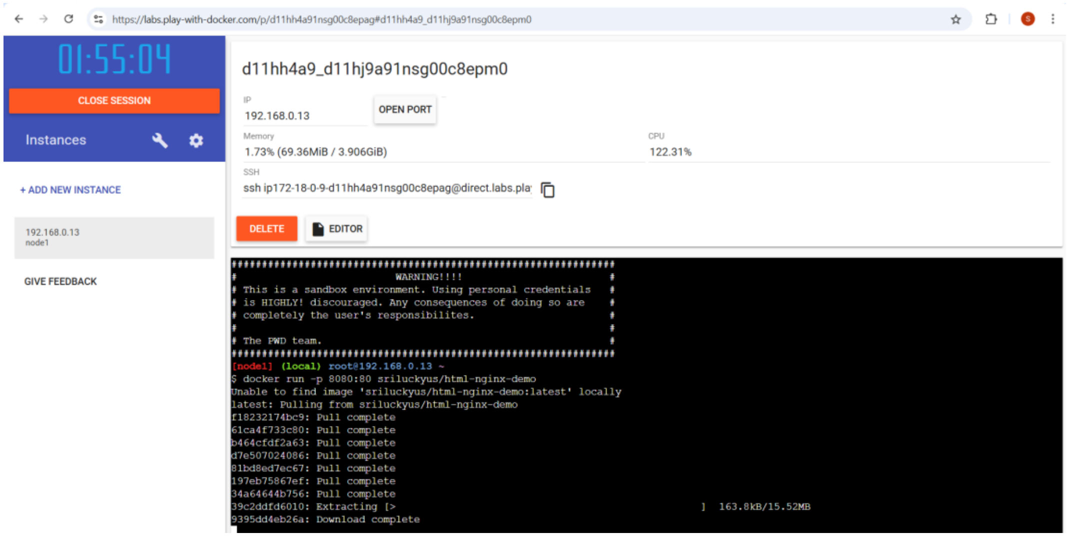


Click on **ADD NEW INSTANCE** to create a new Docker instance

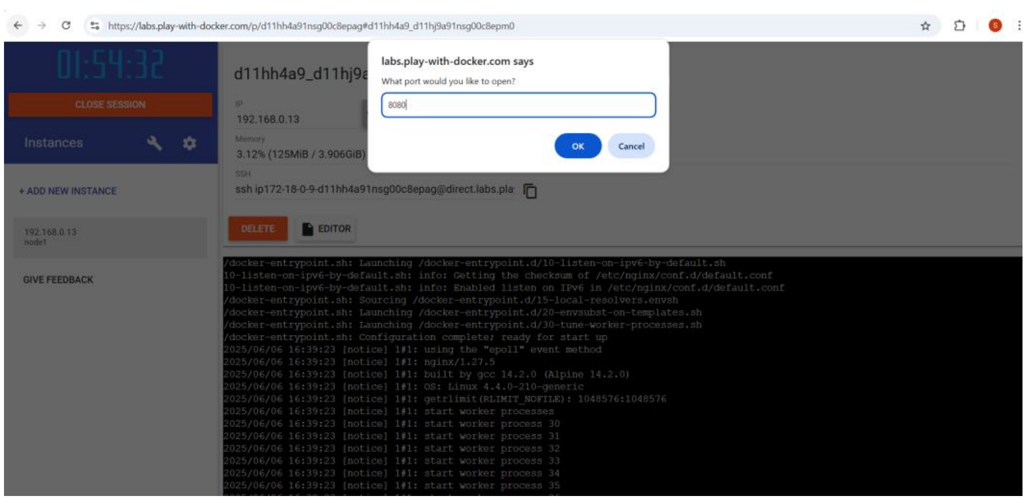


Once the Docker instance is created, run the `html-nginx-demo` image available in `sriluckyus` Docker Hub user by executing the below command in the terminal:

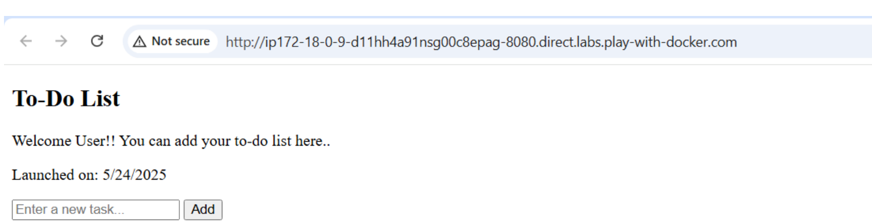
```
docker run -p 8080:80 sriluckyus/html-nginx-demo
```



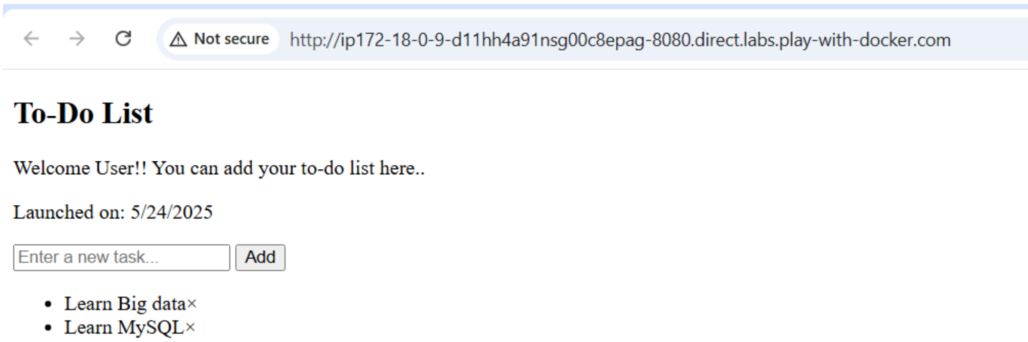
Since our web application is listening on port `8080`, click on **OPEN PORT** button and enter `8080`.



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.



13 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 which uses **YAML** scripting language. Then, all services with inter-connected containers are created and started with a single command from the configuration file.

13.1 Docker Compose File

Docker Compose tool looks for the docker compose file named `compose.yaml` (preferred) or `compose.yml` in the current working directory. If it does not exist, it looks for at least the file named `docker-compose.yaml` or `docker-compose.yml` in the current working directory. This compose file contains the definition of all services that make up the application, along with their dependencies and configurations. It is important to understand YAML scripting language to write Docker compose file since indentations are important for YAML file.

The structure of the docker compose file can include the following elements:

- `version` – It defines the version of the Compose file format specifying which Docker Compose features and syntax to use. It is typically defined at the top of the file. This element has been deprecated in the latest version of Docker Compose.

- `name` – It defines the project name to be used. When this is not defined, Docker compose uses a default project name.

- `services` – This section lists out each containerized service required for the application. In Docker Compose, every component of the application operates as a separate service, with each service running a single container such as a database or web server, or cache. Each service is defined as a separate block with a name and its configuration options such as:
    - `container_name`: It is used to specify a customer container name.
    - `image`: It defines the Docker image to use by the service and this image is the one that is either built locally or pulled from the Docker Hub or any other registry.
    - `build`: It defines to build the image locally from the `Dockerfile` in the specified directory. This is helpful for including custom code in the application. It can have additional configuration options such as `context` *(to define a path to a directory containing a `Dockerfile`)*, `dockerfile` *(to set the name of Docker file used)*, `target` *(to define the stage to build)*, etc. Refer to Compose Build Specification for complete details on build configuration.
    - `environment`: It allows to pass configurations or sensitive information such as database credentials or API keys, to the service.
    - `env_file`: It is used to specify one or more files that contain environment variables to be passed to the containers. Environment variables declared in the `environment` section override these values.
    - `ports`: It defines to map a container's internal ports to those on the host machine, enabling access to services from outside the container.

- `expose`: It defines the port or range of ports that are exposed from the container.
- `volumes`: It attaches persistent storage to a service so that data is accessible even when a container is restarted.
- `working_dir`: It is used to set the working directory. It overrides WORKDIR instruction specified for the container image in `Dockerfile`.
- `restart`: It is used to define how the service should behave in case of failures. It can values including `no`, `always`, `on-failure`, and `unless-stopped`.
- `restart-policy`: It is used to specify the restart policy for individual containers. It overrides the global restart policy set in the services section. It can have additional options such as `condition`, `delay`, `max_attempts`, etc.
- `healthcheck`: It is used to define health check configurations for the service and determine if a service is healthy or not. It can have additional options such as `test`, `interval`, `timeout`, `retries`. It overrides HEALTHCHECK instruction specified for the container image in `Dockerfile`.
- `depends_on`: It is used to define dependencies between services by controlling the startup order of services. It ensures that one service starts only after its dependent services are up and running. It can have additional options such as `restart` *(to restart this service after it updates the dependency service)*, `condition` *(to look for the condition of dependency service before starting this service and it can have values including* `service_started`*,* `service_healthy` *and* `service_completed_successfully`*),* `required` *(by default, set to true to ensure the condition is satisfied and when set to false, Docker compose only warns when the dependency service is not started or available)*.
- `command`: It allows to specify a custom command or arguments when starting the container. It overrides the default CMD instruction declared for the container image in `Dockerfile`.
- `entrypoint`: It declares the default entry point for the service container and overrides ENTRYPOINT instruction in the image `Dockerfile`.
- `hostname`: It is used to set a custom host name to use for the service container.
- `labels`: It is used to add metadata to services and containers.
- `extends`: It allows to extend and override configurations by reusing common configurations across multiple services or environments.

- `logging`: It allows to configure logging options for the service. It can have additional options such as `driver`, `path`, `options`, etc.
- `user`: It is used to specify the user and group that a container should run as to control permissions and access within the container.
- `deploy`: It defines to specify deployment options to manage containers across different environments. This is helpful to deploy containers with multiple replicas in a cluster platform such as Docker Swarm. It can have additional configuration options such as `mode` *(to define the replication model to run a service and it can have values including `global`, `replicated` (default), `replicated-job` and `global-job`)* , `replicas` *(to define the number of containers that should be running when the service is replicated)*, `resources` *(to configure physical resource constraints for container to run on platform)*, `restart_policy` *(to configure how to restart containers when they exit)*, `update_config` *(to configure how the service should be updated,* `rollback_config` *(to configure how the service should be rolled back when update fails)*, etc. Refer to Compose Deploy Specification for complete details on deployment configuration.

- `networks` – This section allows to define custom networks that are attached to various services so that secure communication is established between containers over isolated networks. Services defined in Docker Compose by default uses an implicit `default` network to connect with each other and the `default` network can also be customized with this section. Each network is defined as a separate block with a name and configuration options such as:
  - `name`: It allows to set a custom name for the network.
  - `driver`: It is used to define the network driver type, such as `bridge` *(default for local networks)* or `overlay` (*for multi-host networks in Docker Swarm)*.
  - `driver_opts`: It allows to define additional settings on the network driver.
  - `attachable`: It is used to enable the standalone containers to attach to this network along with the service containers if this is set to `true`. If a standalone container attaches to the network, it can communicate with services and other standalone containers that are also attached to the network.

- `ipam`: IP address management (IPAM) configures the network-level IP settings including subnets and IP ranges to get more control over IP address space assigned to services. It uses additional options such as `driver`, `config`, `options`.
- `external`: It allows to connect to an external network created outside of the Docker compose.
- `internal`: It specifies to create an isolated network when this option is set to `true`. By default, Docker compose provides external connectivity to networks.
- `labels`: It is used to add metadata to networks.

- `volumes` – This section defines various volumes to persist data created by services and to share data between containers. This persisted data exists even if containers are stopped or removed. Each volume is defined as a separate block with a name and its configuration options such as below:
  - `name`: It allows to set a custom name for the volume.
  - `driver`: It indicates the volume driver to be used by the volume. The default driver is `local`.
  - `driver_opts`: It allows to customize the volume driver with additional options such as `type` (`none`, `nfs`), `o`, `device`, etc.
  - `external`: It allows to use external volume that was created outside of Docker Compose.
  - `labels`: It is used to add metadata to volumes.

- `configs` – This section allows to define configurations for the services using external files. These configurations can include environment variables, file paths, and other settings. Each config is defined as a separate block with a name and its configuration options such as below:
  - `name`: It allows to set a custom name for the config object in the container engine.
  - `file`: It is used to create a config with the contents of file at the specified path.
  - `environment`: It allows to create config content with the environment variable value.
  - `content`: It is used to create the content with the in-lined value.
  - `external`: It allows to use external config that was created outside of Docker Compose.

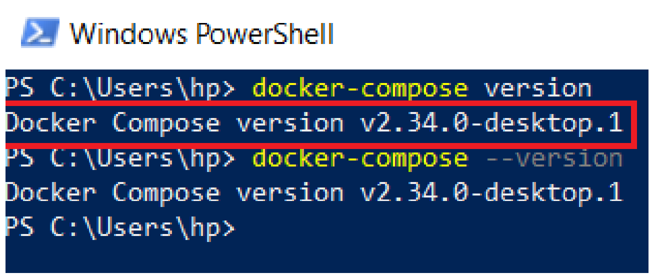
- `secrets` – This section allows to manage the sensitive data provided to the services securely. Secrets are encrypted and stored on the Docker host, and they can be accessed by services within Docker Compose setup. Each secret is defined as a separate block with its configuration options such as below:
    - `file`: It is used to create a secret with the contents of file at the specified path.
    - `environment`: It allows to create secret with the environment variable value.
    - `external`: It allows to use external secret that was created outside of Docker Compose.

13.2 Docker Compose Commands

The `docker compose` commands are used to manage multi-container applications.

- `docker compose version`: It is used to display the version information of Docker Compose.

```
docker compose version
```
or
```
docker compose --version
```



- `docker compose config`: It is used to validate and display the docker compose file configuration. This command accepts various options. Use `docker compose config --help` command for the complete list of options.

  Some common options are:
    - `-o` to save to a file
    - `-q` to validate the configuration only without printing
    - `--images` to print image names

  - `--services` to print service names
  - `--volumes` to print volume names

```
docker compose config <options>
```

- `docker compose build`: It is used to build or rebuild services. Once services are built, they are tagged as project-service, by default. This command accepts various options. Use `docker compose build --help` command for the complete list of options.
  Some common options are:
  - `--build-arg` to set build-time variables for services
  - `--pull` to always attempt to pull latest version of image
  - `--push` to push service images
  - `--no-cache` to not use cache when building the image ensuring fresh image is built

```
docker compose build <options>
```

- `docker compose pull`: It is used to pull the latest versions of images associated with a service defined in the Docker Compose file but does not start the containers based on those images. This command accepts various options. Use `docker compose pull --help` command for the complete list of options.

```
docker compose pull
```

- `docker compose push`: It is used to push images built locally for services to the respective registry/repository assuming access to the build key. This command accepts various options. Use `docker compose push --help` command for the complete list of options.

```
docker compose push
```

- `docker compose images`: It is used to list images used by containers. This command accepts various options such as:
  - `-f` to format the output
  - `-q` to display image IDs

```
docker compose images <options>
```

- `docker compose create`: It is used to create containers for a service without starting them. This command accepts various options. Use `docker compose create --help` command for the complete list of options.
  Some common options are:
  - `--build` to build images before starting containers
  - `--pull` to pull image before running
  - `--scale` to scale service to number of instances

```
docker compose create
```

- `docker compose containers`: It is used to list containers.

```
docker compose containers
```

- `docker compose ps`: It is used to list containers and services including their names, commands, states and ports. This command accepts various options. Use `docker compose ps --help` command for the complete list of options.
  Some common options are:
  - `-a` to show all stopped containers a shutdown timeout in seconds
  - `-q` to show only container IDs
  - `--services` to display services

```
docker compose ps <options>
```

- `docker compose ls`: It is used to list all running compose objects. This command accepts various options such as:
  - `-a` to show all stopped compose objects the output
  - `-f` to format the output
  - `--filter` to filter the output based on conditions
  - `-q` to display project names

```
docker compose ls <options>
```

- `docker compose top`: It is used to display running processes

```
docker compose top
```

- `docker compose run`: It is used to run a one-time command against a service. This command accepts various options. Use `docker compose run --help` command for the complete list of options.
  Some common options are:
  - `--build` to build images before starting containers
  - `--pull` to pull image before running
  - `-u` to run as a specified username
  - `-v` to bind mount a volume
  - `-w` to set working directory inside container

```
docker compose run <options> <service_name> <command>
```

- `docker compose exec`: It is used to execute a command in a running container. This command accepts various options. Use `docker compose exec --help` command for the complete list of options.
  Some common options are:
  - `-d` to run command in background
  - `–e` to set environment variables
  - `-u` to run as a specified username
  - `-w` to set working directory inside container

```
docker compose exec <options> <service_name> <command>
```

- `docker compose logs`: It is used to view log output from containers. This command accepts various options. Use `docker compose logs --help` command for the complete list of options.
  Some common options are:
  - `-f` to follow the log output
  - `–n` to show number of lines from the end of logs

To view logs for specified service:

```
docker compose logs <options> <service_name>
```

To view logs for all services:

```
docker compose logs <options>
```

- `docker compose pause`: It is used to pause running containers of a service.

  To pause for specified service:

  ```
  docker compose pause <service_name>
  ```

  To pause all service:

  ```
  docker compose pause
  ```

- `docker compose unpause`: It is used to unpause paused containers of a service.

  To unpause for specified service:

  ```
  docker compose unpause <service_name>
  ```

  To unpause all service:

  ```
  docker compose unpause
  ```

- `docker compose stop`: It is used to stop running containers without removing them.

  To stop for specified service:

  ```
  docker compose stop <service_name>
  ```

  To stop all services:

  ```
  docker compose stop
  ```

- `docker compose start`: It is used to start containers for all services or the specified service only.

  To start for specified service:

  ```
  docker compose start <service_name>
  ```

To start all service:

```
docker compose start
```

- `docker compose restart`: It is used to restart all stopped and running services or the specified services only.

  To restart for specified service:

  ```
  docker compose restart <service_name>
  ```

  To restart all service:

  ```
  docker compose restart
  ```

- `docker compose kill`: It is used to force stop running containers.

  To kill for specified service:

  ```
  docker compose kill <service_name>
  ```

  To kill all services:

  ```
  docker compose kill
  ```

- `docker compose rm`: It is used to remove stopped service containers. By default, anonymous volumes attached to containers are not removed but can be removed with `-v` option.

  ```
  docker compose rm
  ```

- `docker compose up`: It is used to build, create, start and attach to containers for a service. Once services are built, they are tagged as project-service, by default. This command accepts various options. Use `docker compose up --help` command for the complete list of options.
  Some common options are:
  - `--build` to build images before starting containers
  - `--attach` to attach subset of services
  - `-d` or `--detach` to start containers in the background and leave them running
  - `--pull` to pull image before running

- `--scale` to scale number of services

```
docker compose up <options>
```

- `docker compose down`: It is used to stop and remove images, containers, networks, and volumes created by `docker compose up` command. This command accepts various options. Use `docker compose down --help` command for the complete list of options. Some common options are:
  - `-t` to specify a shutdown timeout in seconds
  - `-v` to remove named volumes

```
docker compose down <options>
```

13.3 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, we would require three containers:

- 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

We will create and manage these containers using Docker Compose.

13.3.1 STOP AND DELETE CONTAINERS

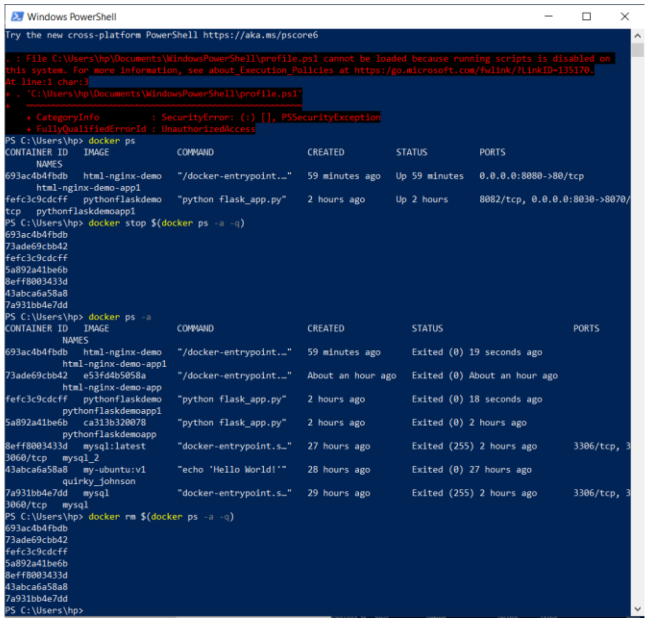
Before using Docker Compose, let us first stop all running containers and remove all containers and delete all available images along with any saved volumes.

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

Stop and delete all containers:

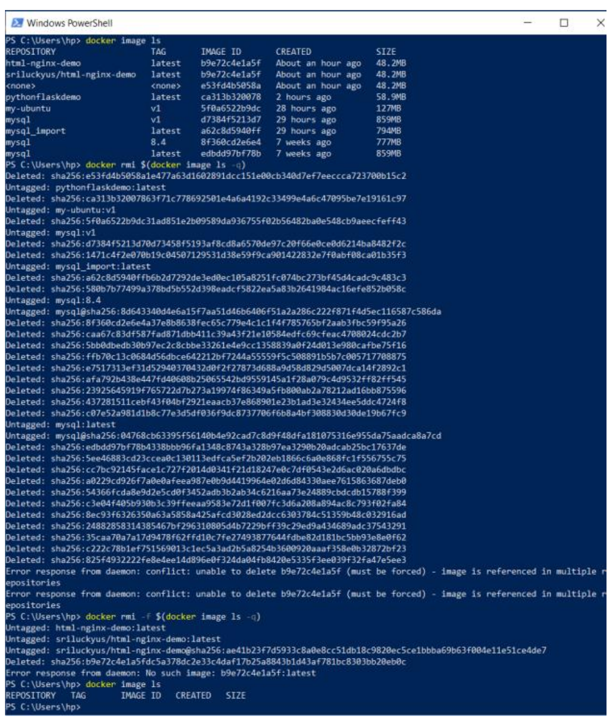
```
docker ps
docker stop $(docker ps -q)
docker ps -a
```

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



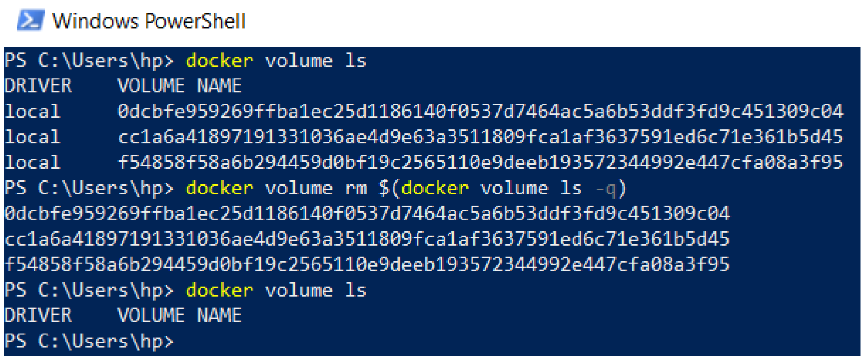
Delete all images:

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



Delete all volumes:

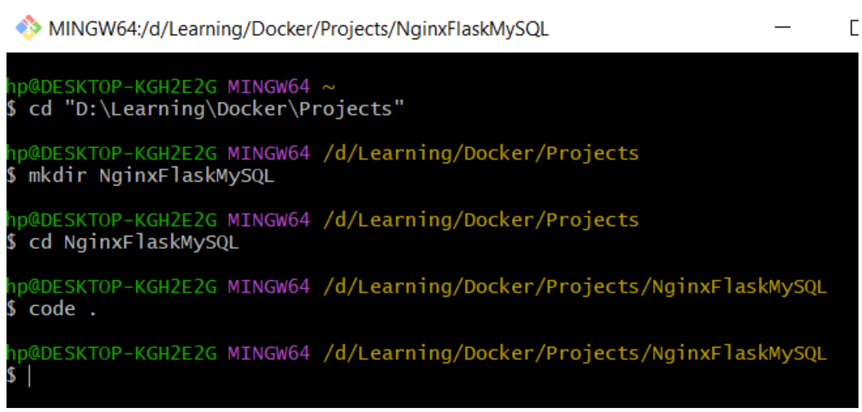
```
docker volume ls
docker volume rm $(docker volume ls -q)
docker volume ls
```



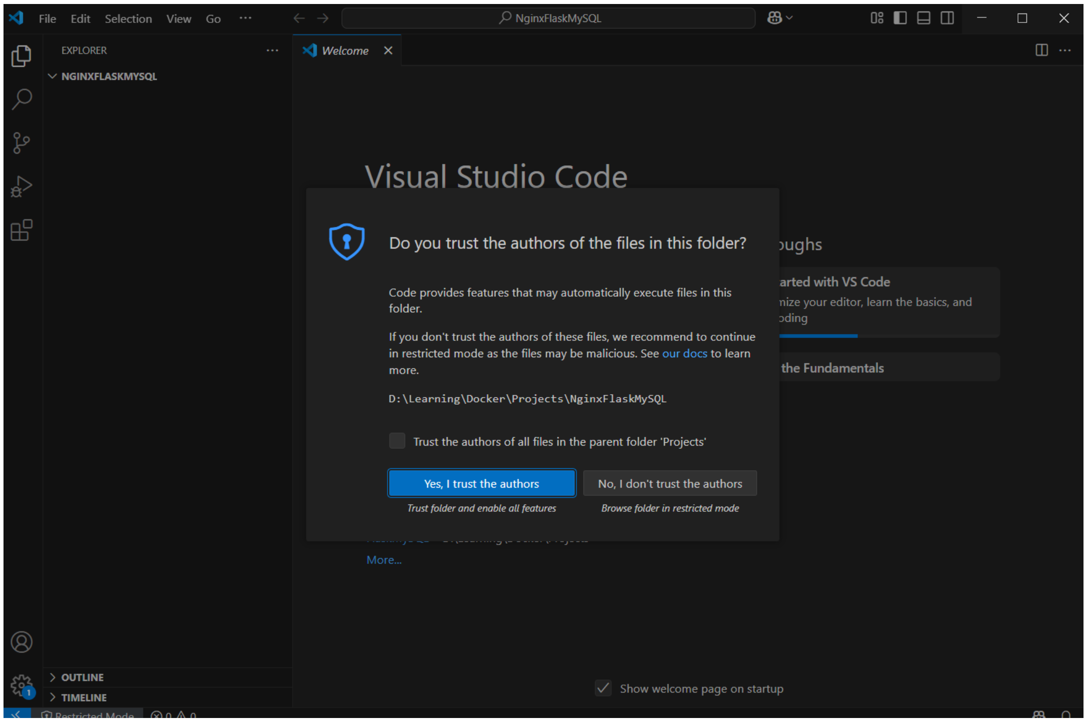
13.3.2 LAUNCH VS CODE

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 .
```



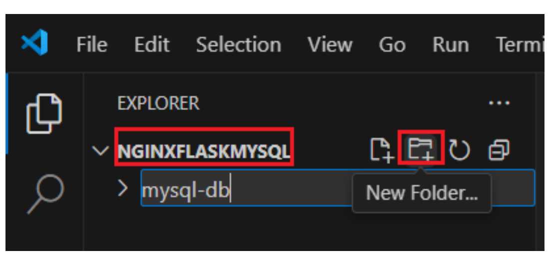
It opens **VS Code** application which might prompt you to confirm if you trust the authors of file in which case, click on **Yes, I trust the authors** button.



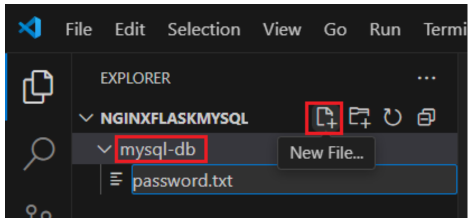
13.3.3 CONFIGURE MYSQL DATABASE

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`

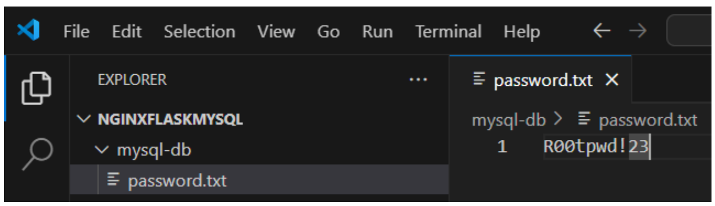


Select the `mysql-db` folder and click on **New File** icon and name it as `password.txt`



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

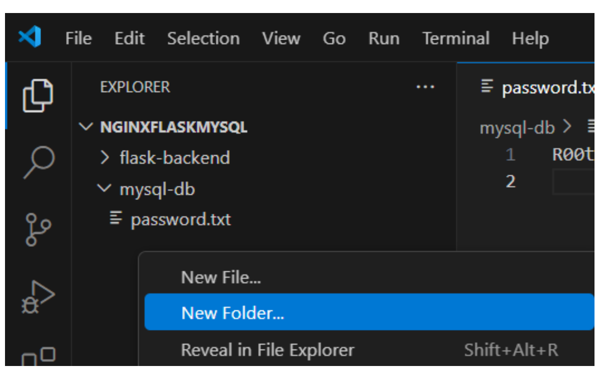
```
R00tpwd!23
```



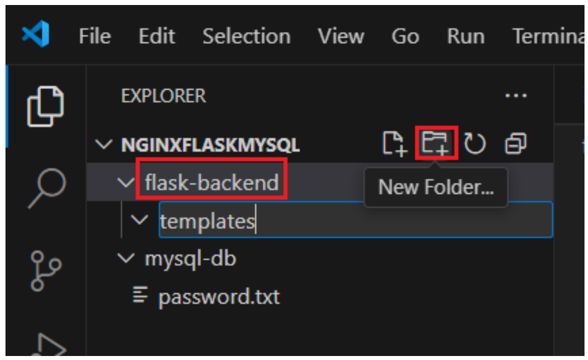
13.3.4 STOP AND DELETE CONTAINERS

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.

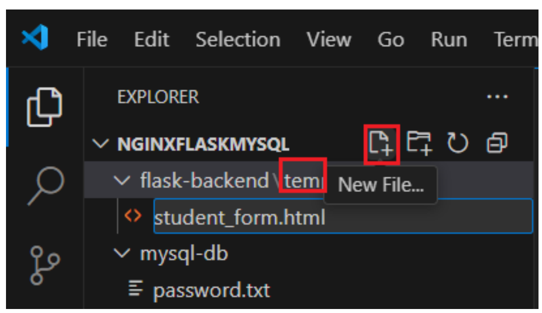
Under `NGINXFLASKMYSQL` project, right click on the empty space and select **New Folder** and name the folder as `flask-backend`



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`



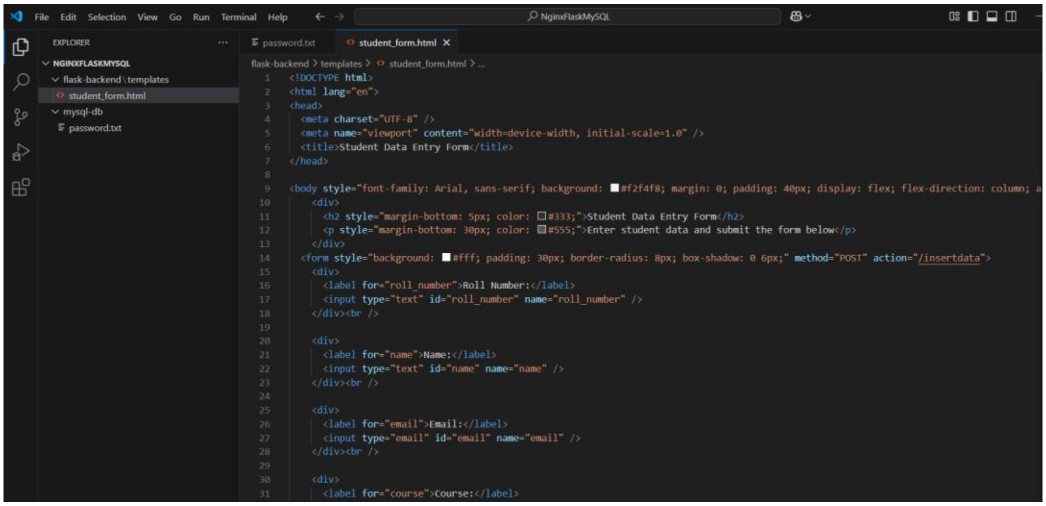
Enter the following code in `student_form.html` file and save it:

```
<!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>
```

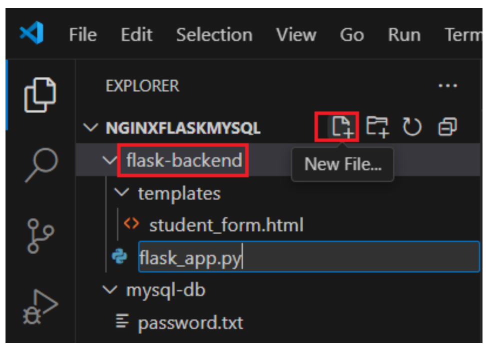
```
  <form style="background: #fff; padding: 30px; border-radius: 8px; box-
shadow: 0 6px;" method="POST" action="/insertdata">
    <div>
      <label for="roll_number">Roll Number:</label>
      <input type="text" id="roll_number" name="roll_number" />
    </div><br />

    <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 to call `/insertdata` page up on clicking **Submit** button.



Now, select `flask-backend` folder 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')
```

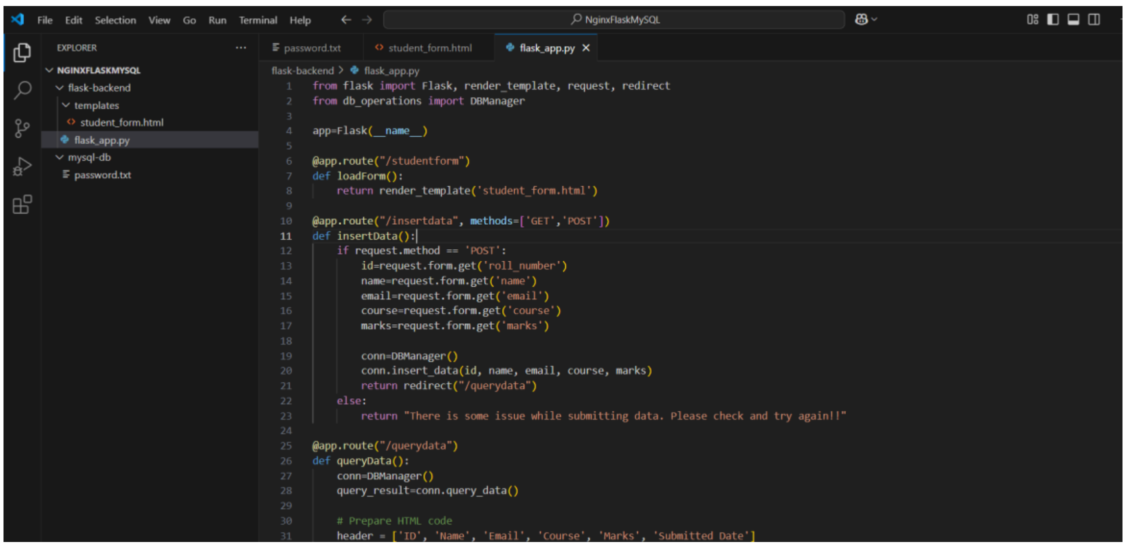
```
        conn=DBManager()
        conn.insert_data(id, name, email, course, marks)
        return redirect("/querydata")
    else:
        return "There is some issue while submitting data. Please check and
try again!!"

@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)*

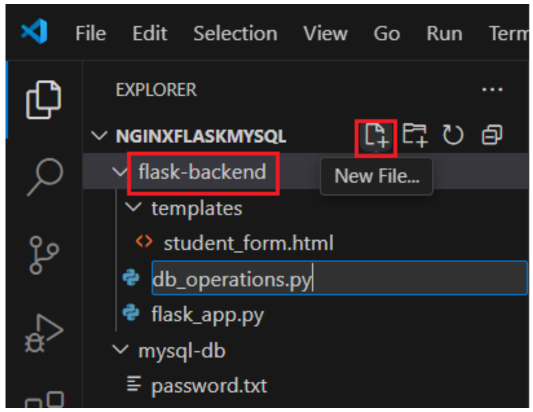
- `/insertdata` route with a Python function that verifies the method. If the method is `POST`, it reads values stored in form ids named `roll_number`, `name`, `email`, `course` and `marks` and creates `DBManager` object to call `insert_data` method with values read from the form and then redirects to `/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`



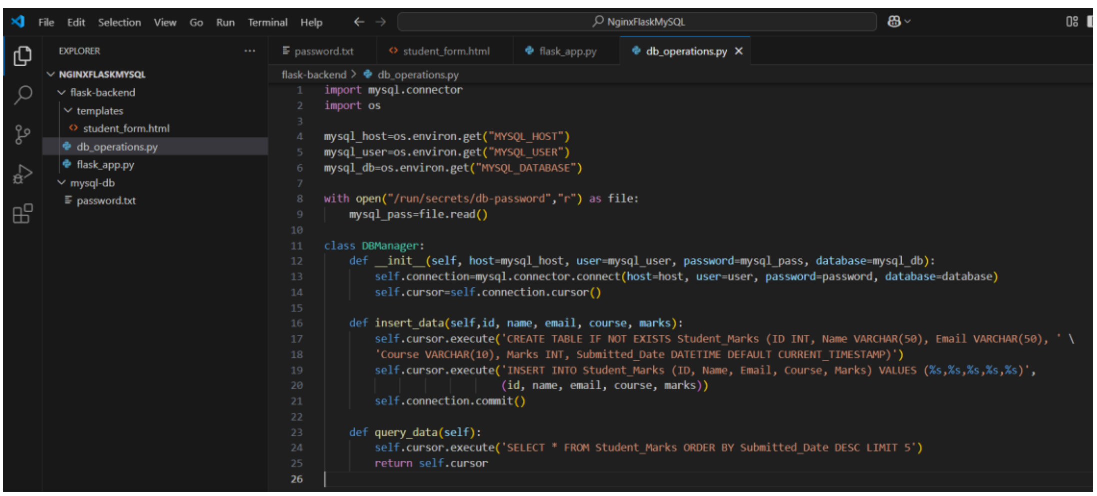
Enter the following code in `db_operations.py` file and save it:

```
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.



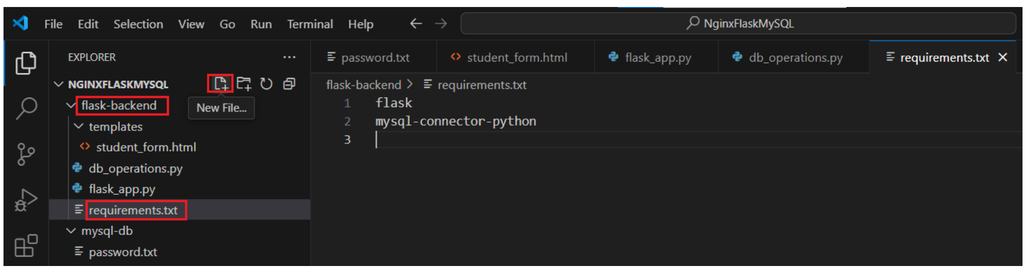
Again, select the `flask-backend` folder and click on **New File** icon and name it as `requirements.txt` in which write the following lines and save the file. This file is used to install Python libraries specified in the file.

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

The above code performs the following:

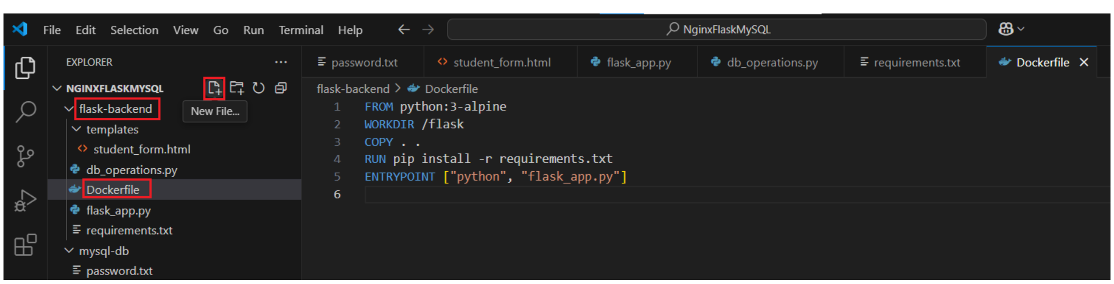
- Use `flask` library
- Use `mysql-connector-python` library which is mandatary for Flask to be able to connect to MySQL database.

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 newer versions 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 introduced in MySQL 8.0 version. If `mysql-connector`, `pymysql` libraries are needed, use either MySQL 8.0 or even lower version image or set `auth_plugin` to `mysql_native_password` in the new versions (8.6 and above) of MySQL image. For more details, refer this blog.



Now, let us create a Docker file for Python Flask application. Select `flask-backend` folder 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:

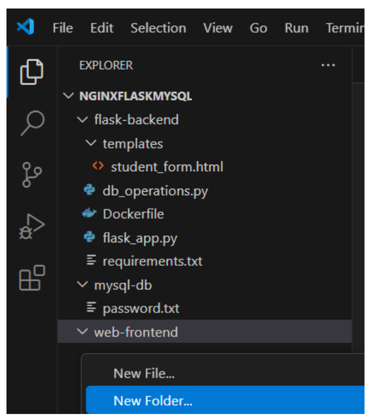
- Pull `python:3-alpine` image if not available in local host.

- 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.

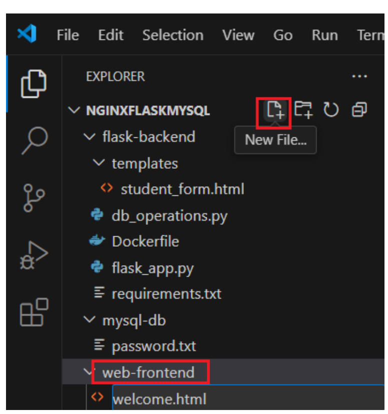
13.3.5 CONFIGURE WEB APP

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`

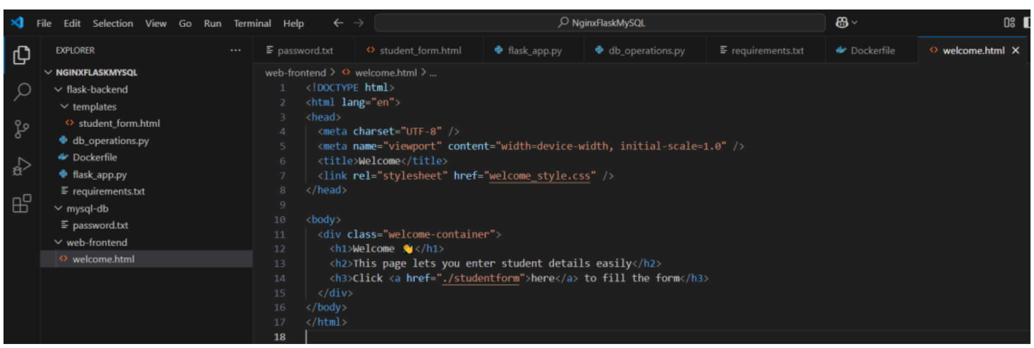


Enter the following code in `welcome.html` file and save it.

```
<!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:

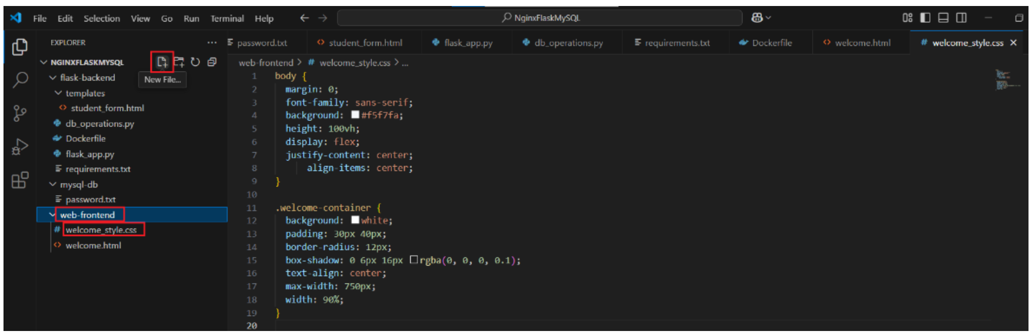
- Build a webpage with a title **Welcome**.
- Display a message to enter student details.
- Display a web link that calls `/studentform` page at background.



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;
```

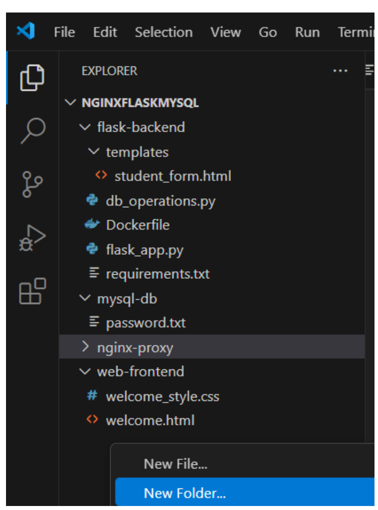
```
  display: flex;
  justify-content: center;
        align-items: center;
}

.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%;
}
```



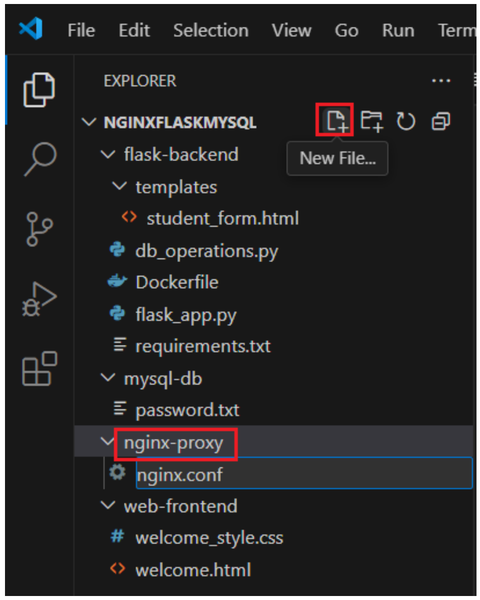
13.3.6 CONFIGURE NGINX SERVER

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`

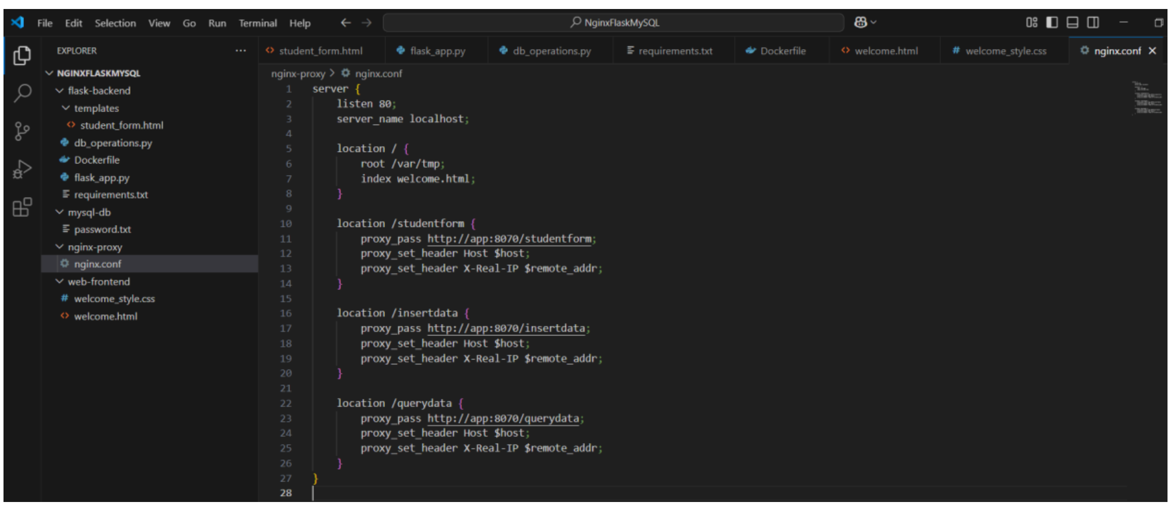


Enter the following code in `nginx.conf` file and save it.

```
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:

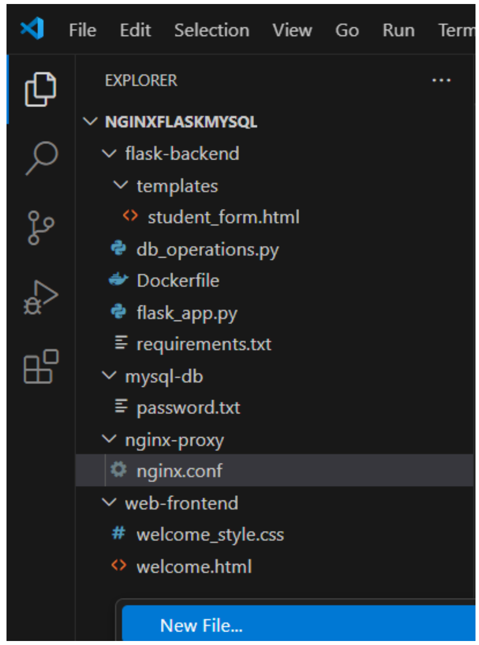
- Set the `server` to listen on port `80`.
- Set the `server_name` as `localhost`.
- Create 4 web locations:
    - `/` location that sets `root` directory as `/var/tmp` and calls `index` page `welcome.html`.
    - `/studentform` location that forwards request to http://app:8070/studentform webpage *(here* `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.



13.3.7 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:
```

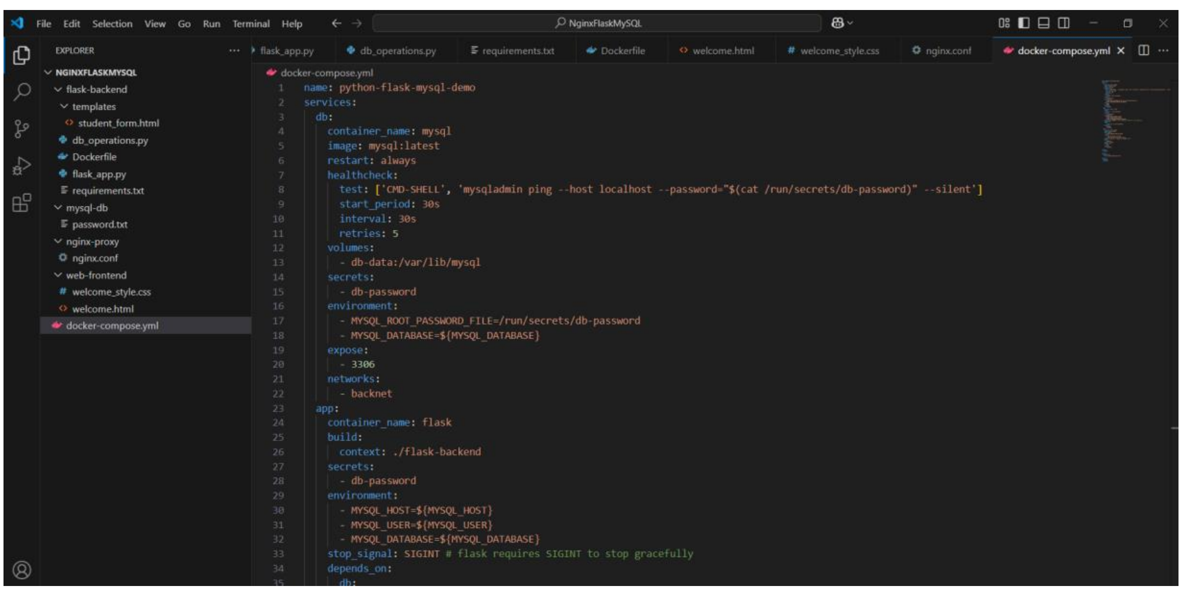
The above code performs the following:

- 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.

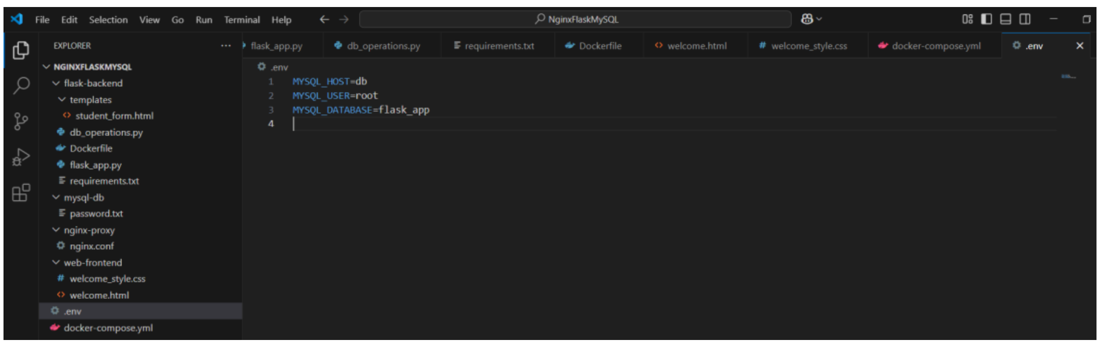


13.3.8 CREATE ENVIRONMENT FILE

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
```



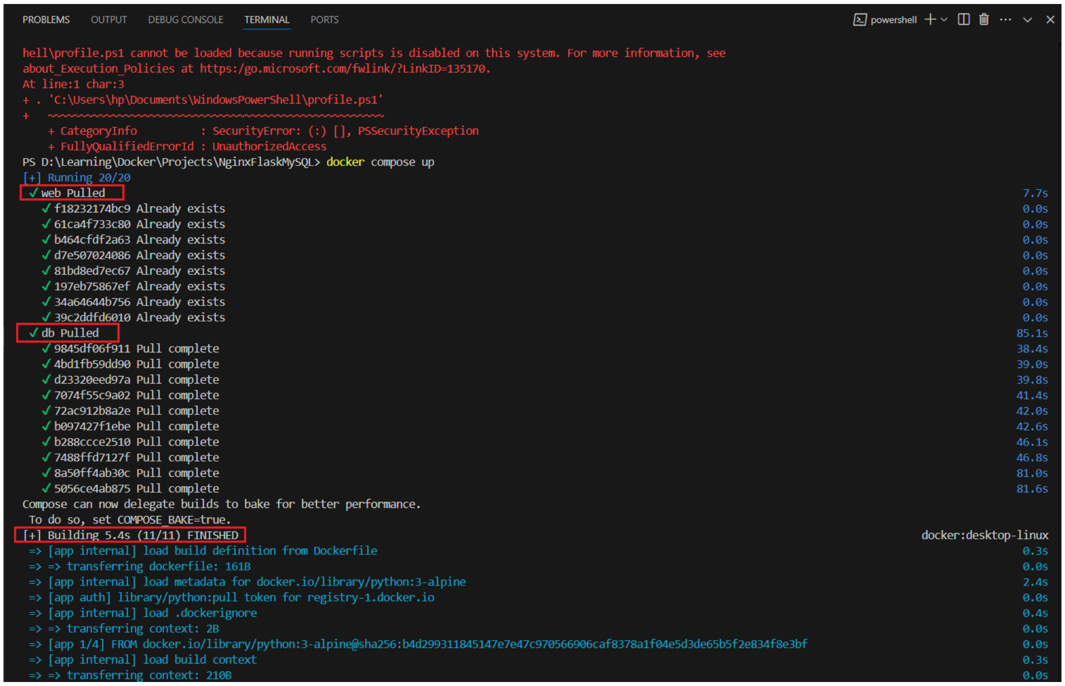
13.3.9 START DOCKER COMPOSE

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:

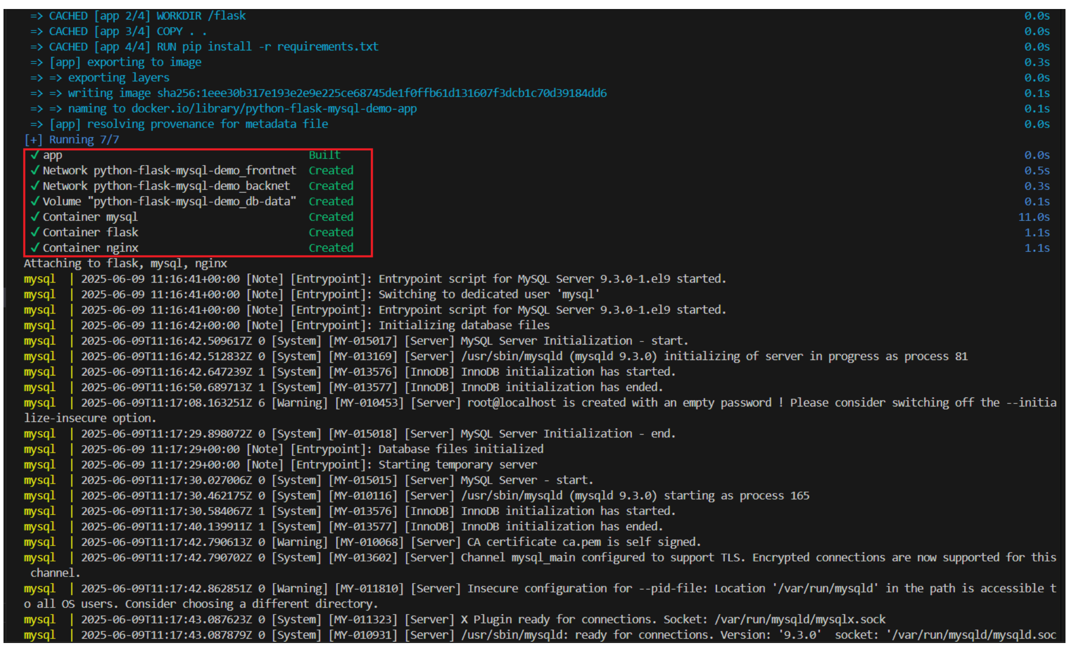
```
docker compose up
```

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



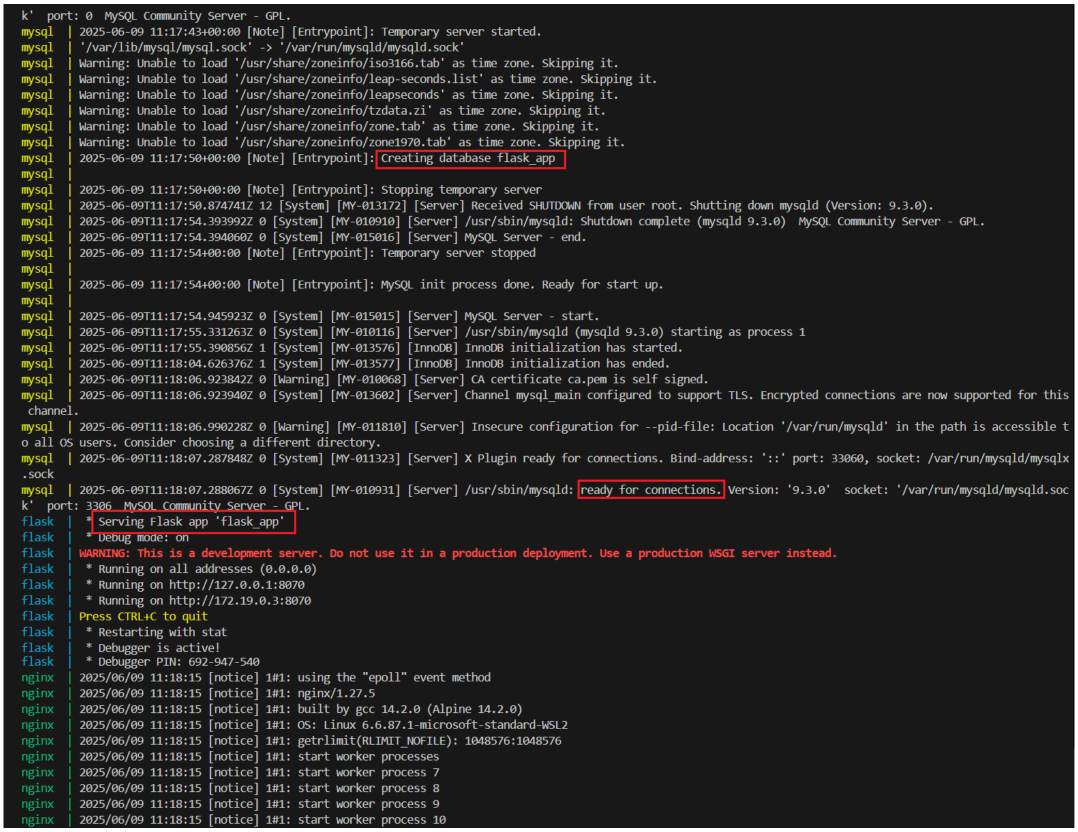
```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS                                  powershell
hell\profile.ps1 cannot be loaded because running scripts is disabled on this system. For more information, see
about_Execution_Policies at https:/go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:3
+ . 'C:\Users\hp\Documents\WindowsPowerShell\profile.ps1'
+   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
PS D:\Learning\Docker\Projects\NginxFlaskMySQL> docker compose up
[+] Running 20/20
 ✓ web Pulled                                                        7.7s
   ✓ f18232174bc9 Already exists                                     0.0s
   ✓ 61ca4f733c80 Already exists                                     0.0s
   ✓ b464cfdf2a63 Already exists                                     0.0s
   ✓ d7e507024086 Already exists                                     0.0s
   ✓ 81bd8ed7ec67 Already exists                                     0.0s
   ✓ 197eb75867ef Already exists                                     0.0s
   ✓ 34a64644b756 Already exists                                     0.0s
   ✓ 39c2ddfd6010 Already exists                                     0.0s
 ✓ db Pulled                                                        85.1s
   ✓ 9845df06f911 Pull complete                                     38.4s
   ✓ 4bd1fb59dd90 Pull complete                                     39.0s
   ✓ d23320eed97a Pull complete                                     39.8s
   ✓ 7074f55c9a02 Pull complete                                     41.4s
   ✓ 72ac912b8a2e Pull complete                                     42.0s
   ✓ b097427f1ebe Pull complete                                     42.6s
   ✓ b288ccce2510 Pull complete                                     46.1s
   ✓ 7488ffd7127f Pull complete                                     46.8s
   ✓ 8a50ff4ab30c Pull complete                                     81.0s
   ✓ 5056ce4ab875 Pull complete                                     81.6s
Compose can now delegate builds to bake for better performance.
 To do so, set COMPOSE_BAKE=true.
[+] Building 5.4s (11/11) FINISHED                                   docker:desktop-linux
 => [app internal] load build definition from Dockerfile             0.3s
 => => transferring dockerfile: 161B                                 0.0s
 => [app internal] load metadata for docker.io/library/python:3-alpine   2.4s
 => [app auth] library/python:pull token for registry-1.docker.io    0.0s
 => [app internal] load .dockerignore                                0.4s
 => => transferring context: 2B                                      0.0s
 => [app 1/4] FROM docker.io/library/python:3-alpine@sha256:b4d299311845147e7e47c970566906caf8378a1f04e5d3de65b5f2e834f8e3bf   0.0s
 => [app internal] load build context                                0.3s
 => => transferring context: 210B                                    0.0s
```

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

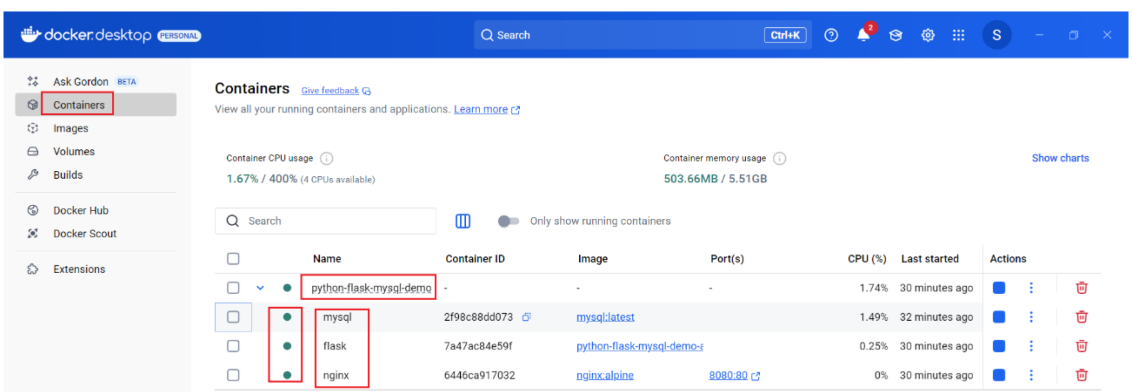


```
 => CACHED [app 2/4] WORKDIR /flask                                  0.0s
 => CACHED [app 3/4] COPY . .                                        0.0s
 => CACHED [app 4/4] RUN pip install -r requirements.txt             0.0s
 => [app] exporting to image                                         0.3s
 => => exporting layers                                              0.0s
 => => writing image sha256:1eee30b317e193e2e9e225ce68745de1f0ffb61d131607f3dcb1c70d39184dd6   0.1s
 => => naming to docker.io/library/python-flask-mysql-demo-app       0.1s
 => [app] resolving provenance for metadata file                     0.0s
[+] Running 7/7
 ✓ app                                         Built                 0.0s
 ✓ Network python-flask-mysql-demo_frontnet    Created               0.5s
 ✓ Network python-flask-mysql-demo_backnet     Created               0.3s
 ✓ Volume "python-flask-mysql-demo_db-data"    Created               0.1s
 ✓ Container mysql                             Created              11.0s
 ✓ Container flask                             Created               1.1s
 ✓ Container nginx                             Created               1.1s
Attaching to flask, mysql, nginx
mysql  | 2025-06-09 11:16:41+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 9.3.0-1.el9 started.
mysql  | 2025-06-09 11:16:41+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
mysql  | 2025-06-09 11:16:41+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 9.3.0-1.el9 started.
mysql  | 2025-06-09 11:16:42+00:00 [Note] [Entrypoint]: Initializing database files
mysql  | 2025-06-09T11:16:42.509617Z 0 [System] [MY-015017] [Server] MySQL Server Initialization - start.
mysql  | 2025-06-09T11:16:42.512832Z 0 [System] [MY-013169] [Server] /usr/sbin/mysqld (mysqld 9.3.0) initializing of server in progress as process 81
mysql  | 2025-06-09T11:16:42.647239Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
mysql  | 2025-06-09T11:16:50.689713Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
mysql  | 2025-06-09T11:17:08.163251Z 6 [Warning] [MY-010453] [Server] root@localhost is created with an empty password ! Please consider switching off the --initia
lize-insecure option.
mysql  | 2025-06-09T11:17:29.898072Z 0 [System] [MY-015018] [Server] MySQL Server Initialization - end.
mysql  | 2025-06-09 11:17:29+00:00 [Note] [Entrypoint]: Database files initialized
mysql  | 2025-06-09 11:17:29+00:00 [Note] [Entrypoint]: Starting temporary server
mysql  | 2025-06-09T11:17:30.027006Z 0 [System] [MY-015015] [Server] MySQL Server - start.
mysql  | 2025-06-09T11:17:30.462175Z 0 [System] [MY-010116] [Server] /usr/sbin/mysqld (mysqld 9.3.0) starting as process 165
mysql  | 2025-06-09T11:17:30.584067Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
mysql  | 2025-06-09T11:17:40.139911Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
mysql  | 2025-06-09T11:17:42.790613Z 0 [Warning] [MY-010068] [Server] CA certificate ca.pem is self signed.
mysql  | 2025-06-09T11:17:42.790702Z 0 [System] [MY-013602] [Server] Channel mysql_main configured to support TLS. Encrypted connections are now supported for this
 channel.
mysql  | 2025-06-09T11:17:42.862851Z 0 [Warning] [MY-011810] [Server] Insecure configuration for --pid-file: Location '/var/run/mysqld' in the path is accessible t
o all OS users. Consider choosing a different directory.
mysql  | 2025-06-09T11:17:43.087623Z 0 [System] [MY-011323] [Server] X Plugin ready for connections. Socket: /var/run/mysqld/mysqlx.sock
mysql  | 2025-06-09T11:17:43.087879Z 0 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections. Version: '9.3.0'  socket: '/var/run/mysqld/mysqld.soc
```

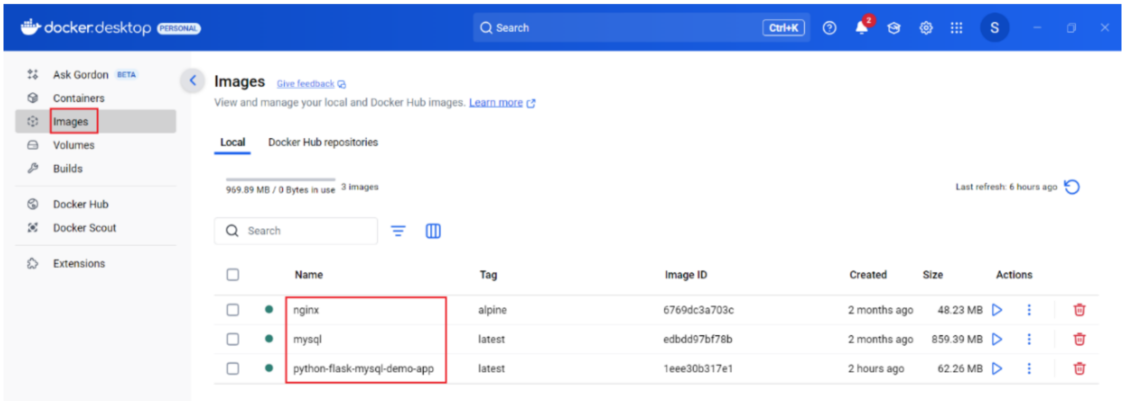
At bottom, we can see that `mysql` container is **creating database flask_app** and making the database **ready for connections**. Then, `flask` container is **serving flask_app** with **running on all addresses** and then brought up the `nginx` container.



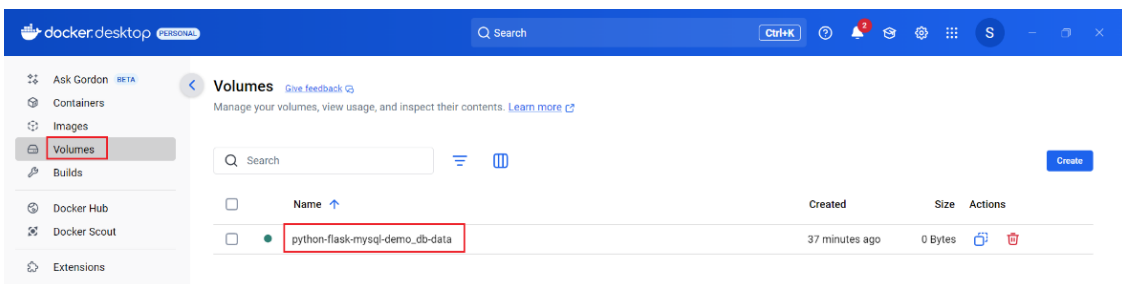
Open **Docker Desktop** and go to **Containers** tab where you can see that `python-flask-mysql-demo` compose stack has been created with three containers – `mysql`, `flask`, `nyginx` which are in **Running** state.



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



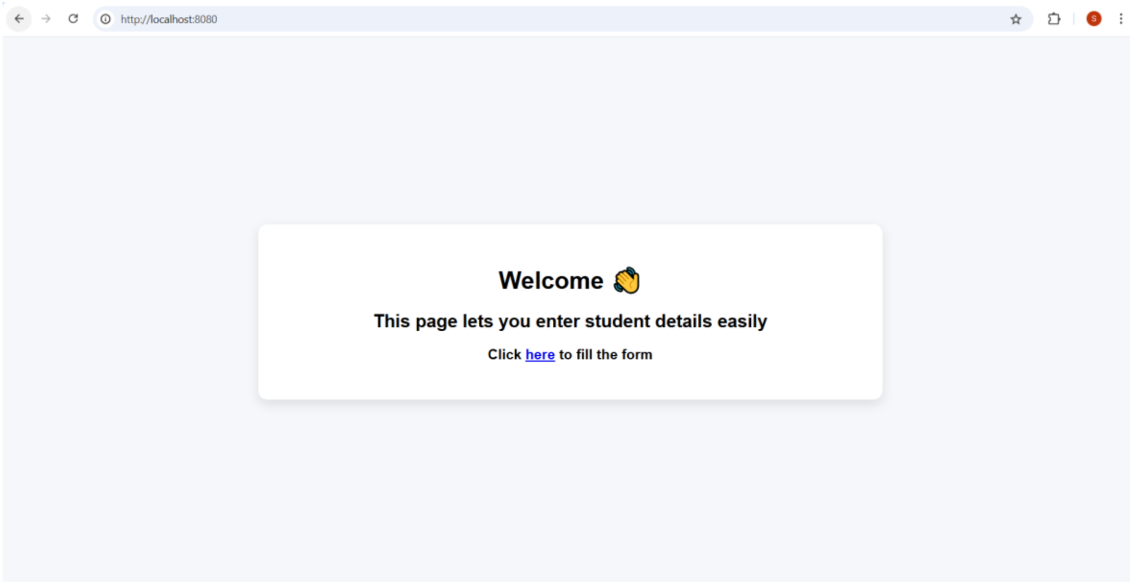
Under **Volumes** tab, you can see that `python-flask-mysql-demo_db-data` volume has been created.



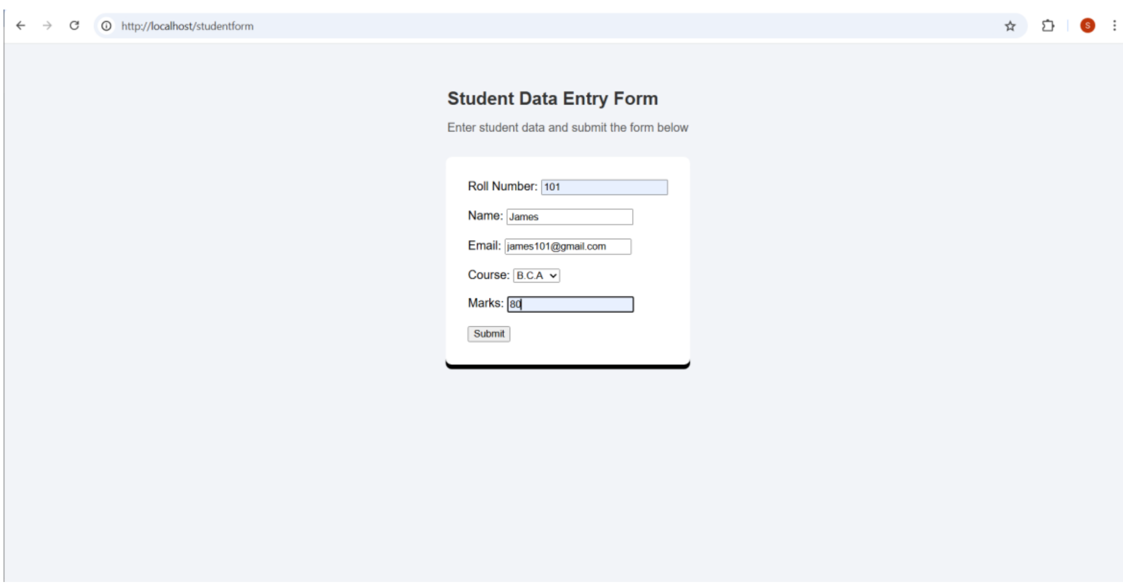
13.3.10 LAUNCH WEB APPLICATION

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

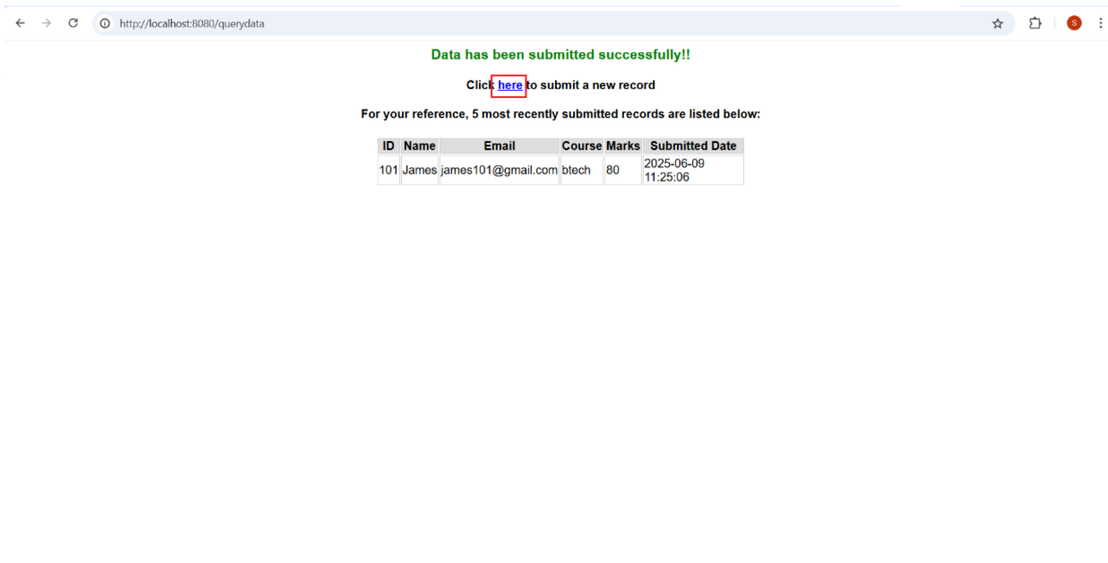
Open web browser and hit http://localhost/ or http://localhost:8080/ URL after which you can see the below welcome page



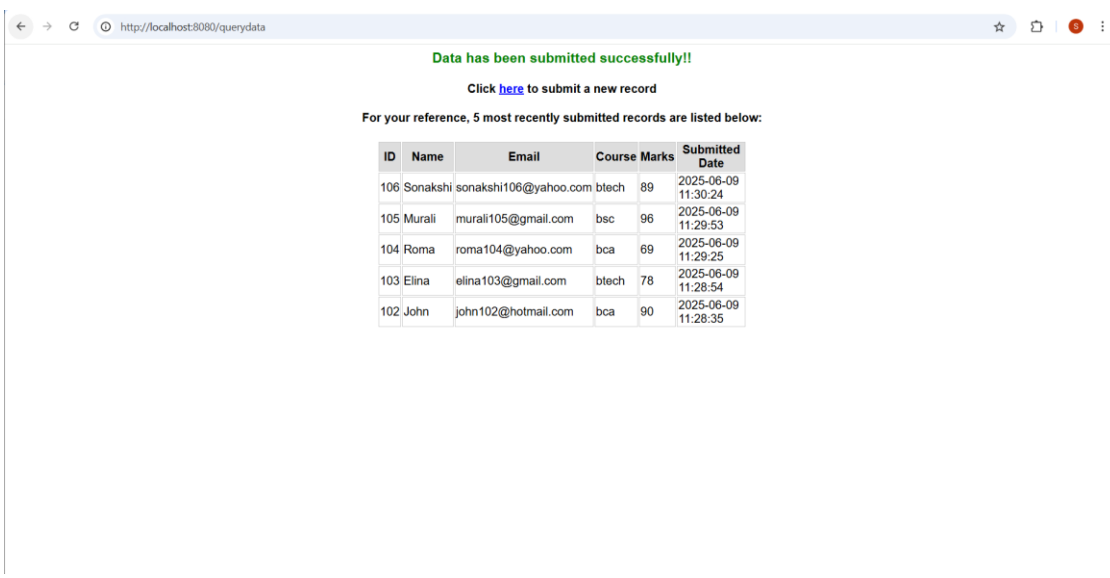
Click on here in the welcome page after which it displays the **Student Data Entry Form** where enter some data and click on **Submit** button.



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:

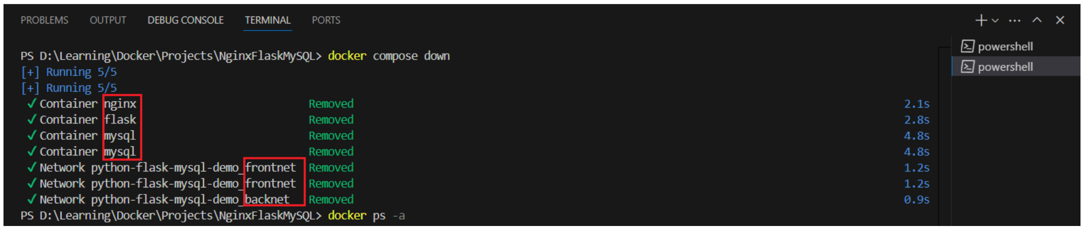


13.3.11 STOP DOCKER COMPOSE

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:

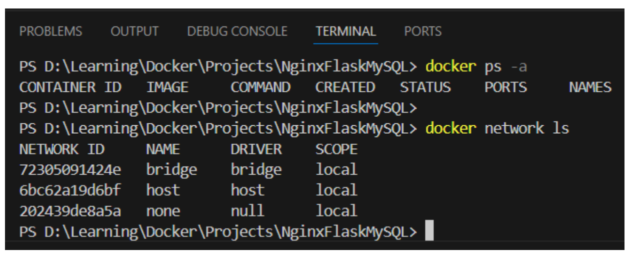
```
docker compose down
```



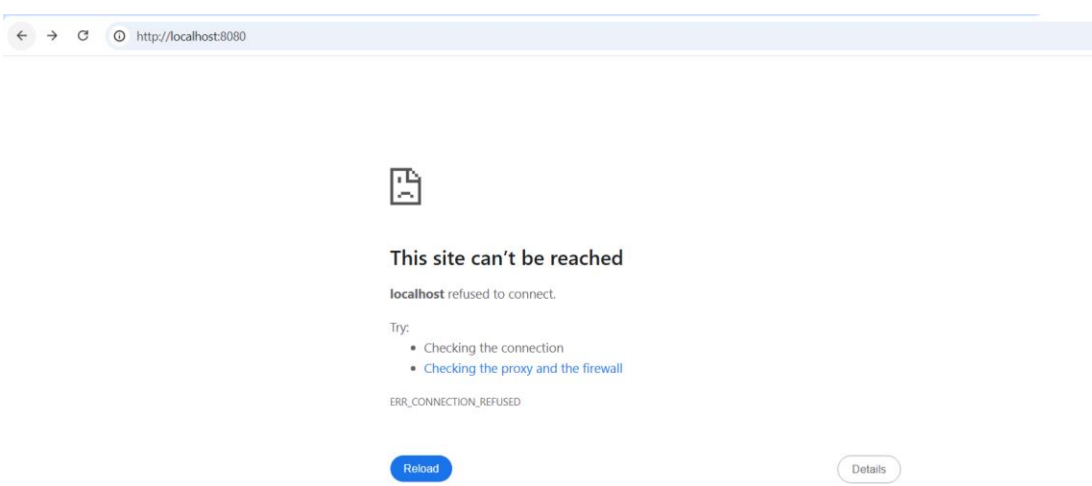
As you can see above, Docker Compose has removed `nginx`, `flask` and `mysql` containers and `frontnet` and `backnet` networks.

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

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



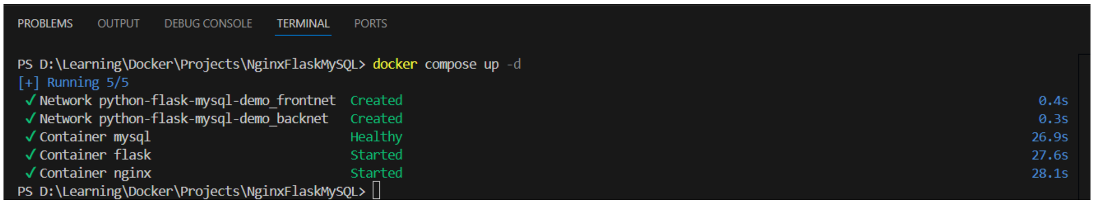
Refresh the web browser to see that application is down.



13.3.12 START DOCKER COMPOSE DETACHED

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

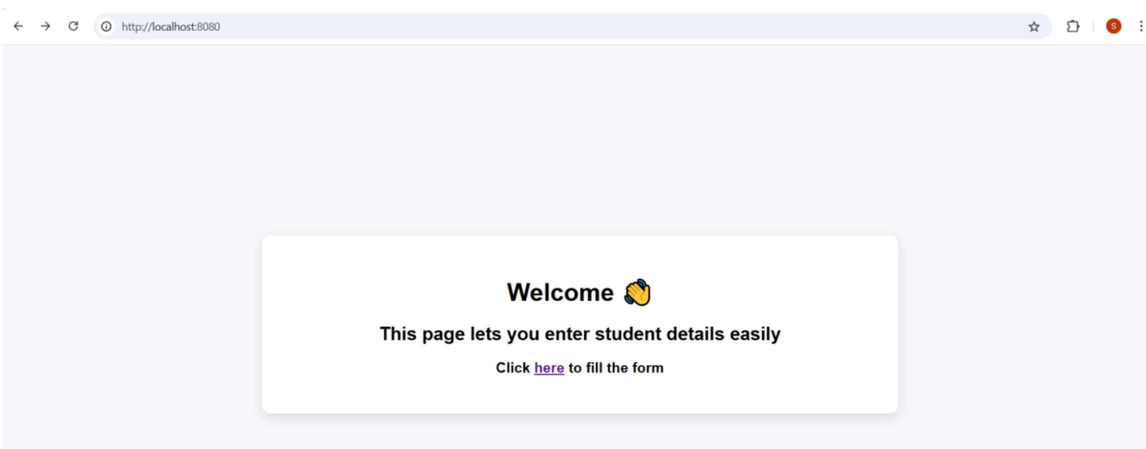
```
docker compose up -d
```



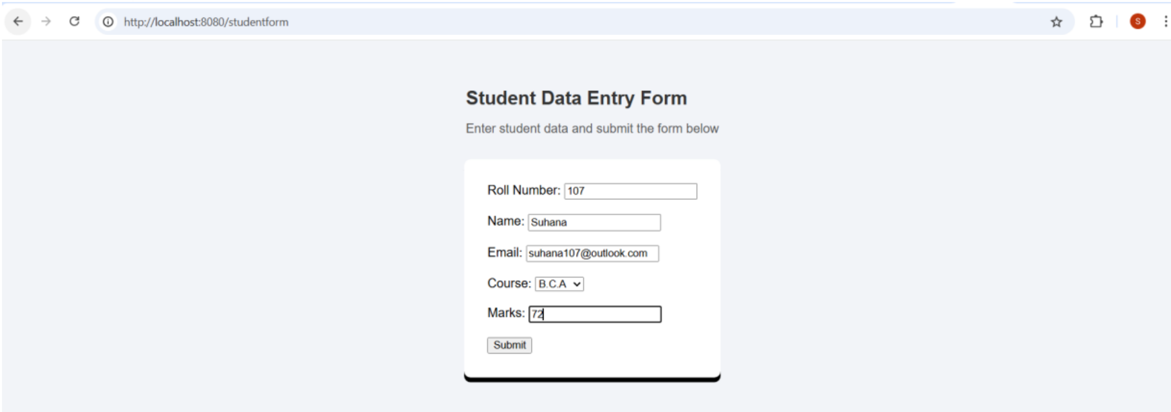
As you see above, the container logs have been suppressed and not being displayed on the console.

13.3.13 VERIFY WEB APPLICATION

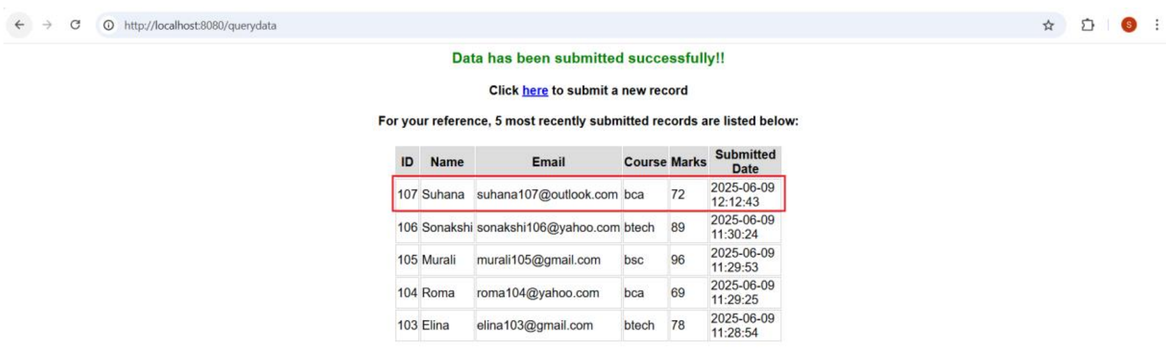
Refresh the web page http://localhost/ or http://localhost:8080/ and we can see that the application is up.



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



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



ID	Name	Email	Course	Marks	Submitted Date
107	Suhana	suhana107@outlook.com	bca	72	2025-06-09 12:12:43
106	Sonakshi	sonakshi106@yahoo.com	btech	89	2025-06-09 11:30:24
105	Murali	murali105@gmail.com	bsc	96	2025-06-09 11:29:53
104	Roma	roma104@yahoo.com	bca	69	2025-06-09 11:29:25
103	Elina	elina103@gmail.com	btech	78	2025-06-09 11:28:54

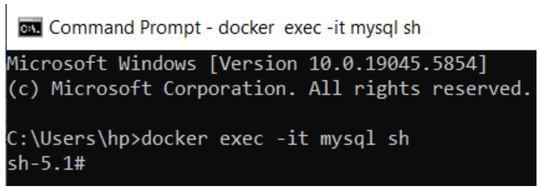
Though the containers have been re-created, it persisted old data since we configured in Docker Compose file to create a volume for data storage.

13.3.14 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
```

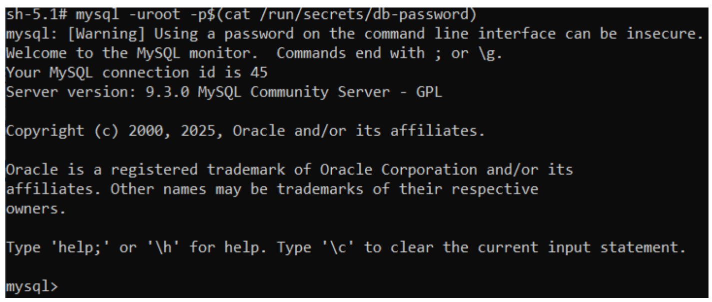


Command Prompt - docker exec -it mysql sh

```
Microsoft Windows [Version 10.0.19045.5854]
(c) Microsoft Corporation. All rights reserved.

C:\Users\hp>docker exec -it mysql sh
sh-5.1#
```

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)
```



```
sh-5.1# mysql -uroot -p$(cat /run/secrets/db-password)
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 45
Server version: 9.3.0 MySQL Community Server - GPL

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

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>
```

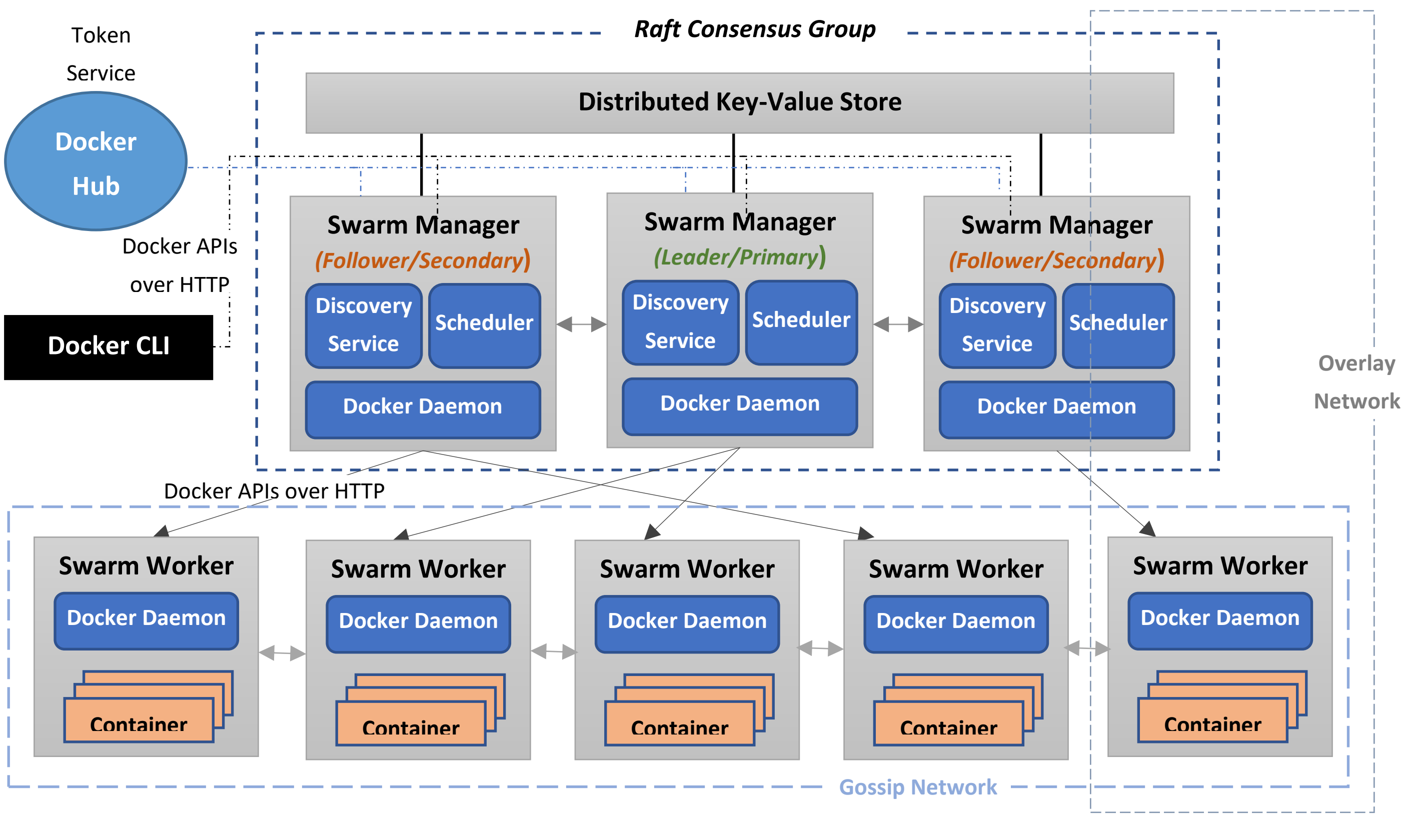
14 DOCKER SWARM

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 features of Docker Swarm include:

1. **Easy to Use:** Docker Swarm mode is integrated with Docker CLI enabling users to create or manage swarm without any additional orchestration software.
2. **Decentralized Architecture**: The Docker Engine manages node roles at runtime instead of deployment time. You can use Docker Engine to deploy both manager and worker nodes, making it possible to create an entire swarm from a single disk image.
3. **Default Security:** All communications between Swarm manager and worker nodes are encrypted. Each node in the swarm uses TLS for mutual authentication and encryption to ensure secure communication between itself and all other nodes.

4. **Easy Scalability**: Each service can specify the number of tasks to run. When scaling up or down, the swarm manager adjusts by adding or removing tasks to match the desired state.
5. **Auto Load Balancing:** Efficient distribution of incoming requests among service replicas to optimize performance. Swarm manager automatically assigns a published port for the service and allows to expose ports to an external load balancer. Swarm uses **`ingress`** load balancing to make services available outside of the swarm.
6. **Service Discovery:** Automated resolution of service names to IP addresses, enabling seamless inter-service communication. The swarm manager assigns each service a unique DNS name and handles load balancing for running containers. Any container in the swarm can be found using a DNS server integrated into the swarm.
7. **High Availability:** Automatic detection and replacement of failed containers to ensure uninterrupted service. If a node fails, Docker Swarm can automatically assign the node's tasks to other nodes, ensuring that services remain available and minimizing downtime.
8. **Fault Tolerance:** Management of distributed nodes to prevent downtime even during partial system failures.



Docker Swarm uses **Overlay networks** to manage communications among the Docker daemons participating in the swarm. The `overlay` network driver is installed on top of the host network and creates a distributed network amongst the group of nodes running Docker.

Docker Swarm consists of two types of nodes – **Manager node** (Swarm Manager) and **Worker node** (Swarm Worker). Docker Swarm must have at least one manager node *(which acts as a* *worker node by default)* to deploy applications to the cluster but Swarm cannot have a worker node without at least one manager node.

- **Manager node** or **Swarm Manager** receives commands from clients through Docker APIs and assigns tasks to Worker nodes. It also takes of cluster management tasks like maintaining the cluster state, scheduling services and serving the swarm mode HTTP API end points. Multiple manager nodes are needed to maintain the high availability and recover from a single manager failure.

  Manager node consists of five components which are **API**, **Orchestrator**, **Allocator**, **Scheduler**, and **Dispatcher**.
    - **API** accepts commands from Docker client and creates services object.
    - **Orchestrator** is used for evaluation of cluster by comparing the desired cluster state with the current cluster state. For example, if user wants to add one more instance of a container, the orchestrator checks the cluster and creates a new task that needs to be assigned to a worker node.
    - **Allocator** provides resources by allocating IP address for the new task that Orchestrator created.
    - **Scheduler** is responsible for assigning tasks to worker nodes. It constantly keeps track of whether a new task has been created and finds suitable nodes to assign tasks to. Scheduler looks for resources on the worker node such as available memory, occupied CPU, number of containers running, etc. and filters nodes accordingly. It uses three types of scheduling strategies – **Spread Strategy** which is a default strategy that looks for the node with minimal number of containers running so containers are spread across nodes, **Binpack Strategy** which looks for the node where most containers are running so instead of spreading containers across nodes, it fills one node first before moving to other and **Random Strategy** which selects nodes randomly. Swarm filters allow the scheduler to eliminate some of the nodes when a container needs to be launched. Such Swarm filter criteria could be running two containers on same host or running containers on nodes meeting constraints including health checks, storage, etc.

      For example, the following command launches `container2` on the node wherever `container1` is running. Here, `affinity` Swarm filter ensures both containers are running on same node.

      ```
      docker tcp://<manager_ip>:<manager_port> run -d --name
      <container2_name> -e affinity:container==<container1_name>
      <image_name>
      ```

    - **Dispatcher** forwards the task to the worker node identified by the scheduler. All worker nodes connect to the dispatcher and each worker reports their information such as available resources, number of containers run, etc. A constant communication takes place between the worker and dispatcher on the manager node, called as **heartbeat**, so that the manager is aware of if the worker node is still alive or down.

- **Worker node** or **Swarm Worker** do not involve in managing tasks but just executes tasks assigned by the manager node and runs the actual containers. All worker nodes communicate with the manager node using Docker APIs over HTTP and talk to each other using the **Gossip** network protocol. Each worker node gossip (broadcast information) about the running service with other worker node configured over specific **overlay** network so that all workers in a specific overlay network are aware of where particular service is running.

  Worker node consists of two components which are **Worker** and **Executor**.
  - **Worker** connects to the dispatcher component on manager node and periodically reports the status of current running tasks and its heartbeat to the dispatcher and also checks if there are new tasks assigned to it.
  - **Executor** executes tasks assigned to the worker node and launches containers.

14.1 Raft Consensus Algorithm

Docker Swarm can have multiple managers running in a cluster to support fault-tolerance and high availability. Docker Swarm uses **Raft Consensus Algorithm** to coordinate between manager nodes and maintain the cluster state.

Consensus algorithm is a protocol that enables multiple nodes in a cluster to agree on a single data value or decision, ensuring consistency and reliability across the cluster even if there are failures of some nodes. There are many consensus algorithms exist such as Raft, Paxos, Zookeeper Automatic Broadcast (ZAB), Proof-of-Work (PoW), Proof-of-Stake (PoS), etc.

Raft consensus algorithm ensures all nodes in a cluster agree on the system's current state, even in the presence of failures or network partitions. Raft achieves consistency using **leader election** and **replicated log** mechanism. In Raft, one node acts as a leader which manages all client requests and ensures data consistency, while other nodes act as followers which follow leader instructions and when the leader fails, followers elect a new leader by voting. Log replication in Raft works such that leader receives updates from client and stores in a log and replicates those log entries to followers and once a majority (quorum) nodes acknowledges, the data is committed which ensures all nodes have same log entries.

In a Docker Swarm, manager nodes use **Raft** to elect a **leader** among themselves and the other manager nodes act as **followers**. The leader is responsible for scheduling tasks, managing state updates, and replicating log entries (that represent changes to the cluster state) to follower nodes ensuring all nodes maintain a consistent view of system. When the leader fails for some reason, the other manager nodes will detect the failure and elect a new leader which keeps the swarm in a consistent state. Raft requires the majority (also called a **quorum**) of manager nodes to agree on any state change to swarm such as node additions or deletions. Raft can tolerate up to **(N-1)/2** manager nodes failure in a cluster of N manager nodes to avoid any conflicting states and ensure data integrity even during network partitions. For example, a five-manager swarm can tolerate the failure up to two manager nodes without disrupting operations. Docker recommends to have odd number (3 or 5 or 7) of manager nodes with a maximum of seven.

Docker Swarm implements Raft using **`etcd`** library which is an open-source, reliable, robust and **distributed key-value store** that is built on the Raft consensus algorithm and is used to store configuration data and cluster state, coordinate distributed systems, and facilitate service discovery. However, Docker Swarm allows to configure a different distributed store such as **Zookeeper**. Managers in Docker swarm share the distributed key-value store and also maintain a local copy of this store which contains configuration and other information needed for manager nodes.

Raft logs in Docker Swarm are encrypted since they contain the secret that is used to securely transfer the sensitive information to the containers running on Swarm. Raft logs are encrypted using private key to ensure secure TLS communication between nodes. The encrypted Raft logs can be decrypted using `swarm-rafttool` from SwarmKit.

14.2 Services vs Tasks

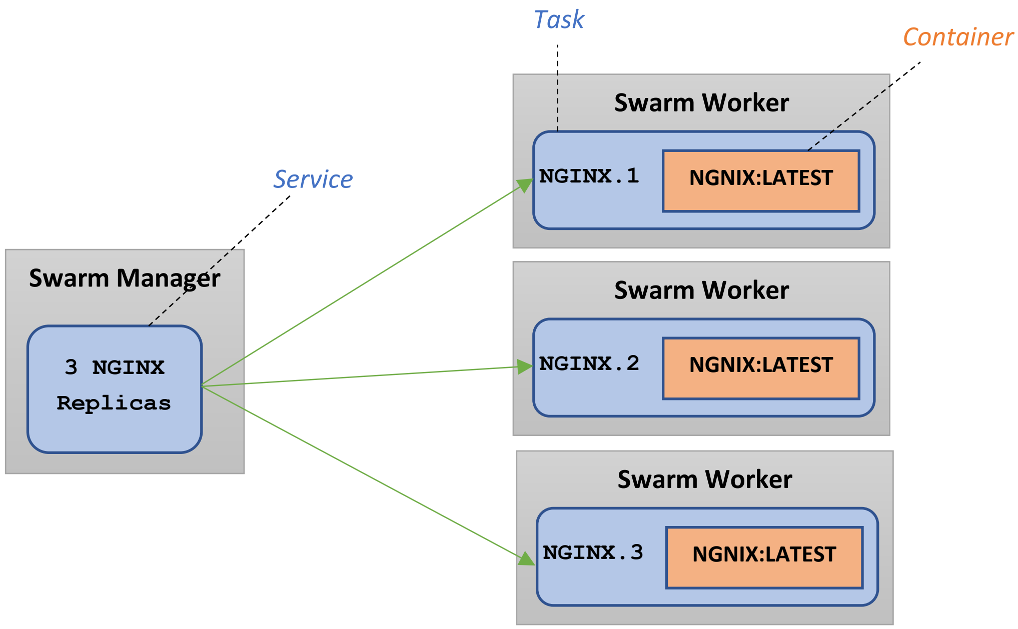
In Docker Swarm, a **Service** is an abstraction to deploy container workloads. A service defines the desired state of containers by specifying the container image, number of replicas, network settings and resource limits. Services can be used to deploy any application, such as web servers, databases, or messaging queues. Services can be accessed by any of the nodes in the same cluster.
In simpler terms, a service is the description of tasks to execute on manager or worker nodes. It is the central part of the Swarm system and primary way of user interaction with the Swarm. When we create a service, we specify which container image to use and which commands to execute inside running containers.

Services can be deployed to in Docker Swarm in two different ways:

- **Global services** enable the swarm manager to run one instance of a task for a service on every available node in the cluster.
- **Replicated services** enable the swarm manager to distribute a specified number of replica tasks across nodes in the cluster, ensuring high availability and fault tolerance.

A **Task** is the smallest scheduling unit of service in Swarm. A collection of tasks forms a service. A task represents a slot where the scheduler can place a container. Once the container is live, the scheduler recognizes that the task is in a running state.

In simpler terms, a task carries a Docker container and commands to run inside the container. Tasks are assigned by the manager node to worker nodes according to the number of replicas set in the service and the worker nodes will run tasks and report on their status. Once a task is assigned to a node, it can only run on the assigned node or fail but cannot be assigned to another node. Each task progresses sequentially through a series of stages which are assigned, prepared, running, etc.



14.3 Swarm Commands

The commonly used Swarm mode CLI commands are:

14.3.1 DOCKER SWARM COMMANDS

The `docker swarm` commands are used to manage the Docker swarm.

- `docker swarm init`: It is used to initialize a swarm. The node on which this command is executed becomes the manager in the newly created single-node swarm.
  This command accepts various options. Use `docker swarm init --help` command for the complete list of options.
  Some common options are:
  - `--advertise-addr` to specify the advertised address in the format `ip:port`
  - `–autolock` to enable manager autolocking after which manager requires an unlock key to start the manager
  - `–availability` to specify the availability of the node such as active, pause, drain. The default value is active

```
docker swarm init <options>
```

The `docker swarm init` command generates two random tokens, one is a worker token and other is a manager token. When the new node is added to the swarm, the node joins as either a worker or manager based upon the worker or manage token passed.

- `docker swarm join`: It is used to add the node to a swarm. The node can join as manager or worker based on the token passed to the `--token` flag.

```
docker swarm join --token <token>
```

- `docker swarm leave`: It allows the node to leave the swarm. It accepts `--force` option which can be used for a manger to remove from the swarm forcefully but the safe way to remove a manager from a swarm is to demote it to a worker and then direct it to leave the swarm.

```
docker swarm leave
```

- `docker swarm unlock`: It is used to unlock a locked manager using the provided unlock key. The unlock key is printed at the time when autolock is enabled, and is also available from the `docker swarm unlock-key` command.

```
docker swarm unlock
```

- `docker swarm update`: It is used to update the swarm with new parameter values including manager autolocking setting, validity period for node certifications, dispatcher heartbeat period etc. It accepts various options including:
    - `–autolock` to change manager autolocking setting to `true` or `false`
    - `--cert-expiry` to set the validity period for node certificates. Default value is `2160h0m0s`
    - `--dispatcher-heartbeat` to set the dispatcher heartbeat period which is `5s` by default.
    - `--external-ca` to update specifications of one or more certificate signing endpoints
    - `--max-snapshots` to set the number of Raft snapshots to retain

- `--snapshot-interval` to set the number of log entries between Raft snapshots. Default value is `10000`
- `--task-history-limit` to set the task history retention limit which is `5`, by default.

```
docker swarm update <options>
```

14.3.2 DOCKER NODE COMMANDS

The `docker` node commands are used to manage nodes in swarm.

- `docker node ls`: It is used to list all nodes connected in the swarm.

```
docker node ls
```

- `docker node ps`: It is used to list tasks that are running on the current node or on the specified node.

```
docker node ps
```

or

```
docker node ps <node_name>
```

- `docker node inspect`: It is used to get the detailed information of one or mode nodes.

```
docker node inspect <node_name>
```

Use the following command to inspect the current node:

```
docker node inspect self
```

- `docker node promote`: It is used to promote a node to manager.

```
docker node promote <node_name>
```

- `docker node demote`: It is used to demote a node from the existing manager so that it is no longer a manager in a swarm.

```
docker node demote <node_name>
```

- `docker node update`: It is used to update the node with new parameter values. It accepts various options including:
  - `--availability` to set the availability of the node as either `active` or `pause` or `drain`
  - `--label-add` to add or update a node label
  - `--label-rm` to remove a node label if exist
  - `--role` to set the role of the node as either `worker` or `manager`

```
docker node update <options> <node_name>
```

- `docker node rm`: It allows to remove the specified node from the Docker swarm.

```
docker node rm <node_name>
```

14.3.3 DOCKER SERVICE COMMANDS

The `docker service` commands are used to manage the services in swarm.

- `docker service create`: It is used to create a new service as described by the specific parameter passed. Use `docker service create --help` command for the complete list of options.
  Some common options are:
  - `--name` to specify the name of service
  - `--mode` to specify the mode of service such as `replicated` (default), `global`, `replicated-job`, `global-jobmanager`
  - `--network` to attach the specific network of the service

```
docker service create <options>
```

- `docker service ls`: It is used to list all services running in the swarm.

```
docker service ls
```

- `docker service ps`: It is used to list tasks that are running under specified services in the swarm.

```
docker service ps <service_name>
```

- `docker service inspect`: It is used to get the details of a specified service.

```
docker service inspect <service_name>
```

- `docker service logs`: It is used to fetch the logs of a container. It accepts various options including:

```
docker service inspect <service_name>
```

- `docker service update`: It is used to update the service with new parameter values. The parameters are the same as `docker service create` command.

```
docker service update <service_name> <options>
```

- `docker service scale`: It is used to scale one or more replicated services.

```
docker service scale <service_name>=<number_of_replicas>
```

- `docker service rollback`: It is used to rollback changes to the service's configuration.

```
docker service rollback <service_name>
```

- `docker service rm`: It allows to remove the specified service from the Docker swarm.

```
docker service rm <service_name>
```

14.3.4 DOCKER SECRET COMMANDS

The `docker secret` commands are used to manage the swarm secrets.

- `docker secret create`: It is used to create a secret from a file or from user input. Use `docker secret create --help` command for the complete list of options.

```
docker secret create <secrete_name> <file>
```

- `docker secret ls`: It is used to list all secrets created in the swarm. This command must be executed on the manager node to get the list of all secrets.

```
docker secret ls
```

- `docker secret inspect`: It is used to get the detailed information of a specified secret.

```
docker secret inspect <secret_name>
```

- `docker secret rm`: It allows to remove the specified secret from the Docker swarm.

```
docker secret rm <secret_name>
```

14.3.5 DOCKER STACK COMMANDS

The `docker stack` commands are used to manage the Swarm stacks.

- `docker stack config`: It is used to output the final Compose file after doing merges and interpolations of the specified compose files.

```
docker stack config -c <compose_file>
```

- `docker stack deploy`: It is used to create a new stack or update existing stack in Docker Swarm as described by the specific parameters passed. Use `docker stack deploy --help` command for the complete list of options.
  Some common options are:
    - `-c` or `--compose-file` to specify the fully qualified name of the compose file
    - `-d` to exit immediately without waiting for stack services to complete

  - `--prune` to prune services that are no longer services
  - `-q` to suppress the progress output

```
docker stack deploy -c <compose_file> <options>
```

If the configuration is split between multiple Compose files such as a base configuration and environment-specific overrides, we can provide multiple `--compose-file` flags as below:

```
docker stack deploy --compose-file docker-compose.yml -c docker-
compose.prod.yml service_name
```

- `docker stack ls`: It is used to list all stacks available in the swarm. It can have addition options including:
  - `--format` to format output using a custom template

```
docker stack ls
```

- `docker stack ps`: It is used to list all tasks running as part of specified stack available in the swarm. It can have addition options including:
  - `-f` to filter output based on given conditions
  - `--format` to format output using a custom template
  - `--no-resolve` to not map IDs to Names
  - `--no-trunc` to not truncate the output
  - `-q` to displays task IDs

```
docker stack ps <stack_name>
```

- `docker stack services`: It is used to list all services running as part of specified stack available in the swarm. It can have addition options including:
  - `-f` to filter output based on given conditions
  - `--format` to format output using a custom template
  - `-q` to displays task IDs

```
docker stack service <stack_name>
```

- `docker stack rm`: It allows to remove the specified stack and its associated components including services, networks and secrets from the Docker swarm. It can have additional options including:
    - `-d` to not wait for stack removal

```
docker stack rm <stack_name>
```

14.4 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).*

14.4.1 CONFIGURE NGINX SERVER

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)
```

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

. : File C:\Users\hp\Documents\WindowsPowerShell\profile.ps1 cannot be loaded because running scripts is disabled on
this system. For more information, see about_Execution_Policies at https:/go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:3
+ . 'C:\Users\hp\Documents\WindowsPowerShell\profile.ps1'
+   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
PS C:\Users\hp> docker ps
CONTAINER ID   IMAGE                       COMMAND                  CREATED       STATUS                  PORTS
          NAMES
cc2c9918ee8d   nginx:alpine                "/docker-entrypoint.…"   3 hours ago   Up 3 hours              0.0.0.0:8080-
>80/tcp   nginx
1bc83a45cd93   python-flask-mysql-demo-app   "python flask_app.py"    3 hours ago   Up 3 hours
          flask
7c1a05cf11c1   mysql:latest                "docker-entrypoint.s…"   3 hours ago   Up 3 hours (healthy)   3306/tcp, 330
60/tcp    mysql
PS C:\Users\hp> docker stop $(docker ps -q)
cc2c9918ee8d
1bc83a45cd93
7c1a05cf11c1
PS C:\Users\hp> docker ps -a
CONTAINER ID   IMAGE                       COMMAND                  CREATED       STATUS                      PORTS
 NAMES
cc2c9918ee8d   nginx:alpine                "/docker-entrypoint.…"   3 hours ago   Exited (0) 8 seconds ago
 nginx
1bc83a45cd93   python-flask-mysql-demo-app   "python flask_app.py"    3 hours ago   Exited (0) 8 seconds ago
 flask
7c1a05cf11c1   mysql:latest                "docker-entrypoint.s…"   3 hours ago   Exited (0) 3 seconds ago
 mysql
PS C:\Users\hp> docker rm $(docker ps -a -q)
cc2c9918ee8d
1bc83a45cd93
7c1a05cf11c1
PS C:\Users\hp>
```

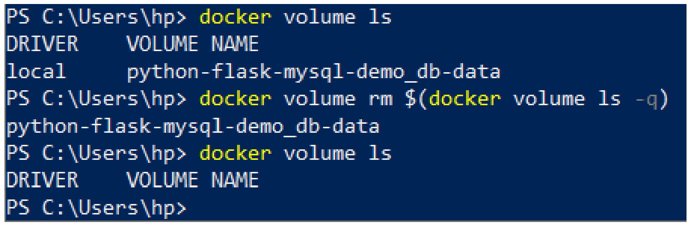
Delete all images:

```
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>
```

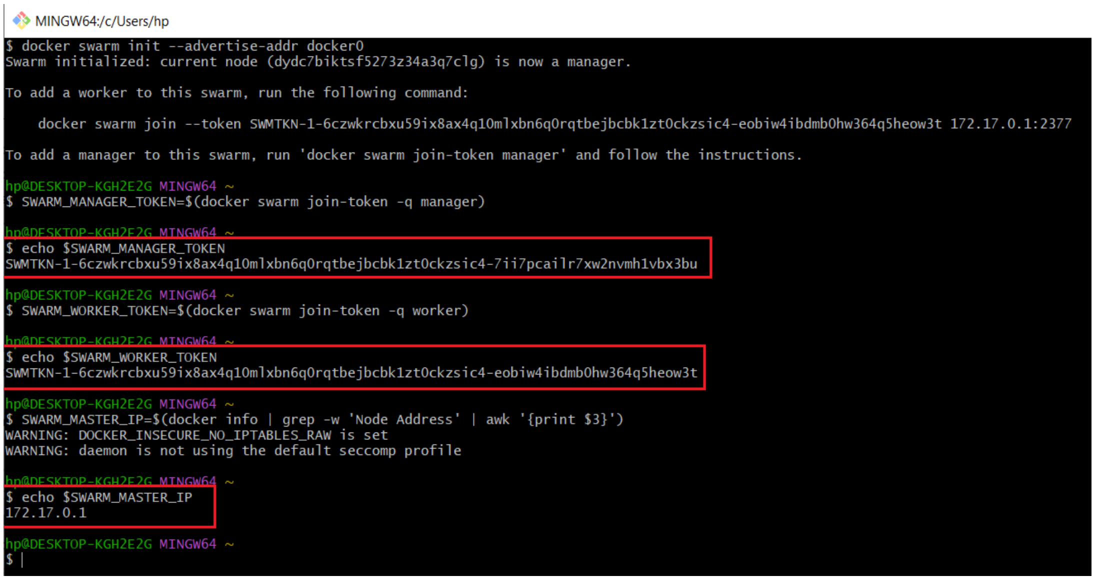
Going forward, we may need to execute few Linux commands related to Docker Swarm for which we need to use **GitBash** command prompt.

14.4.2 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:

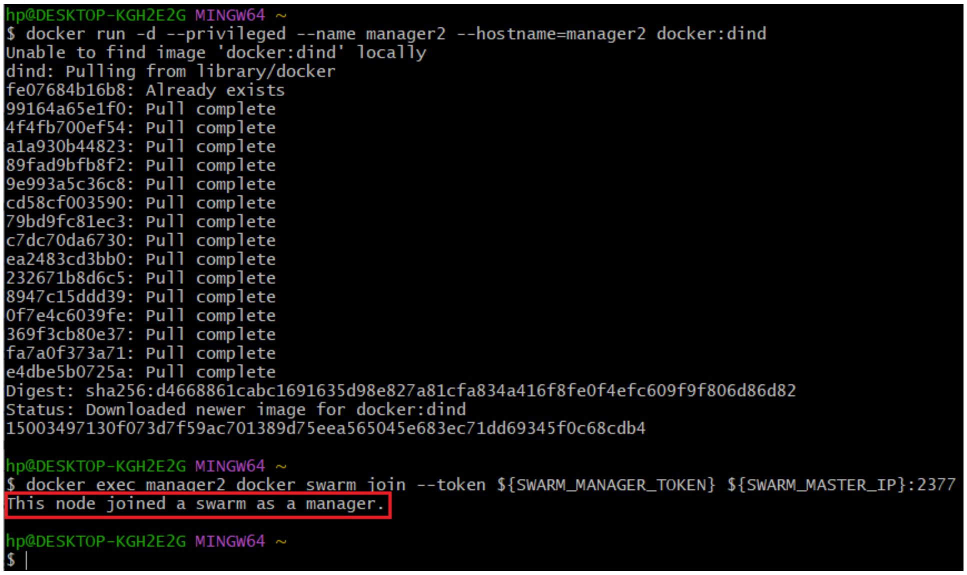
```
docker swarm init --advertise-addr 127.0.0.1
```

14.4.3 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

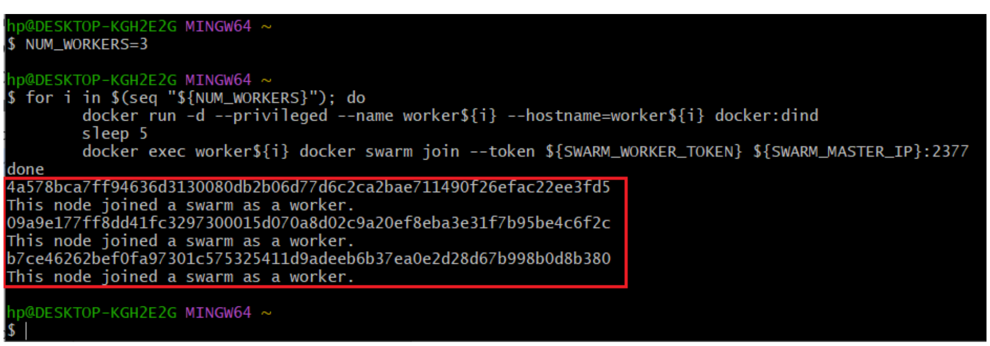


Once the node has joined the Swarm successfully, it displays message **"This node joined a swarm as a manager".**

14.4.4 ADD WORKER NODES

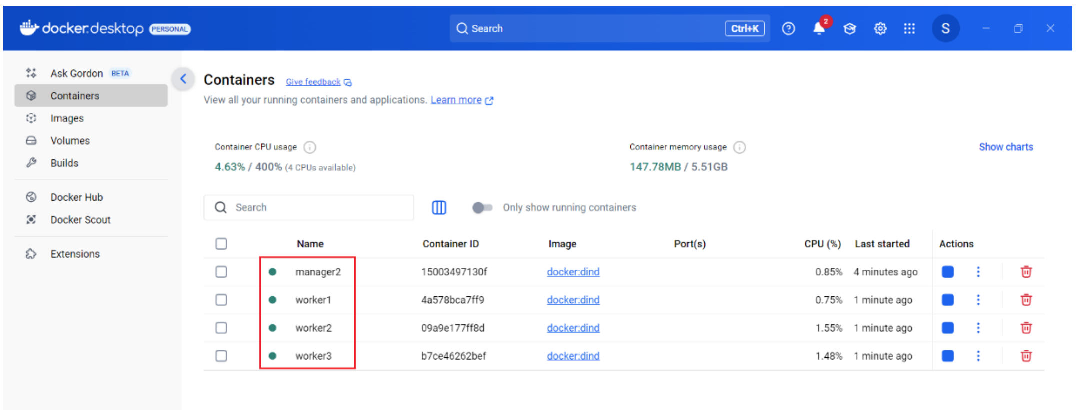
Run the following commands to launch 3 Docker containers and add those as worker nodes to the Docker Swarm:

```
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.

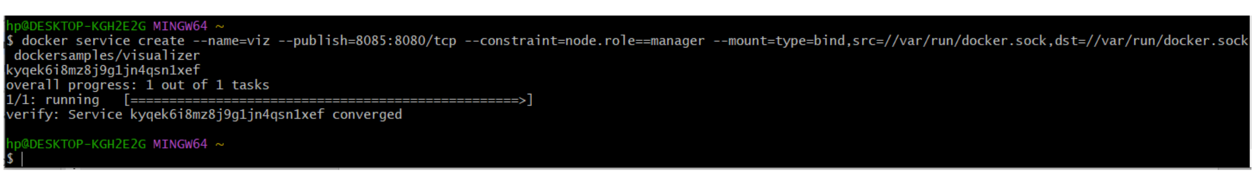


14.4.5 START VISUALIZER SERVICE

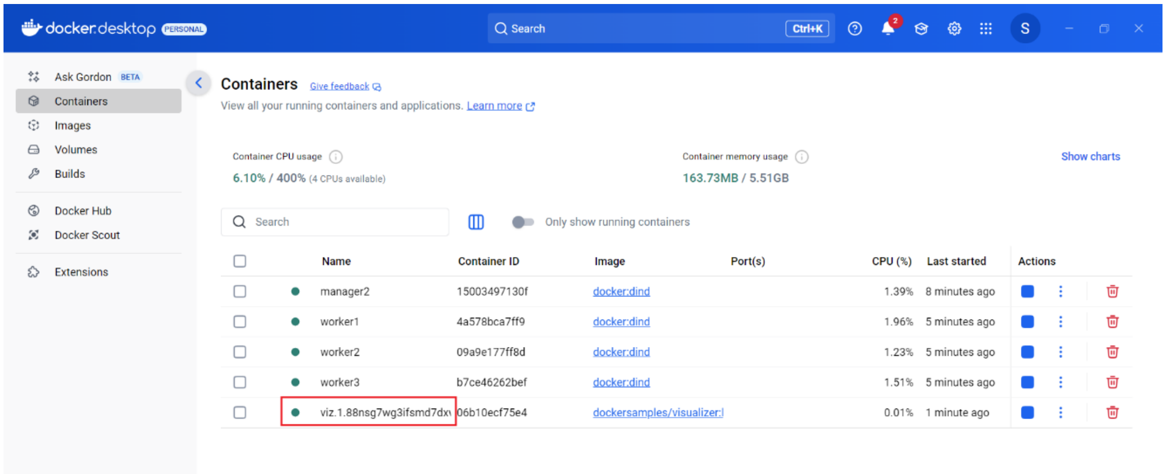
Docker provides **visualizer** image to monitor the state of Docker Swarm cluster. This visualizer image is a sample app created by Docker Samples for learning Docker Swarm and is available in Docker Hub.

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
```



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.

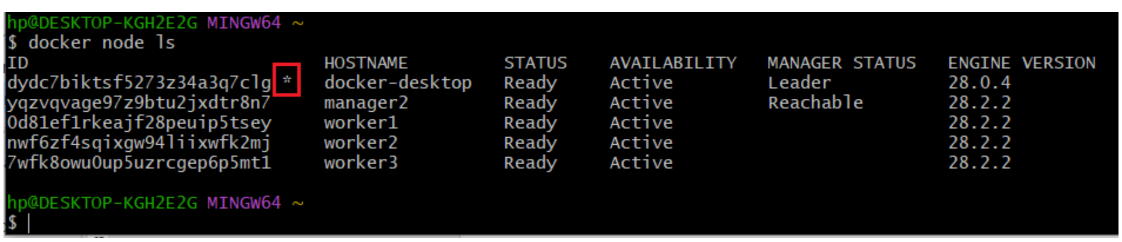


14.4.6 VERIFY DOCKER SWARM

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
```



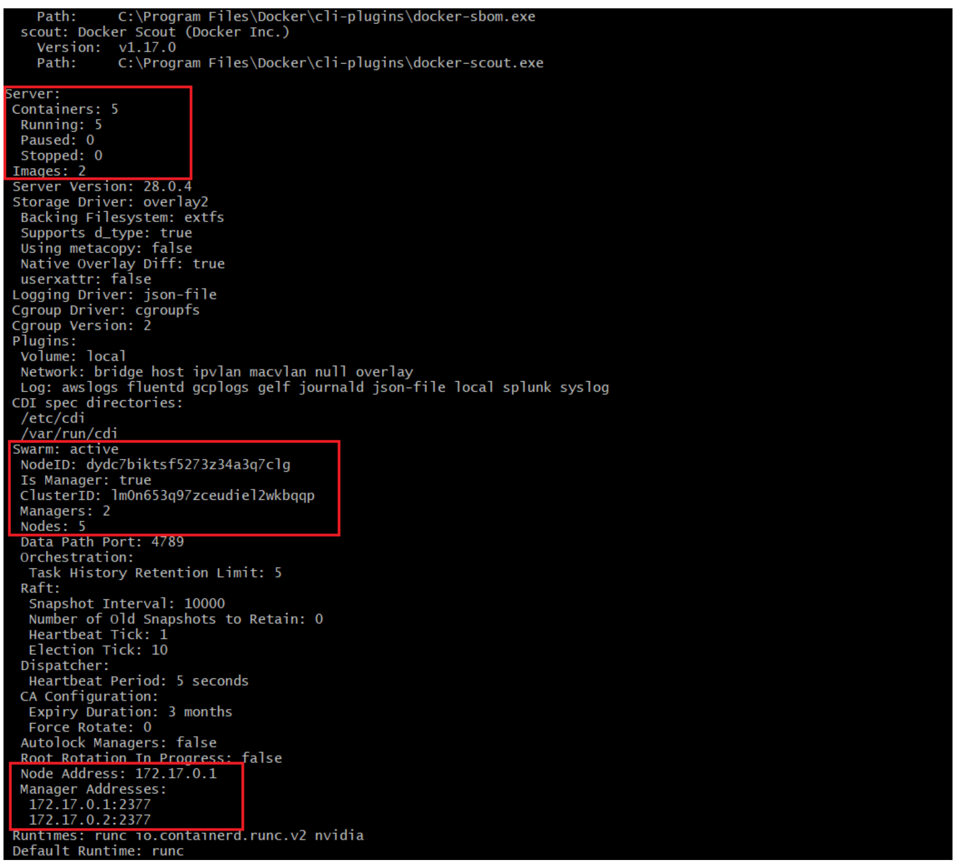
As we can see above, it displays the information of all Docker Swarm nodes:

- 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:

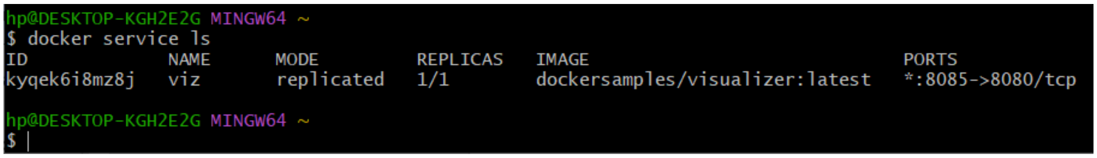
```
docker info
```



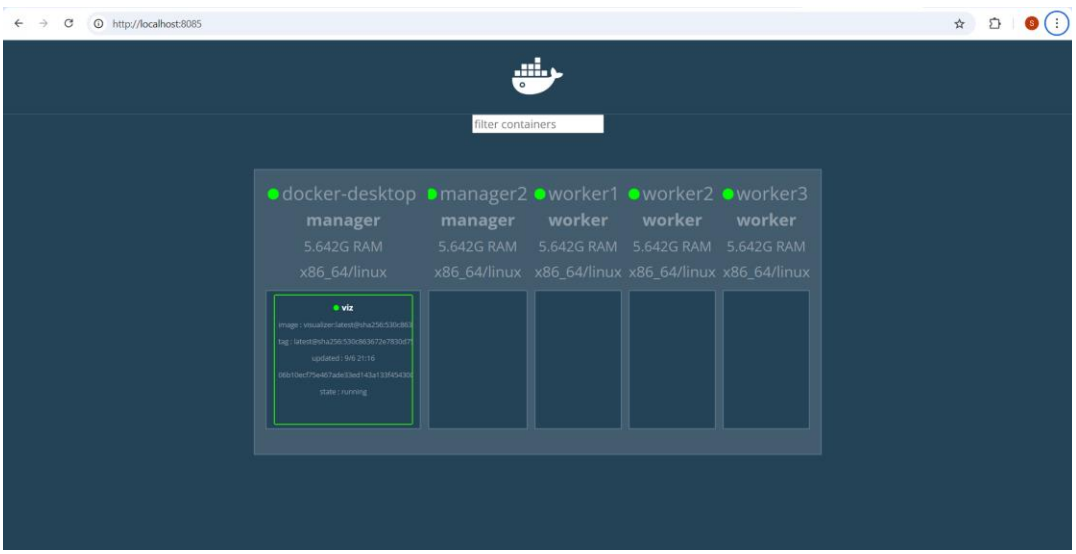
As we can see above, it displays the current node information along with the **Swarm** status as **Active** with **5 Nodes** of which **2** are **Manager** nodes.

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.



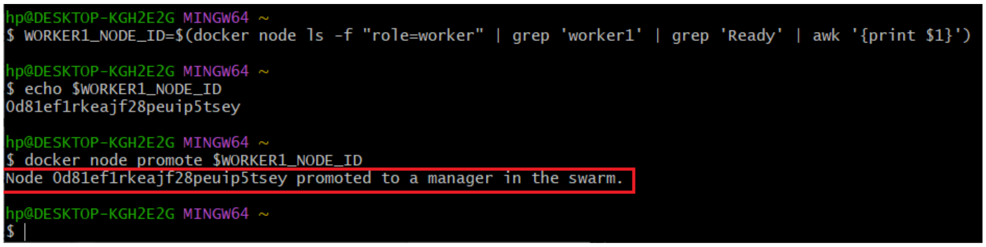
14.4.7 PROMOTE OR DEMOTE NODE

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

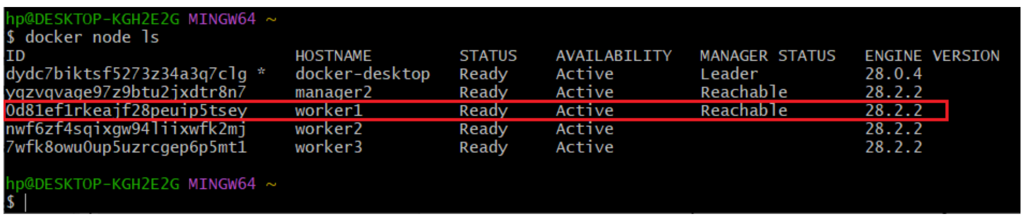
```
WORKER1_NODE_ID=$(docker node ls -f "role=worker" | grep 'worker1' |
grep 'Ready' | awk '{print $1}')

echo $WORKER1_NODE_ID
```

```
docker node promote $WORKER1_NODE_ID
```



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

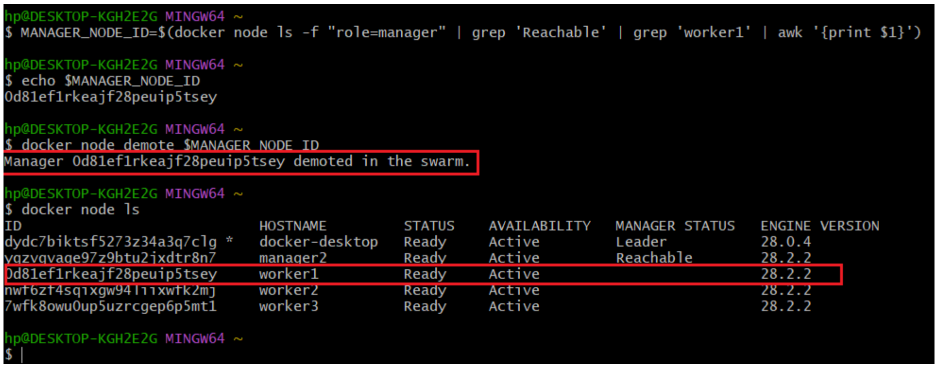
```
docker node ls
```



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
```



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

14.4.8 CONFIGURE NGINX SERVER

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.

<u>Note:</u>

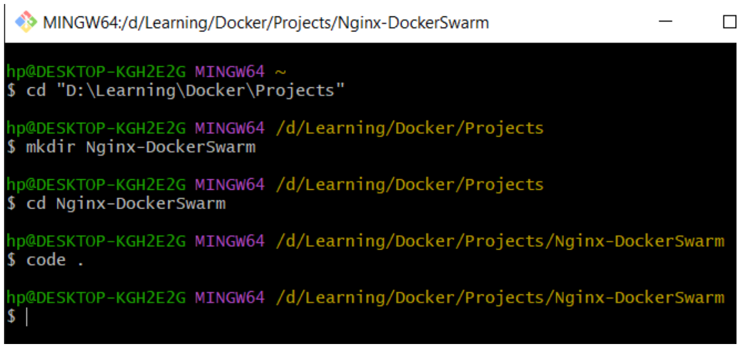
Docker Compose, as a single-node multi-container orchestrator, is able to build images for its services but Docker Swarm, as a multi-node multi-container orchestrator, cannot build images since the image would be potentially required on each node of the cluster. In Docker Swarm, we should either use pre-existing images from Docker Hub, or create images locally and push them to Docker Hub *(or a private registry)* and then use that image in the Swarm Stack Docker Compose File. Another manual approach without using Docker Hub or private registry is to build the image on one of the Swarm nodes, save the image to a `.tar` file and load it into the other nodes and then use it in the Swarm Stack Docker Compose File.

14.4.8.1 Launch VS Code

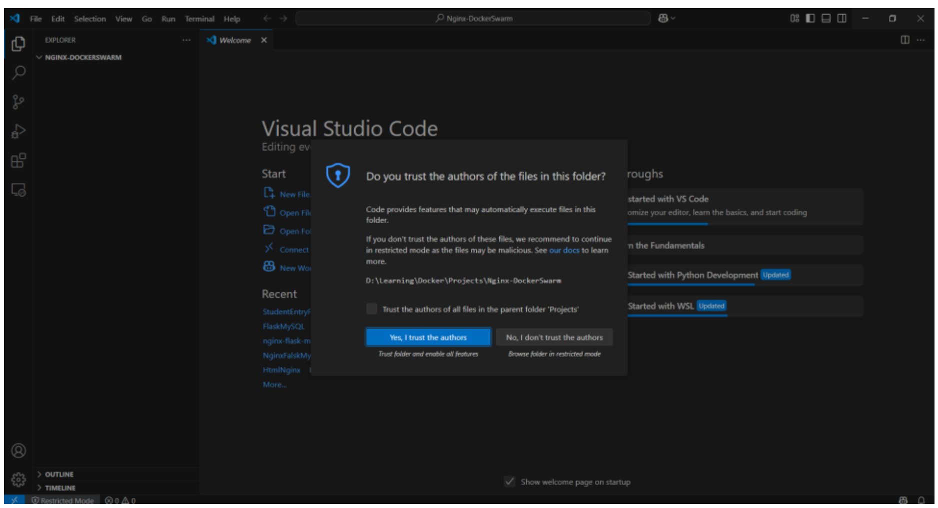
Open **Git Bash** command prompt, run the following commands to create a new directory at `D:\Learning\Docker\Projects` location and open **VS Code**. You can choose any location to create a new directory.

```
cd "D:\Learning\Docker\Projects"
```

```
mkdir Nginx-DockerSwarm
cd Nginx-DockerSwarm
code .
```

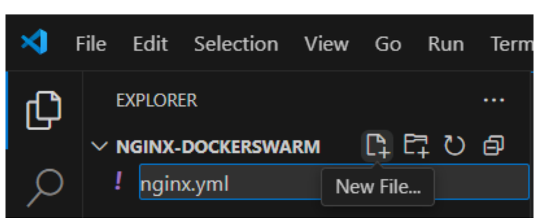


It opens **VS Code** application which might prompt you to confirm if you trust the authors of file in which case, click on **Yes, I trust the authors** button.



14.4.8.2 *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:
```

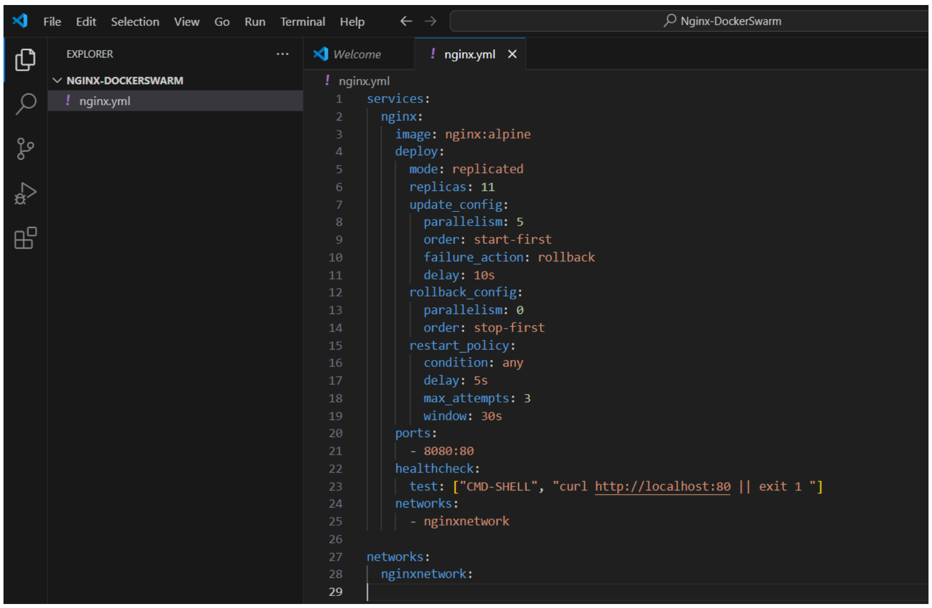
The above code performs the following:

- 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.

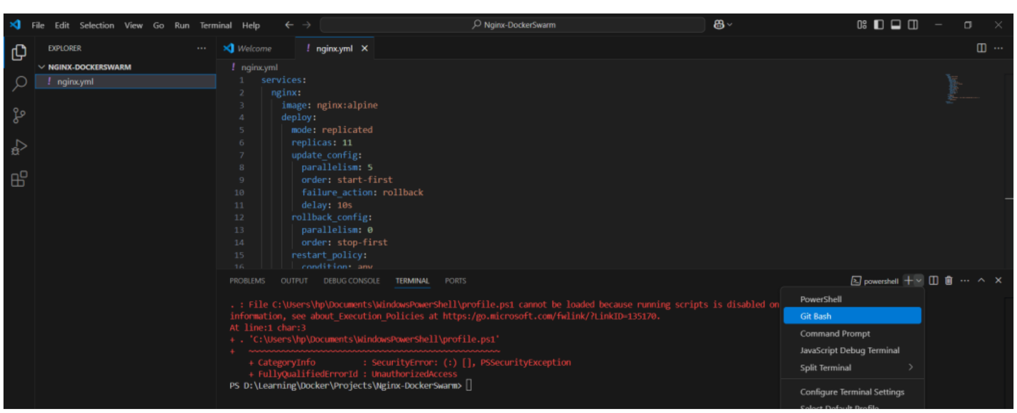
- Maps local host port `8080` to container port `80`.
- Performs healthcheck to verify web URL of `nginx` server and exit with `1` if it returns any failure.
- Maps the container to `ngnixnetwork` network.

- Create `nginxnetwork` network



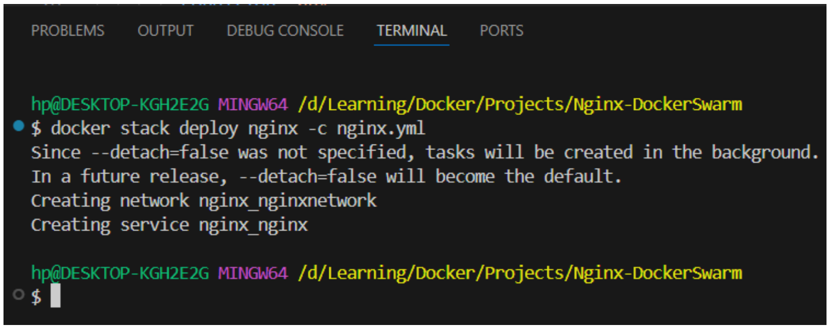
14.4.8.3 *Deploy Stack*

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:

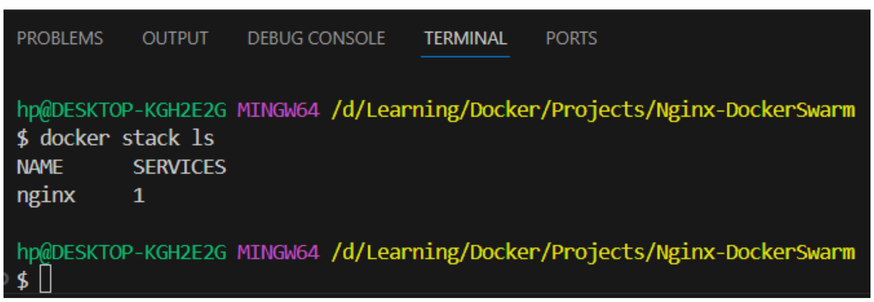
```
docker stack deploy nginx -c nginx.yml
```



14.4.8.4 Verify Stack and Services

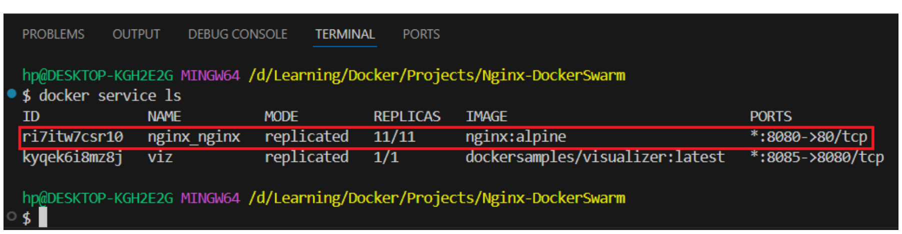
Verify if the stack has been created using the below command:

```
docker stack ls
```



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

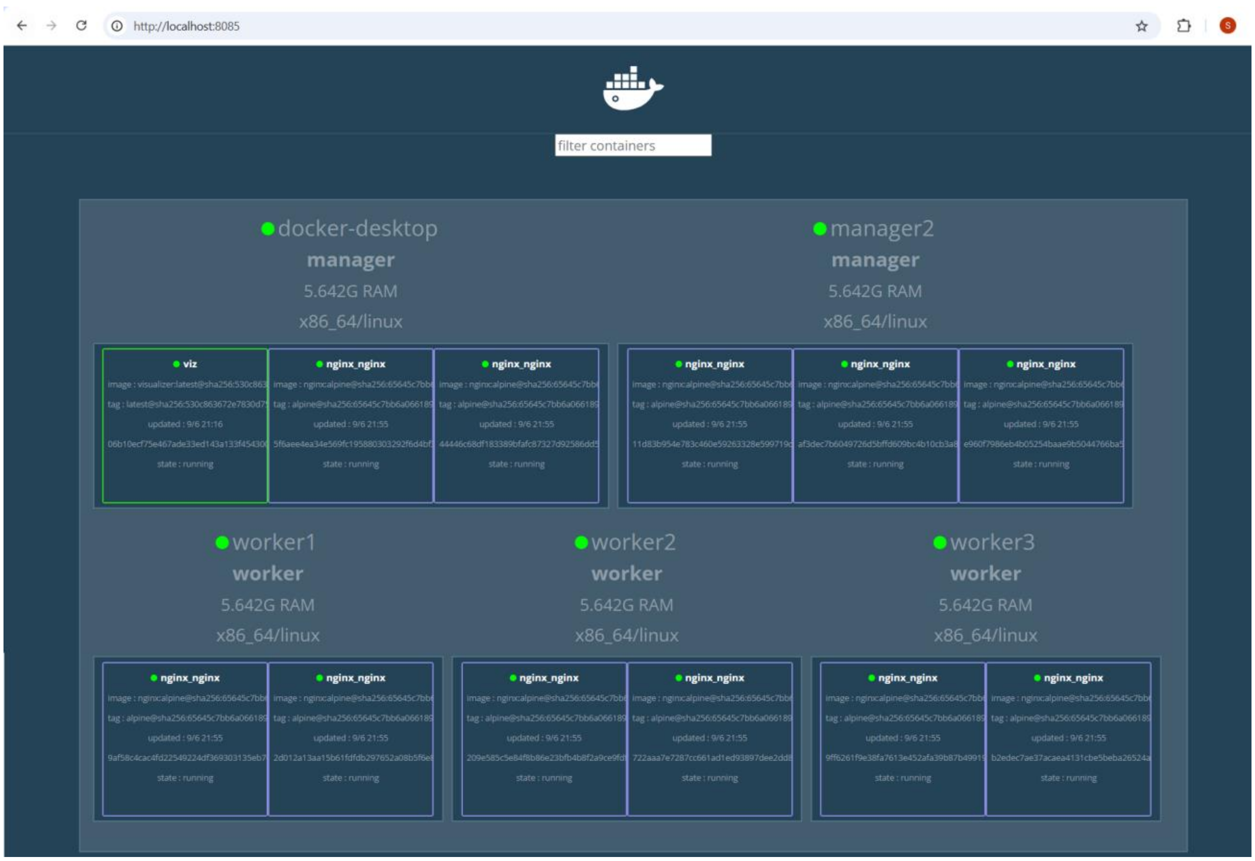
```
docker service ls
```



It displays `nginx` service created with 11 replicas.

14.4.8.5 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)
```

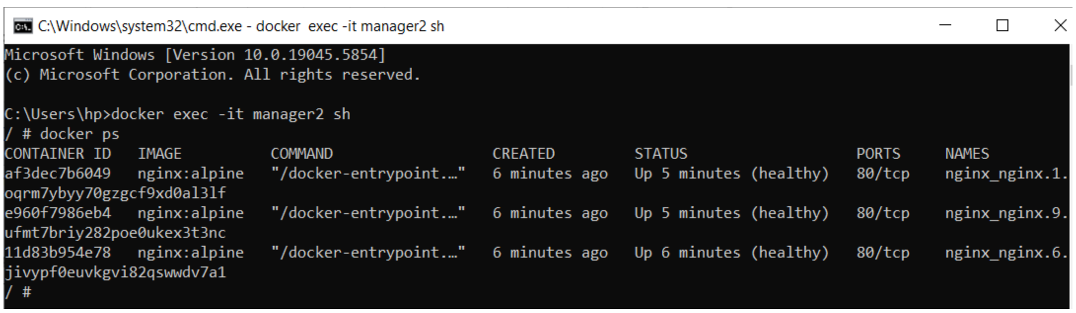
```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

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
$
```

14.4.8.6 Validate Task Containers

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

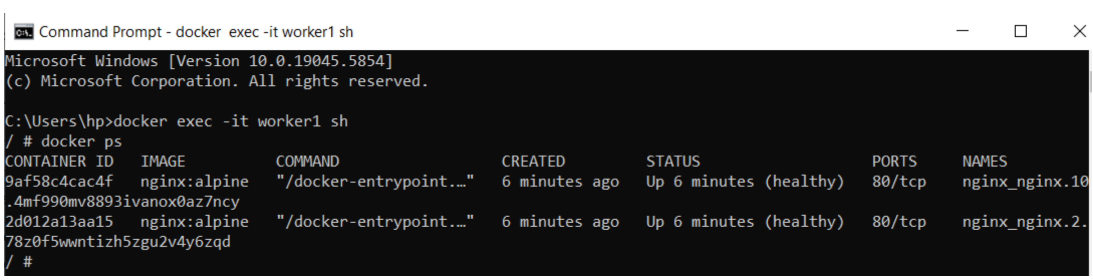
Open new **Command Prompt** or **Windows Powershell** and run the below commands to connect to `manager2` instance and verify the containers running:

```
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
```



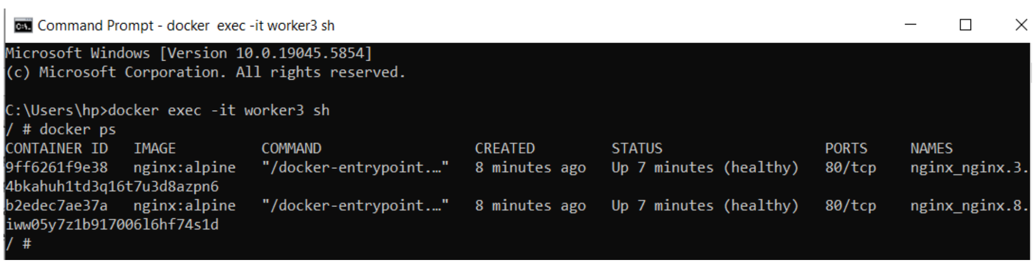
Open new **Command Prompt** and run the below commands to connect to `worker2` instance and verify the containers running:

```
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
```



14.4.9 TEST RESILIENCE

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

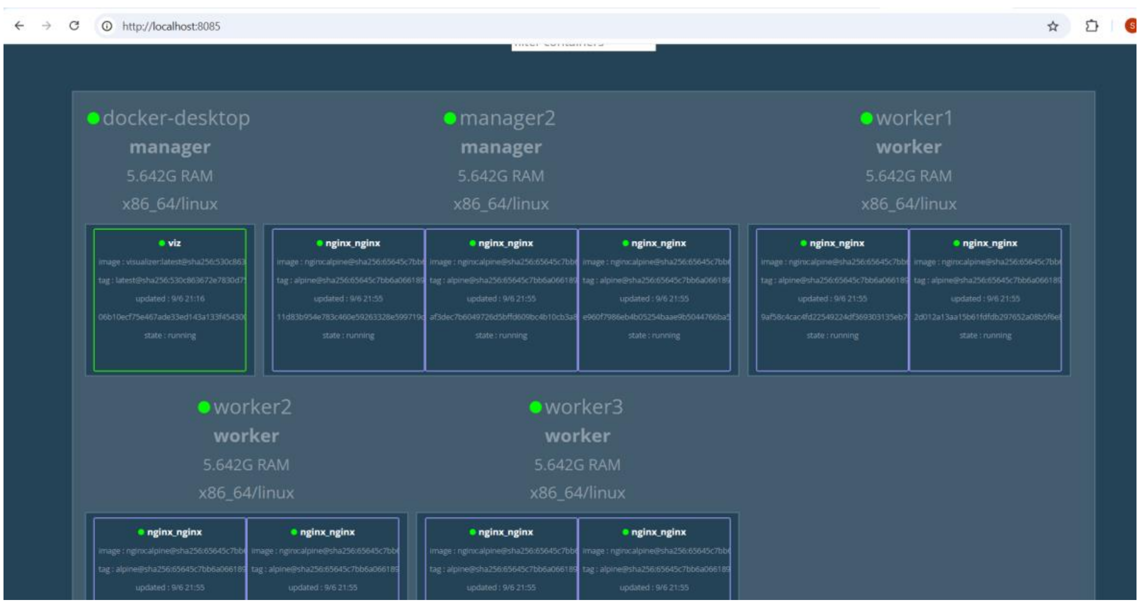
14.4.9.1 Delete Containers

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.

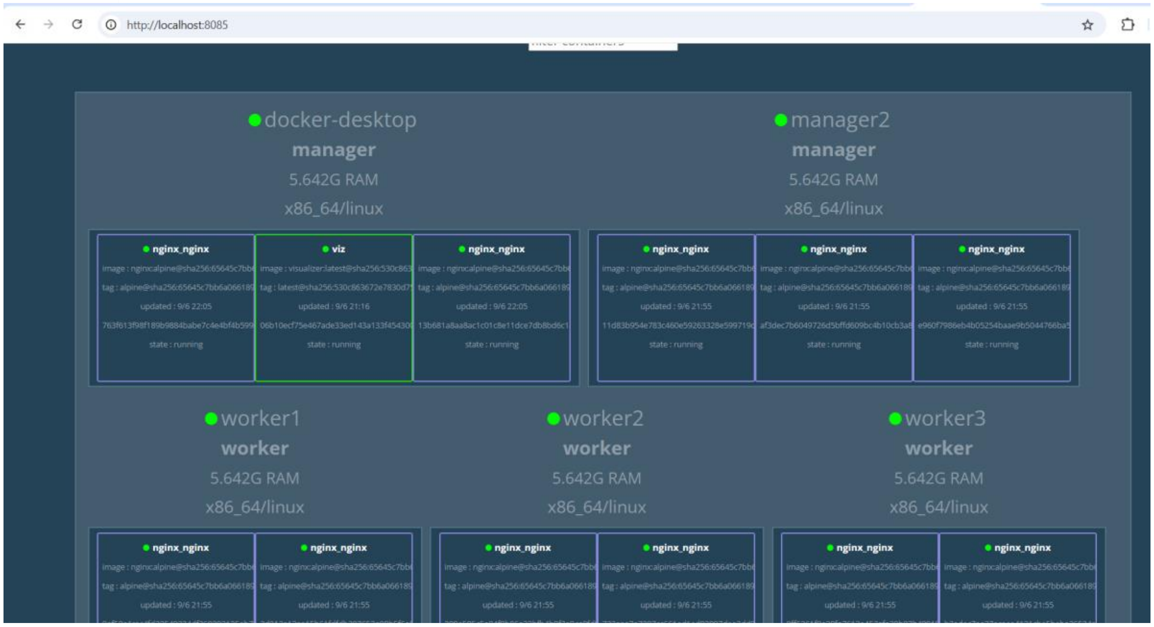
```
docker ps
docker rm -f $(docker ps --filter name=nginx -q)
docker ps
```

```
hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker ps
CONTAINER ID   IMAGE                              COMMAND                  CREATED          STATUS                    PORTS            NAMES
44446c68df18   nginx:alpine                       "/docker-entrypoint.…"   10 minutes ago   Up 10 minutes (healthy)   80/tcp           nginx_nginx.7.tz6nkxtocqvvh
d38ed59kb1r6
5f6aee4ea34e   nginx:alpine                       "/docker-entrypoint.…"   10 minutes ago   Up 10 minutes (healthy)   80/tcp           nginx_nginx.4.tntatll5ebzjg
ccrtl24lq5kd
06b10ecf75e4   dockersamples/visualizer:latest   "/sbin/tini -- node …"   48 minutes ago   Up 48 minutes (healthy)   8080/tcp         viz.1.88nsg7wg3ifsmd7dxw554
6k10
b7ce46262bef   docker:dind                        "dockerd-entrypoint.…"   52 minutes ago   Up 52 minutes             2375-2376/tcp    worker3
09a9e177ff8d   docker:dind                        "dockerd-entrypoint.…"   52 minutes ago   Up 52 minutes             2375-2376/tcp    worker2
4a578bca7ff9   docker:dind                        "dockerd-entrypoint.…"   52 minutes ago   Up 52 minutes             2375-2376/tcp    worker1
15003497130f   docker:dind                        "dockerd-entrypoint.…"   54 minutes ago   Up 54 minutes             2375-2376/tcp    manager2

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker rm -f $(docker ps --filter name=nginx -q)
44446c68df18
5f6aee4ea34e

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker ps
CONTAINER ID   IMAGE                              COMMAND                  CREATED              STATUS                    PORTS            NAMES
763f613f98f1   nginx:alpine                       "/docker-entrypoint.…"   About a minute ago   Up 57 seconds (healthy)   80/tcp           nginx_nginx.4.815x7h4up
72o8o5fwiqxm9e6p
13b681a8aa8a   nginx:alpine                       "/docker-entrypoint.…"   About a minute ago   Up 57 seconds (healthy)   80/tcp           nginx_nginx.7.hjhioej2g
nieu2uh9xewdouu9
06b10ecf75e4   dockersamples/visualizer:latest   "/sbin/tini -- node …"   50 minutes ago       Up 50 minutes (healthy)   8080/tcp         viz.1.88nsg7wg3ifsmd7dx
w5546k10
b7ce46262bef   docker:dind                        "dockerd-entrypoint.…"   53 minutes ago       Up 53 minutes             2375-2376/tcp    worker3
09a9e177ff8d   docker:dind                        "dockerd-entrypoint.…"   54 minutes ago       Up 54 minutes             2375-2376/tcp    worker2
4a578bca7ff9   docker:dind                        "dockerd-entrypoint.…"   54 minutes ago       Up 54 minutes             2375-2376/tcp    worker1
15003497130f   docker:dind                        "dockerd-entrypoint.…"   56 minutes ago       Up 56 minutes             2375-2376/tcp    manager2

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

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



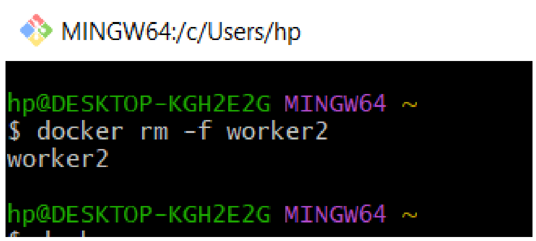
Immediately, 2 new containers have been started on the leader node as soon as the old containers were shutdown.



14.4.9.2 Crash Node

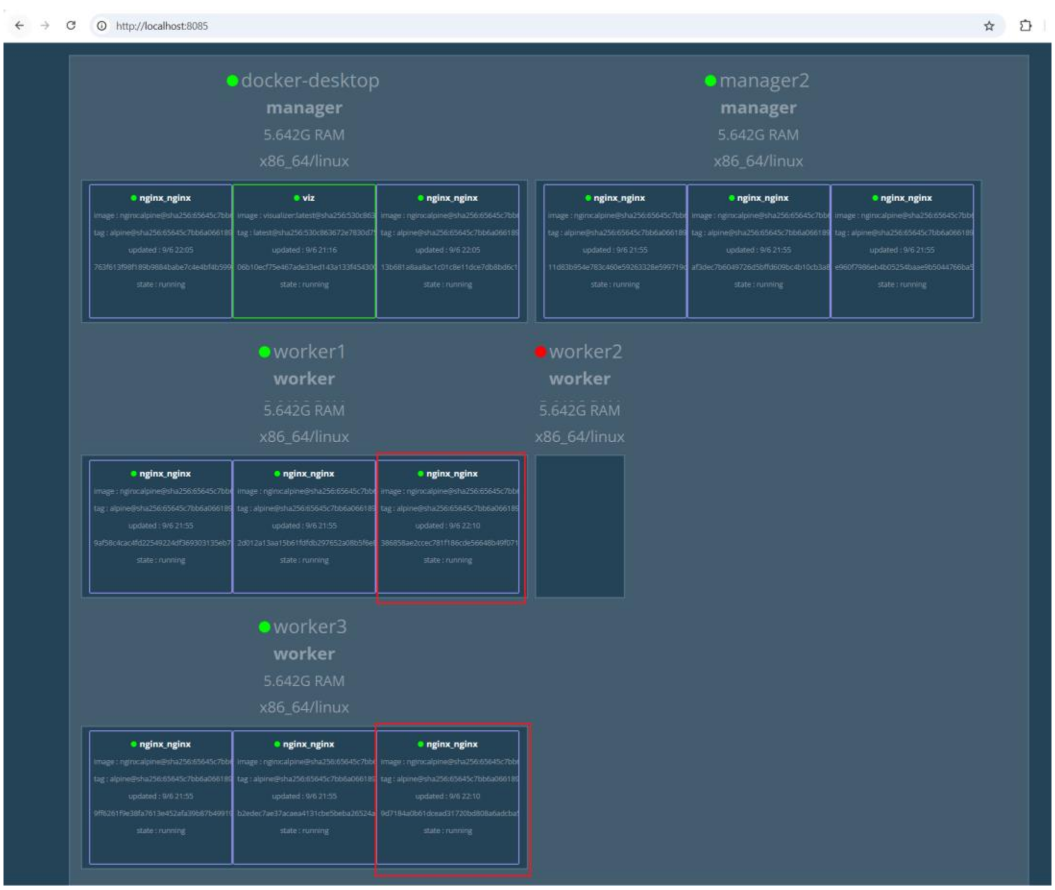
Now, let us make it even worse of crashing a worker node. In the **GitBash** prompt, run the following container to remove `worker2` container that is acting as a Swarm node.

```
docker rm -f worker2
```



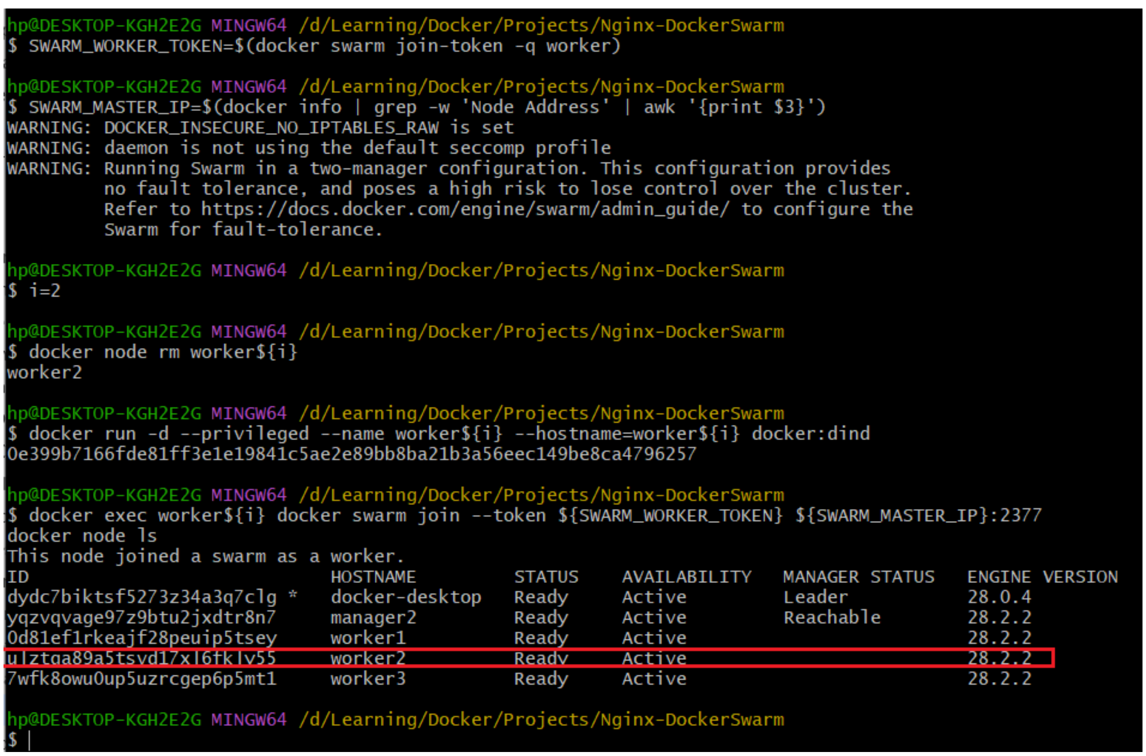
The Visualizer also reports that `worker2` node is down *(status changed to Red)* and containers on `worker2` are removed.

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

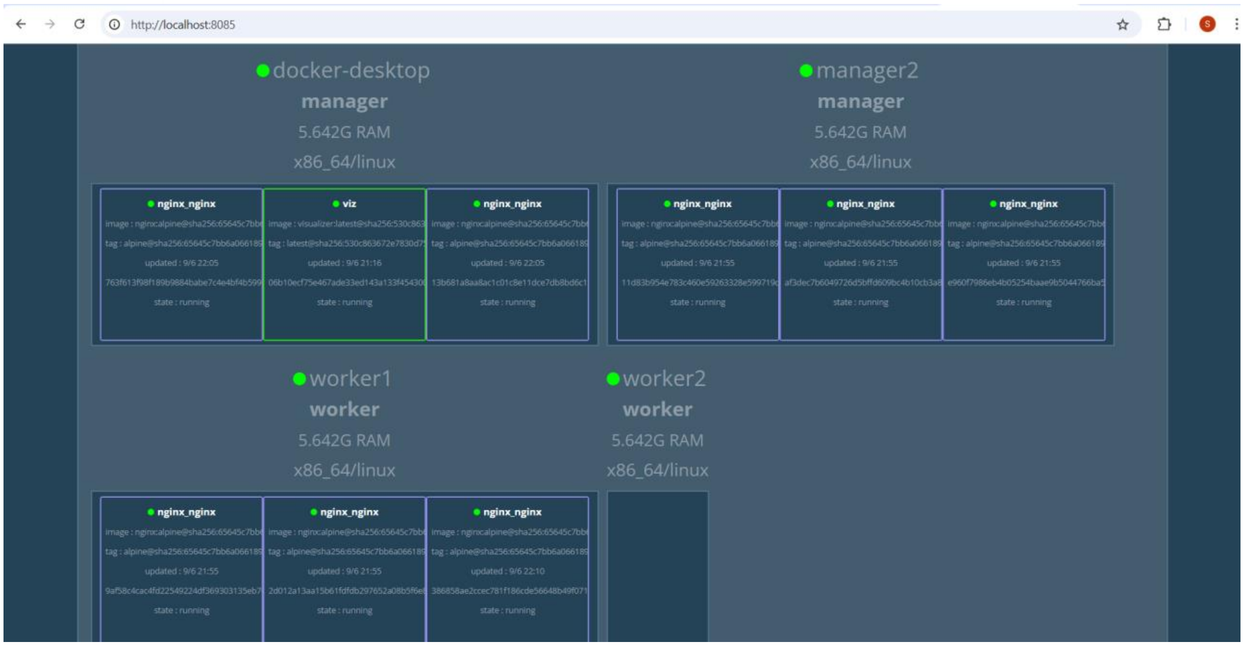


Let us restart the failed node by executing the following commands on the **GitBash**:

```
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
```



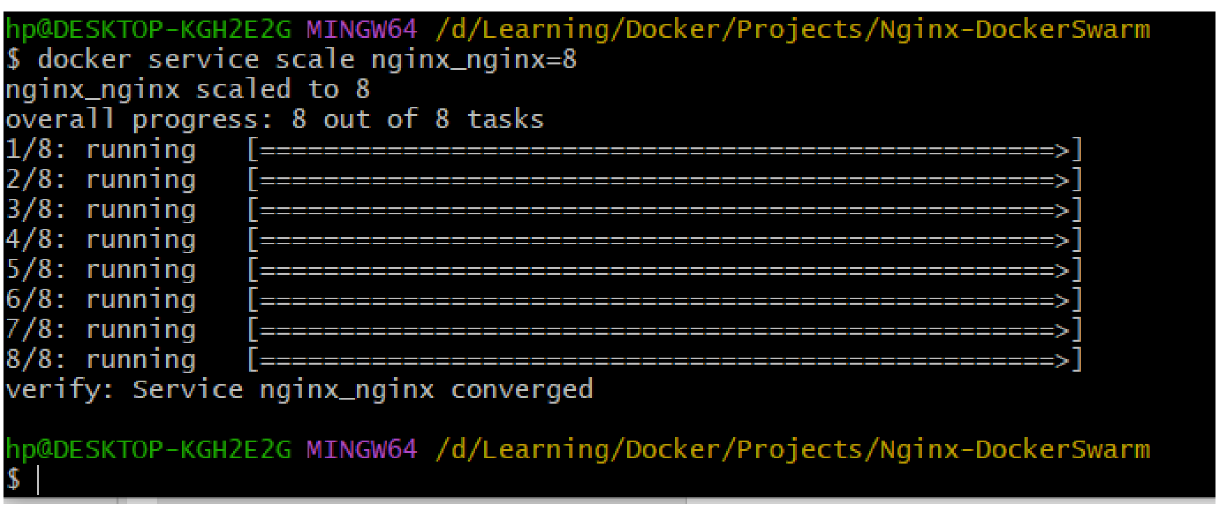
As we see above, `worker2` node is added back to Swarm. Refresh the Visualizer application which displays `worker2` node is back online. Though the node is online, the containers are not rebalanced to the new worker node automatically.



Now, we need to rebalance containers manually.

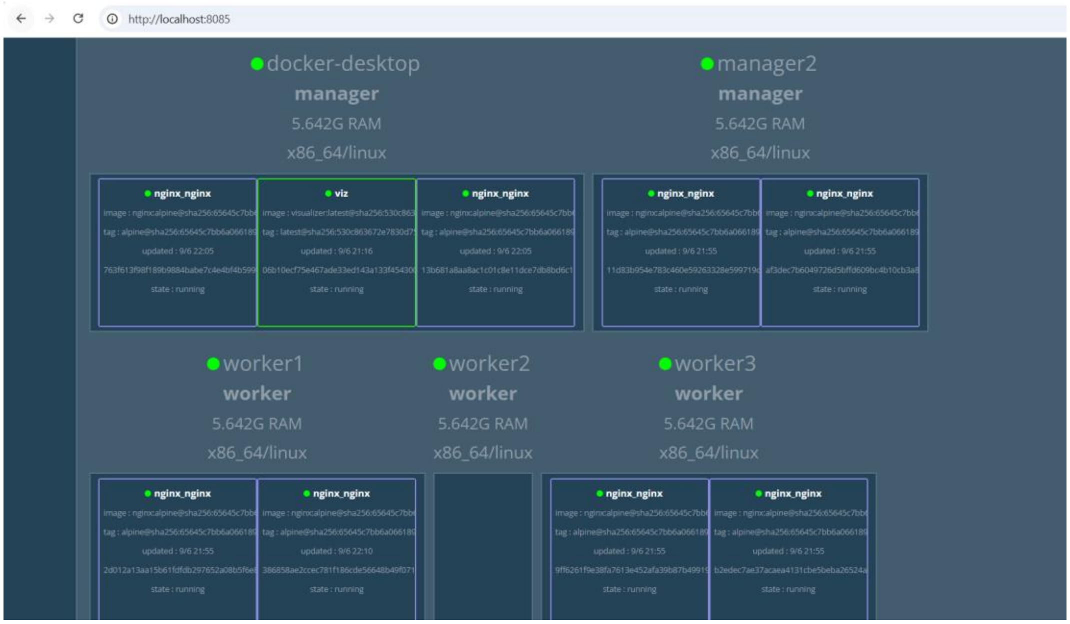
First, run the following command to scale down the containers:

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



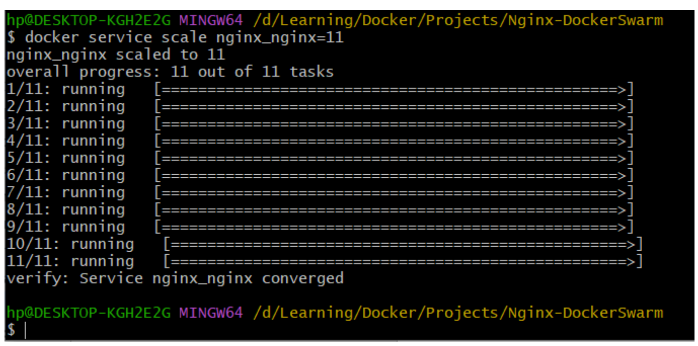
```
hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker service scale nginx_nginx=8
nginx_nginx scaled to 8
overall progress: 8 out of 8 tasks
1/8: running   [==================================================>]
2/8: running   [==================================================>]
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 as below:

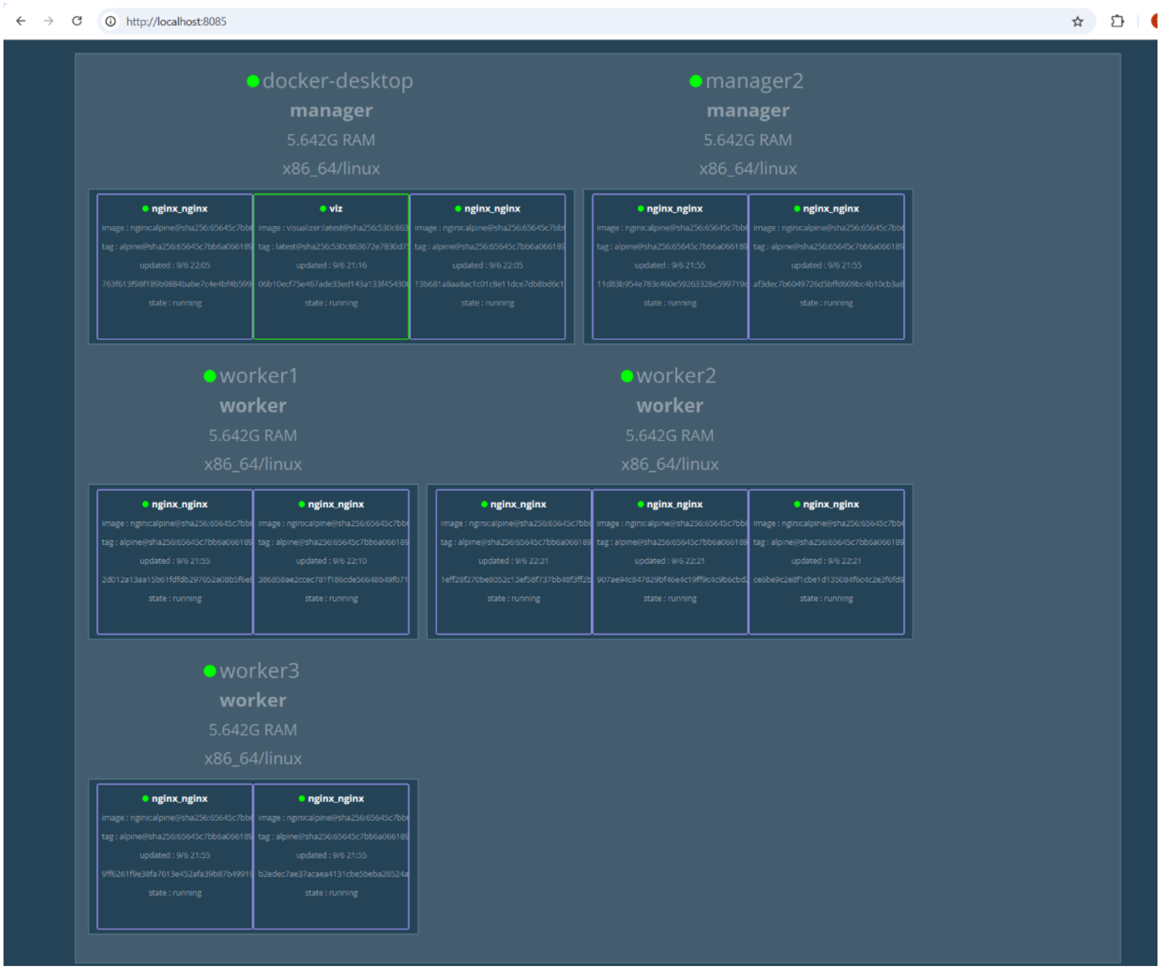


Next, run the following command to scale up the containers:

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



Now, the containers got scheduled to the `worker2` node as seen in Visualizer.

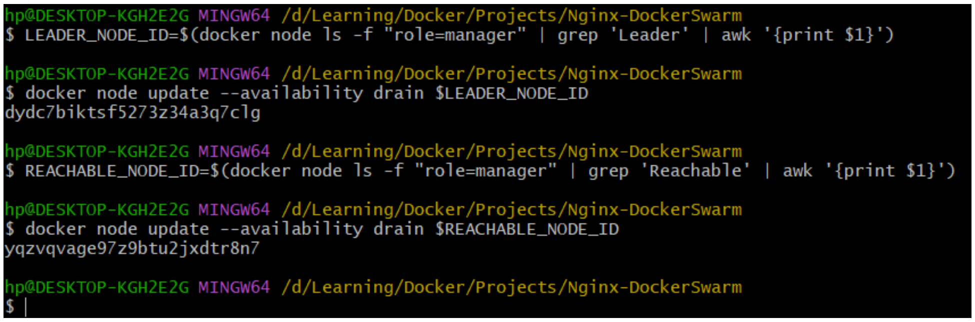


14.4.10 DRAIN MANAGER NODES

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:

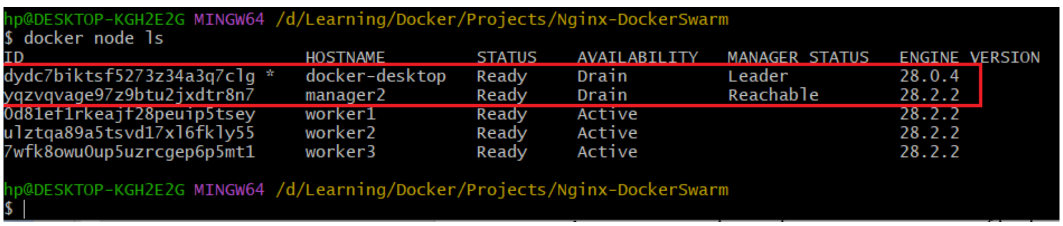
```
LEADER_NODE_ID=$(docker node ls -f "role=manager" | grep 'Leader' |
awk '{print $1}')
docker node update --availability drain $LEADER_NODE_ID
REACHABLE_NODE_ID=$(docker node ls -f "role=manager" | grep
'Reachable' | awk '{print $1}')
docker node update --availability drain $REACHABLE_NODE_ID
```



```
hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ LEADER_NODE_ID=$(docker node ls -f "role=manager" | grep 'Leader' | awk '{print $1}')

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker node update --availability drain $LEADER_NODE_ID
dydc7biktsf5273z34a3q7clg

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ REACHABLE_NODE_ID=$(docker node ls -f "role=manager" | grep 'Reachable' | awk '{print $1}')

hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker node update --availability drain $REACHABLE_NODE_ID
yqzvqvage97z9btu2jxdtr8n7

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

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

```
docker node ls
```



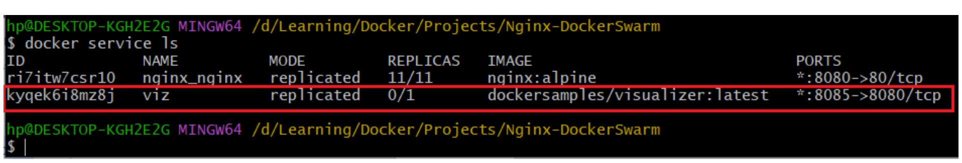
```
hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker node ls
ID                            HOSTNAME         STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
dydc7biktsf5273z34a3q7clg *   docker-desktop   Ready     Drain          Leader           28.0.4
yqzvqvage97z9btu2jxdtr8n7     manager2         Ready     Drain          Reachable        28.2.2
0d81ef1rkeajf28peuip5tsey     worker1          Ready     Active                          28.2.2
ulztqa89a5tsvd17x16fkly55     worker2          Ready     Active                          28.2.2
7wfk8owu0up5uzrcgep6p5mt1     worker3          Ready     Active                          28.2.2

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

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
```

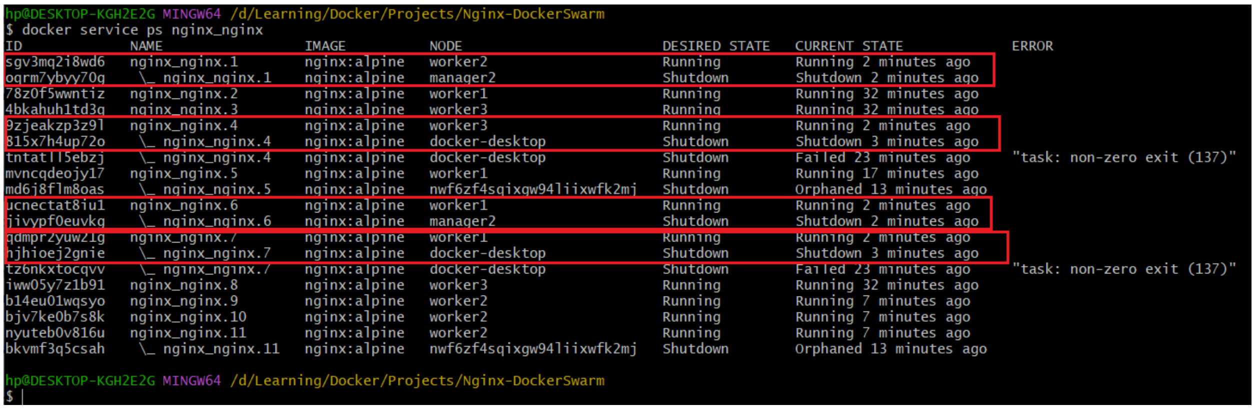


```
hp@DESKTOP-KGH2E2G MINGW64 /d/Learning/Docker/Projects/Nginx-DockerSwarm
$ docker service ls
ID             NAME          MODE         REPLICAS   IMAGE                             PORTS
ri7itw7csr10   nginx_nginx   replicated   11/11      nginx:alpine                      *:8080->80/tcp
kyqek6i8mz8j   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
oqrm7ybyy70g    \_ 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   nwf6zf4sqixgw94liixwfk2mj   Shutdown        Orphaned 13 minutes ago
ucnectat8iu1   nginx_nginx.6       nginx:alpine   worker1                     Running         Running 2 minutes ago
jivypf0euvkg    \_ nginx_nginx.6   nginx:alpine   manager2                    Shutdown        Shutdown 2 minutes ago
qdmpr2yuw21g   nginx_nginx.7       nginx:alpine   worker1                     Running         Running 2 minutes ago
hjhioej2gnie    \_ 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   nwf6zf4sqixgw94liixwfk2mj   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
```

With this, we have successfully configured Docker Swarm and validated the cluster functionality.