	Bansilal Ramnath Agarwal Charitable Trust's Vishwakarma Institute of Information Technology **Department of** **Artificial Intelligence and Data Science**

Name: Siddhesh Dilip Khairnar		
Class: TY	Division: B	Roll No: 372028
Semester: 5th	Academic Year: 2023-2024	
Subject Name & Code: Cloud Computing and Analytics (ADUA31203)		
Title of Assignment: Deploy a web application using Docker.		

Assignment 7

Title: Deploy a static website using Docker.

Theory:

1) **What is Docker?**

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code, you can significantly reduce the delay between writing code and running it in production.

The Docker platform

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security lets you run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you don't need to rely on what's installed on the host. You can share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Docker provides tooling and a platform to manage the lifecycle of your containers:

Develop your application and its supporting components using containers.

The container becomes the unit for distributing and testing your application.

When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

What can I use Docker for?
Fast, consistent delivery of your applications
Docker streamlines the development lifecycle by allowing developers to work in standardized environments using local containers which provide your applications and services. Containers are great for continuous integration and continuous delivery (CI/CD) workflows.

Consider the following example scenario:

Your developers write code locally and share their work with their colleagues using Docker containers.
They use Docker to push their applications into a test environment and run automated and manual tests.
When developers find bugs, they can fix them in the development environment and redeploy them to the test environment for testing and validation.
When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.

Responsive deployment and scaling

Docker's container-based platform allows for highly portable workloads. Docker containers can run on a developer's local laptop, on physical or virtual machines in a data center, on cloud providers, or in a mixture of environments.

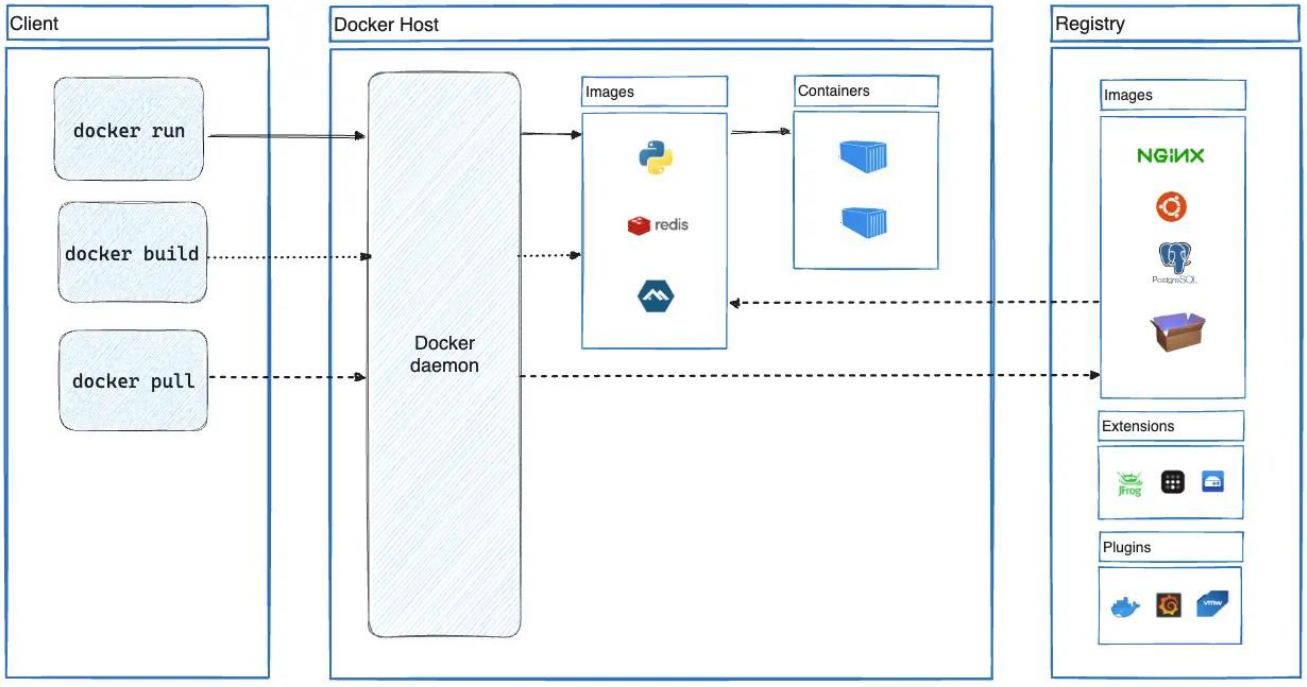
Docker's portability and lightweight nature also make it easy to dynamically manage workloads, scaling up or tearing down applications and services as business needs dictate, in near real time.

Running more workloads on the same hardware
Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines, so you can use more of your server capacity to achieve your business goals. Docker is perfect for high density environments and for small and medium deployments where you need to do more with fewer resources.

2) Docker Architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

Docker Architecture diagram



The Docker daemon
The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client
The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker Desktop
Docker Desktop is an easy-to-install application for your Mac, Windows or Linux environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper. For more information, see Docker Desktop.

Docker registries
A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker looks for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, Docker pulls the required images from your configured registry. When you use the docker push command, Docker pushes your image to your configured registry.

Docker objects
When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

Images
An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

Containers
A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that aren't stored in persistent storage disappear.

Example docker run command
The following command runs an ubuntu container, attaches interactively to your local command-line session, and runs /bin/bash.

1. $ docker run -i -t ubuntu /bin/bash
   When you run this command, the following happens (assuming you are using the default registry configuration):

2. If you don't have the ubuntu image locally, Docker pulls it from your configured registry, as though you had run docker pull ubuntu manually.

3. Docker creates a new container, as though you had run a docker container create command manually.

4. Docker allocates a read-write filesystem to the container, as its final layer. This allows a running container to create or modify files and directories in its local filesystem.

5. Docker creates a network interface to connect the container to the default network, since you didn't specify any networking options. This includes assigning an IP address to the container. By default, containers can connect to external networks using the host machine's network connection.

6. Docker starts the container and executes /bin/bash. Because the container is running interactively and attached to your terminal (due to the -i and -t flags), you can provide input using your keyboard while Docker logs the output to your terminal.

7. When you run exit to terminate the /bin/bash command, the container stops but isn't removed. You can start it again or remove it.

The underlying technology
Docker is written in the Go programming language and takes advantage of several features of the Linux kernel to deliver its functionality. Docker uses a technology called namespaces to provide the isolated workspace called the container. When you run a container, Docker creates a set of namespaces for that container.

These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

3) Difference between Docker and Virtual machine

Both virtual machines (VMs) and Docker address the challenge of running applications across different environments. But they do so for slightly different reasons and with different approaches.

Objective
Virtual machines were originally designed to allow multiple operating systems to run on a single physical machine. The objective is to allow users to create a virtual environment that's isolated from the underlying hardware. VMs abstract hardware details to make it easier to run applications on different hardware architectures and use hardware resources more efficiently.

Docker, on the other hand, was designed to provide a lightweight and portable way to package and run applications in an isolated and reproducible environment. Docker abstracts operating system details to address the challenge of deploying applications across different environments, such as development, testing, and production. It can be very challenging to manage software environment updates and maintain environment consistency everywhere. This is especially true for organizations that run hundreds of applications or decompose applications into hundreds of microservices. Docker addresses this problem through containerization.

End product
Docker is the name of the open-source container platform that's owned and operated by the company Docker. There are alternative platforms like Podman, although they're far less popular; Docker is synonymous with containerization. The container is the artifact, the usable part for the end user.

A virtual machine itself is the usable part for the end user. The technology isn't associated with a specific brand. You can deploy VMs in on-premises data centers or access them via APIs as a managed cloud service.

Architecture
A virtual machine runs its own kernel and host operating system, along with applications and their dependencies like libraries and other binary files. A hypervisor coordinates between the hardware (host machine or server) and the virtual machine. It allocates the physical hardware resources outlined during instantiation to the virtual machine for its exclusive use. Multiple virtual machines can exist on a single powerful server, managed by a single hypervisor, with hundreds of applications running on each virtual machine.

A Docker container contains only its dependencies. The software Docker Engine powers virtualization in Docker. It provides coordination between running containers and the underlying operating system, whether it's a physical or virtual machine.

For more advanced virtualization management with Docker, use Kubernetes. For more information, read What's the Difference Between Kubernetes And Docker?

Resource sharing
Both virtual machines and Docker containers use resource multiplexing, or resource sharing between virtualized instances.

Virtual machines request a specific amount of the resource up-front from the hardware and continue to steadily occupy that amount, so long as the virtual machine is running.

Docker containers, on the other hand, use resources on demand. Rather than asking for a specific amount of physical hardware resourcing as virtual machines do, they simply request what they need from the single operating system kernel. Multiple containers share the same operating system. Docker containers direct resource sharing with the kernel leads and may use less system resources compared to a VM.

Security
Because Docker containers share the kernel with the host operating system, for lightweight resource consumption, they're at risk if there are vulnerabilities in the kernel. However, Docker also provides many advanced security controls.

Conversely, as a VM runs an entire operating system, there's an added level of isolation when running applications. VMs offer higher security as long as the operating system has strict security measures in place.

4) Docker Commands

- **docker version**

  Syntax: docker version [OPTIONS]
  Description: Displays information about the Docker client and server versions, including version numbers, build information, and more.

- **docker info**

  Syntax: docker info [OPTIONS]
  Description: Provides detailed information about the Docker system, such as the number of containers and images, storage usage, and system-related information.

- **docker pull**

  Syntax: docker pull [OPTIONS] NAME[:TAG|@DIGEST]
  Description: Downloads an image or a repository from a registry. The NAME is the name of the image, and TAG is an optional version tag (e.g., latest).

- **docker images**

  Syntax: docker images [OPTIONS] [REPOSITORY[:TAG]]
  Description: Lists all local images. You can filter the list based on the repository and tag.

- **docker ps**

  Syntax: docker ps [OPTIONS]
  Description: Lists all currently running containers. The -a option can be added to include stopped containers.

- **docker ps -a**

  Syntax: docker ps -a [OPTIONS]

Description: Lists all containers, both running and stopped.

- **docker run**

  Syntax: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
  Description: Creates and runs a new container based on the specified image. You can also specify a command to run inside the container.

- **docker exec**

  Syntax: docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
  Description: Runs a command inside a running container. This is useful for executing commands in an already running container.

- **docker build**

  Syntax: docker build [OPTIONS] PATH | URL | -
  Description: Builds a Docker image from a Dockerfile located in the specified path or URL. The . at the end of the command denotes the current directory.

- **docker stop**

  Syntax: docker stop [OPTIONS] CONTAINER [CONTAINER...]
  Description: Stops one or more running containers. You can use the container's name or ID.

- **docker start**

  Syntax: docker start [OPTIONS] CONTAINER [CONTAINER...]
  Description: Starts one or more stopped containers. Again, you can use the container's name or ID.

- **docker restart**

  Syntax: docker restart [OPTIONS] CONTAINER [CONTAINER...]
  Description: Restarts one or more containers, stopping and then starting them.

- **docker rm**

  Syntax: docker rm [OPTIONS] CONTAINER [CONTAINER...]
  Description: Removes one or more containers. Use the -f option to force removal, even if the container is running.

- **docker rmi**

  Syntax: docker rmi [OPTIONS] IMAGE [IMAGE...]
  Description: Removes one or more images. You can specify the image by its name or ID.

- **docker network ls**

Syntax: docker network ls [OPTIONS]
Description: Lists all networks. This command provides information about the networks created by Docker.

- **docker volume ls**

  Syntax: docker volume ls [OPTIONS]
  Description: Lists all volumes. Docker volumes are used to persist data generated by and used by Docker containers.

- **docker-compose up**

  Syntax: docker-compose up [OPTIONS] [SERVICE...]
  Description: Builds, (re)creates, starts, and attaches to containers for a service defined in a docker-compose.yml file.

- **docker-compose down**

  Syntax: docker-compose down [OPTIONS]
  Description: Stops and removes containers, networks, and volumes defined in a docker-compose.yml file.

5) Dockerfile

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. This page describes the commands you can use in a Dockerfile.

Format

Here is the format of the Dockerfile:

```
# Comment
INSTRUCTION arguments
```

The instruction is not case-sensitive. However, convention is for them to be UPPERCASE to distinguish them from arguments more easily.

Docker runs instructions in a Dockerfile in order. A Dockerfile must begin with a FROM instruction. This may be after parser directives, comments, and globally scoped ARGs. The FROM instruction specifies the Parent Image from which you are building. FROM may only be preceded by one or more ARG instructions, which declare arguments that are used in FROM lines in the Dockerfile.

Docker treats lines that begin with # as a comment, unless the line is a valid parser directive. A # marker anywhere else in a line is treated as an argument. This allows statements like:

```
# Comment
RUN echo 'we are running some # of cool things'
```

Comment lines are removed before the Dockerfile instructions are executed, which means that the comment in the following example is not handled by the shell executing the echo command, and both examples below are equivalent:

```
RUN echo hello \
# comment
world
RUN echo hello \
world
```

Line continuation characters are not supported in comments.

Parser directives

Parser directives are optional, and affect the way in which subsequent lines in a Dockerfile are handled. Parser directives do not add layers to the build, and will not be shown as a build step. Parser directives are written as a special type of comment in the form # directive=value. A single directive may only be used once.

Once a comment, empty line or builder instruction has been processed, Docker no longer looks for parser directives. Instead it treats anything formatted as a parser directive as a comment and does not attempt to validate if it might be a parser directive. Therefore, all parser directives must be at the very top of a Dockerfile.

Parser directives are not case-sensitive. However, convention is for them to be lowercase. Convention is also to include a blank line following any parser directives. Line continuation characters are not supported in parser directives.

Due to these rules, the following examples are all invalid:

Invalid due to line continuation:

```
# direc \
tive=value
```

Invalid due to appearing twice:

```
# directive=value1
# directive=value2

FROM ImageName
```

Treated as a comment due to appearing after a builder instruction:

```
FROM ImageName
# directive=value
```

Treated as a comment due to appearing after a comment which is not a parser directive:

```
# About my dockerfile
# directive=value
FROM ImageName
```

The unknown directive is treated as a comment due to not being recognized. In addition, the known directive is treated as a comment due to appearing after a comment which is not a parser directive.

```
# unknowndirective=value
# knowndirective=value
```

Non line-breaking whitespace is permitted in a parser directive. Hence, the following lines are all treated identically:

```
#directive=value
# directive =value
#   directive= value
# directive = value
#     dIrEcTiVe=value
```

The following parser directives are supported:

```
syntax
escape
syntax
```

This feature is only available when using the BuildKit backend, and is ignored when using the classic builder backend.

See Custom Dockerfile syntax page for more information.

```
escape
# escape=\ (backslash)
```

Or

```
# escape=` (backtick)
```

The escape directive sets the character used to escape characters in a Dockerfile. If not specified, the default escape character is \.

The escape character is used both to escape characters in a line, and to escape a newline. This allows a Dockerfile instruction to span multiple lines. Note that regardless of whether the escape parser directive is included in a Dockerfile, escaping is not performed in a RUN command, except at the end of a line.

Setting the escape character to ` is especially useful on Windows, where \ is the directory path separator. ` is consistent with Windows PowerShell.

Consider the following example which would fail in a non-obvious way on Windows. The second \ at the end of the second line would be interpreted as an escape for the newline, instead of a target of the escape from the first \. Similarly, the \ at the end of the third line would, assuming it was actually handled as an instruction, cause it be treated as a line continuation. The result of this dockerfile is that second and third lines are considered a single instruction:

```
FROM microsoft/nanoserver
COPY testfile.txt c:\\
RUN dir c:\
```

Results in:

```
PS E:\myproject> docker build -t cmd .

Sending build context to Docker daemon 3.072 kB
Step 1/2 : FROM microsoft/nanoserver
 ---> 22738ff49c6d
Step 2/2 : COPY testfile.txt c:\RUN dir c:
GetFileAttributesEx c:RUN: The system cannot find the file specified.
PS E:\myproject>
```

One solution to the above would be to use / as the target of both the COPY instruction, and dir. However, this syntax is, at best, confusing as it is not natural for paths on Windows, and at worst, error prone as not all commands on Windows support / as the path separator.

By adding the escape parser directive, the following Dockerfile succeeds as expected with the use of natural platform semantics for file paths on Windows:

```
# escape=`

FROM microsoft/nanoserver
COPY testfile.txt c:\
RUN dir c:\
```

Results in:

```
PS E:\myproject> docker build -t succeeds --no-cache=true .

Sending build context to Docker daemon 3.072 kB
Step 1/3 : FROM microsoft/nanoserver
 ---> 22738ff49c6d
Step 2/3 : COPY testfile.txt c:\
 ---> 96655de338de
Removing intermediate container 4db9acbb1682
Step 3/3 : RUN dir c:\
 ---> Running in a2c157f842f5
 Volume in drive C has no label.
 Volume Serial Number is 7E6D-E0F7

 Directory of c:\

10/05/2016  05:04 PM         1,894 License.txt
10/05/2016  02:22 PM    DIR        Program Files
10/05/2016  02:14 PM    DIR        Program Files (x86)
10/28/2016  11:18 AM            62 testfile.txt
10/28/2016  11:20 AM    DIR        Users
```

```
10/28/2016  11:20 AM    DIR         Windows
         2 File(s)         1,956 bytes
         4 Dir(s)  21,259,096,064 bytes free
 ---> 01c7f3bef04f
Removing intermediate container a2c157f842f5
Successfully built 01c7f3bef04f
PS E:\myproject>
```

Environment replacement
Environment variables (declared with the ENV statement) can also be used in certain instructions as variables to be interpreted by the Dockerfile. Escapes are also handled for including variable-like syntax into a statement literally.

Environment variables are notated in the Dockerfile either with $variable_name or ${variable_name}. They are treated equivalently and the brace syntax is typically used to address issues with variable names with no whitespace, like ${foo}_bar.

The ${variable_name} syntax also supports a few of the standard bash modifiers as specified below:

${variable:-word} indicates that if variable is set then the result will be that value. If variable is not set then word will be the result.
${variable:+word} indicates that if variable is set then word will be the result, otherwise the result is the empty string.
In all cases, word can be any string, including additional environment variables.

Escaping is possible by adding a \ before the variable: \$foo or \${foo}, for example, will translate to $foo and ${foo} literals respectively.

Example (parsed representation is displayed after the #):

```
FROM busybox
ENV FOO=/bar
WORKDIR ${FOO}   # WORKDIR /bar
ADD . $FOO       # ADD . /bar
COPY \$FOO /quux # COPY $FOO /quux
```

Environment variables are supported by the following list of instructions in the Dockerfile:

ADD
COPY
ENV
EXPOSE
FROM
LABEL
STOPSIGNAL
USER
VOLUME
WORKDIR
ONBUILD (when combined with one of the supported instructions above)

Environment variable substitution will use the same value for each variable throughout the entire instruction. In other words, in this example:

```
ENV abc=hello
ENV abc=bye def=$abc
ENV ghi=$abc
```

will result in def having a value of hello, not bye. However, ghi will have a value of bye because it is not part of the same instruction that set abc to bye.

.dockerignore file
You can use .dockerignore file to exclude files and directories from the build context. For more information, see .dockerignore file.

6) Docker-Compose and Docker-swarm

Docker-Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Compose works in all environments; production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

The key features of Compose that make it effective are:

- Have multiple isolated environments on a single host
- Preserve volume data when containers are created
- Only recreate containers that have changed
- Support variables and moving a composition between environments

Key features and use cases of Docker Compose
Using Compose is essentially a three-step process:

1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.

2. Define the services that make up your app in a compose.yaml file so they can be run together in an isolated environment.

3. Run docker compose up and the Docker compose command starts and runs your entire app.

A compose.yaml looks like this:

```
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    depends_on:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

For more information about the Compose file, see the Compose file reference.

Key features of Docker Compose

Have multiple isolated environments on a single host

Compose uses a project name to isolate environments from each other. You can make use of this project name in several different contexts:

- On a dev host, to create multiple copies of a single environment, such as when you want to run a stable copy for each feature branch of a project
- On a CI server, to keep builds from interfering with each other, you can set the project name to a unique build number
- On a shared host or dev host, to prevent different projects, which may use the same service names, from interfering with each other

The default project name is the base name of the project directory. You can set a custom project name by using the -p command line option or the COMPOSE_PROJECT_NAME environment variable.

The default project directory is the base directory of the Compose file. A custom value for it can be defined with the --project-directory command line option.

Preserves volume data when containers are created.

Compose preserves all volumes used by your services. When docker composes up runs, if it finds any containers from previous runs, it copies the volumes from the old container to the new container. This process ensures that any data you've created in volumes isn't lost.

If you use docker compose on a Windows machine, see Environment variables, and adjust the necessary environment variables for your specific needs.

Only recreate containers that have changed.

Composition caches the configuration used to create a container. When you restart a service that has not changed, compose re-uses the existing containers. Re-using containers means that you can make changes to your environment very quickly.

Supports variables and moving a composition between environments.
Compose supports variables in the Compose file. You can use these variables to customize your composition for different environments, or different users.

You can extend a Compose file using the extends field or by creating multiple Compose files. For more details, see Working with Multiple Compose files.

Common use cases of Docker Compose
Composing can be used in many different ways. Some common use cases are outlined below.

Development environments
When you're developing software, the ability to run an application in an isolated environment and interact with it is crucial. The Compose command line tool can be used to create the environment and interact with it.

The Compose file provides a way to document and configure all of the application's service dependencies (databases, queues, caches, web service APIs, etc). Using the Compose command line tool you can create and start one or more containers for each dependency with a single command (docker compose up).

Together, these features provide a convenient way for developers to get started on a project. Compose can reduce a multi-page "developer getting started guide" to a single machine readable Compose file and a few commands.

Automated testing environments
An important part of any Continuous Deployment or Continuous Integration process is the automated test suite. Automated end-to-end testing requires an environment in which to run tests. Compose provides a convenient way to create and destroy isolated testing environments for your test suite. By defining the full environment in a Compose file, you can create and destroy these environments in just a few commands:

```
$ docker compose up -d
$ ./run_tests
$ docker compose down
```

Single host deployments
Compose has traditionally been focused on development and testing workflows, but with each release we're making progress on more production-oriented features.

Docker Swarm
To use Docker in swarm mode, install Docker. See installation instructions for all operating systems and platforms.

Current versions of Docker include swarm mode for natively managing a cluster of Docker Engines called a swarm. Use the Docker CLI to create a swarm, deploy application services to a swarm, and manage swarm behavior.

Docker Swarm mode is built into the Docker Engine. Do not confuse Docker Swarm mode with Docker Classic Swarm which is no longer actively developed.

Feature highlights

- **Cluster management integrated with Docker Engine**: Use the Docker Engine CLI to create a swarm of Docker Engines where you can deploy application services. You don't need additional orchestration software to create or manage a swarm.

- **Decentralized design**: Instead of handling differentiation between node roles at deployment time, the Docker Engine handles any specialization at runtime. You can deploy both kinds of nodes, managers and workers, using the Docker Engine. This means you can build an entire swarm from a single disk image.

- **Declarative service model**: Docker Engine uses a declarative approach to let you define the desired state of the various services in your application stack. For example, you might describe an application consisting of a web front end service with message queueing services and a database backend.

- **Scaling**: For each service, you can declare the number of tasks you want to run. When you scale up or down, the swarm manager automatically adapts by adding or removing tasks to maintain the desired state.

- **Desired state reconciliation**: The swarm manager node constantly monitors the cluster state and reconciles any differences between the actual state and your expressed desired state. For example, if you set up a service to run 10 replicas of a container, and a worker machine hosting two of those replicas crashes, the manager creates two new replicas to replace the replicas that crashed. The swarm manager assigns the new replicas to workers that are running and available.

- **Multi-host networking**: You can specify an overlay network for your services. The swarm manager automatically assigns addresses to the containers on the overlay network when it initializes or updates the application.

- **Service discovery**: Swarm manager nodes assign each service in the swarm a unique DNS name and load balance running containers. You can query every container running in the swarm through a DNS server embedded in the swarm.

- **Load balancing**: You can expose the ports for services to an external load balancer. Internally, the swarm lets you specify how to distribute service containers between nodes.

- **Secure by default**: Each node in the swarm enforces TLS mutual authentication and encryption to secure communications between itself and all other nodes. You have the option to use self-signed root certificates or certificates from a custom root CA.

- **Rolling updates**: At rollout time you can apply service updates to nodes incrementally. The swarm manager lets you control the delay between service

deployment to different sets of nodes. If anything goes wrong, you can roll back to a previous version of the service.

Swarm mode key concepts
This topic introduces some of the concepts unique to the cluster management and orchestration features of Docker Engine 1.12.

What is a swarm?
The cluster management and orchestration features embedded in the Docker Engine are built using swarm kit. Swarm kit is a separate project which implements Docker's orchestration layer and is used directly within Docker.

A swarm consists of multiple Docker hosts which run in swarm mode and act as managers (to manage membership and delegation) and workers (which run swarm services). A given Docker host can be a manager, a worker, or perform both roles. When you create a service, you define its optimal state (number of replicas, network and storage resources available to it, ports the service exposes to the outside world, and more). Docker works to maintain that desired state. For instance, if a worker node becomes unavailable, Docker schedules that node's tasks on other nodes. A task is a running container which is part of a swarm service and is managed by a swarm manager, as opposed to a standalone container.

One of the key advantages of swarm services over standalone containers is that you can modify a service's configuration, including the networks and volumes it is connected to, without the need to manually restart the service. Docker will update the configuration, stop the service tasks with out-of-date configuration, and create new ones matching the desired configuration.

When Docker is running in swarm mode, you can still run standalone containers on any of the Docker hosts participating in the swarm, as well as swarm services. A key difference between standalone containers and swarm services is that only swarm managers can manage a swarm, while standalone containers can be started on any daemon. Docker daemons can participate in a swarm as managers, workers, or both.

In the same way that you can use Docker Compose to define and run containers, you can define and run Swarm service stacks.

Keep reading for details about concepts related to Docker swarm services, including nodes, services, tasks, and load balancing.

Nodes
A node is an instance of the Docker engine participating in the swarm. You can also think of this as a Docker node. You can run one or more nodes on a single physical computer or cloud server, but production swarm deployments typically include Docker nodes distributed across multiple physical and cloud machines.

To deploy your application to a swarm, you submit a service definition to a manager node. The manager node dispatches units of work called tasks to worker nodes.

Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm. Manager nodes elect a single leader to conduct orchestration tasks.

Worker nodes receive and execute tasks dispatched from manager nodes. By default manager nodes also run services as worker nodes, but you can configure them to run manager tasks exclusively and be manager-only nodes. An agent runs on each worker node and reports on the tasks assigned to it. The worker node notifies the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker.

Services and tasks
A service is the definition of the tasks to execute on the manager or worker nodes. It is the central structure of the swarm system and the primary root of user interaction with the swarm.

When you create a service, you specify which container image to use and which commands to execute inside running containers.

In the replicated services model, the swarm manager distributes a specific number of replica tasks among the nodes based upon the scale you set in the desired state.

For global services, the swarm runs one task for the service on every available node in the cluster.

A task carries a Docker container and the commands to run inside the container. It is the atomic scheduling unit of swarm. Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale. Once a task is assigned to a node, it cannot move to another node. It can only run on the assigned node or fail.

Load balancing
The swarm manager uses ingress load balancing to expose the services you want to make available externally to the swarm. The swarm manager can automatically assign the service a Published Port or you can configure a Published Port for the service. You can specify any unused port. If you do not specify a port, the swarm manager assigns the service a port in the 30000-32767 range.

External components, such as cloud load balancers, can access the service on the Published Port of any node in the cluster whether the node is currently running the task for the service. All nodes in the swarm route ingress connections to a running task instance.

Swarm mode has an internal DNS component that automatically assigns each service in the swarm a DNS entry. The swarm manager uses internal load balancing to distribute requests among services within the cluster based upon the DNS name of the service.

Create a swarm.
After you complete the tutorial setup steps, you're ready to create a swarm. Make sure the Docker Engine daemon is started on the host machines.

Open a terminal and ssh into the machine where you want to run your manager node. This tutorial uses a machine named manager1.

Run the following command to create a new swarm:

```
$ docker swarm init --advertise-addr <MANAGER-IP>
```

In the tutorial, the following command creates a swarm on the manager1 machine:

```
$ docker swarm init --advertise-addr 192.168.99.100
```

Swarm initialized: current node (dxn1zf6l61qsb1josjja83ngz) is now a manager.

To add a worker to this swarm, run the following command:

```
    docker swarm join \
    --token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-
8vxv8rssmk743ojnwacrr2e7c \
    192.168.99.100:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
The --advertise-addr flag configures the manager node to publish its address as 192.168.99.100. The other nodes in the swarm must be able to access the manager at the IP address.

The output includes the commands to join new nodes to the swarm. Nodes will join as managers or workers depending on the value for the --token flag.

Run docker info to view the current state of the swarm:

```
$ docker info

Containers: 2
Running: 0
Paused: 0
Stopped: 2
  ...snip...
Swarm: active
  NodeID: dxn1zf6l61qsb1josjja83ngz
  Is Manager: true
  Managers: 1
  Nodes: 1
  ...snip…
```

Run the docker node ls command to view information about nodes:

```
$ docker node ls

ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
dxn1zf6l61qsb1josjja83ngz *  manager1  Ready   Active        Leader
```

The * next to the node ID indicates that you're currently connected on this node.

Implementation:

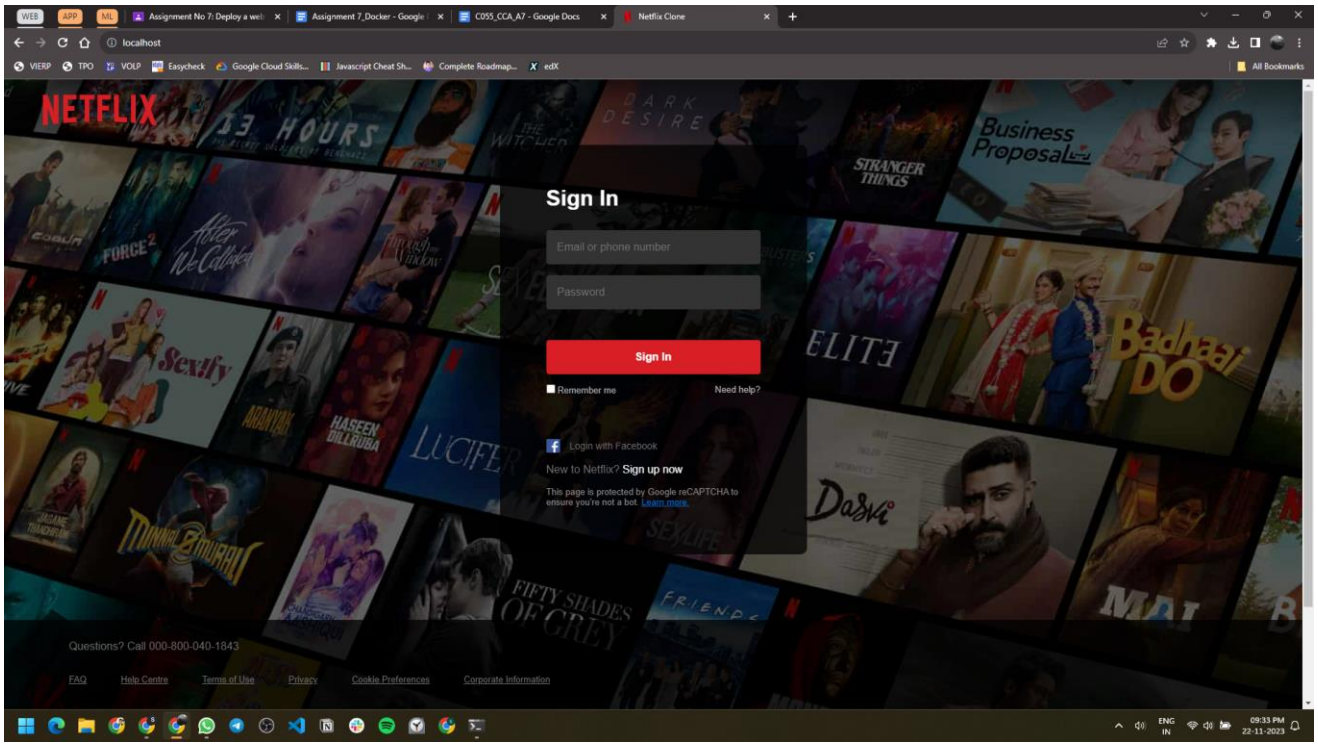
Step 1: Install nginx on windows follow the link:

http://nginx.org/en/docs/windows.html

```
cd c:\
unzip nginx-1.23.4.zip
cd nginx-1.23.4
start nginx
```

Step 2: Copy the sample-website in "C:\nginx\html\" folder

Step 3: open browser and run "localhost:80"



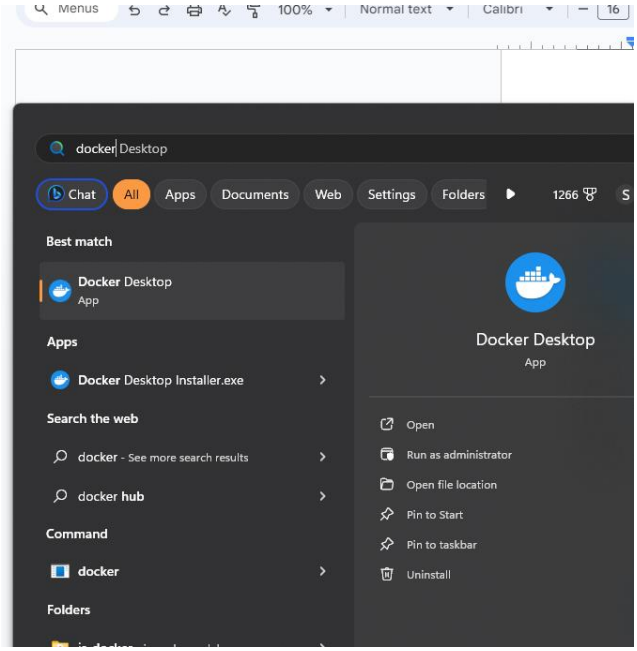
Step 4: Download Docker for windows, follow the link

https://docs.docker.com/desktop/install/windows-install/

Step 5: Start Docker Desktop

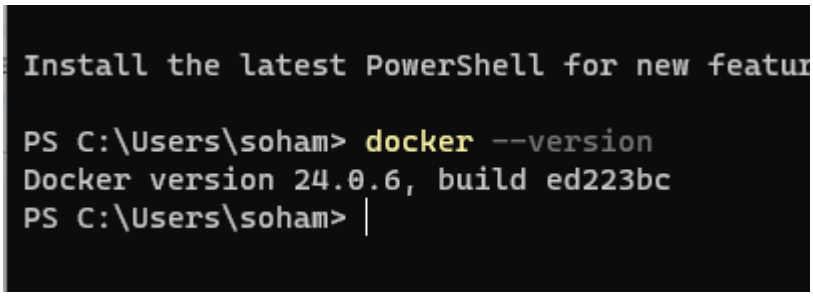
Docker Desktop does not start automatically after installation. To start Docker Desktop:

1. Search for Docker and select **Docker Desktop** in the search results.



Step 6: Open Powershell and check Docker installation using commands:

a. docker –version



b. docker info

```
PS C:\Users\soham> docker info
Client:
 Version:    24.0.6
 Context:    default
 Debug Mode: false
 Plugins:
  buildx: Docker Buildx (Docker Inc.)
    Version:  v0.11.2-desktop.5
    Path:     C:\Program Files\Docker\cli-plugins\docker-buildx.exe
  compose: Docker Compose (Docker Inc.)
    Version:  v2.23.0-desktop.1
    Path:     C:\Program Files\Docker\cli-plugins\docker-compose.exe
  dev: Docker Dev Environments (Docker Inc.)
    Version:  v0.1.0
    Path:     C:\Program Files\Docker\cli-plugins\docker-dev.exe
  extension: Manages Docker extensions (Docker Inc.)
    Version:  v0.2.20
    Path:     C:\Program Files\Docker\cli-plugins\docker-extension.exe
  init: Creates Docker-related starter files for your project (Docker Inc.)
    Version:  v0.1.0-beta.9
    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
  scan: Docker Scan (Docker Inc.)
    Version:  v0.26.0
    Path:     C:\Program Files\Docker\cli-plugins\docker-scan.exe
  scout: Docker Scout (Docker Inc.)
    Version:  v1.0.9
    Path:     C:\Program Files\Docker\cli-plugins\docker-scout.exe

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 0
 Server Version: 24.0.6
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Using metacopy: false
  Native Overlay Diff: true
  userxattr: false
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Cgroup Version: 1
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
```

```
 Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 Swarm: inactive
 Runtimes: io.containerd.runc.v2 runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: 8165feabfdfe38c65b599c4993d227328c231fca
 runc version: v1.1.8-0-g82f18fe
 init version: de40ad0
 Security Options:
  seccomp
   Profile: unconfined
 Kernel Version: 5.15.133.1-microsoft-standard-WSL2
 Operating System: Docker Desktop
 OSType: linux
 Architecture: x86_64
 CPUs: 8
 Total Memory: 7.637GiB
 Name: Soham
 ID: b6ec94e8-afc2-4946-ac33-bef768f7f53e
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
 HTTP Proxy: http.docker.internal:3128
 HTTPS Proxy: http.docker.internal:3128
 No Proxy: hubproxy.docker.internal
 Experimental: false
 Insecure Registries:
  hubproxy.docker.internal:5555
  127.0.0.0/8
 Live Restore Enabled: false

WARNING: No blkio throttle.read_bps_device support
WARNING: No blkio throttle.write_bps_device support
WARNING: No blkio throttle.read_iops_device support
WARNING: No blkio throttle.write_iops_device support
WARNING: daemon is not using the default seccomp profile
PS C:\Users\soham>
```

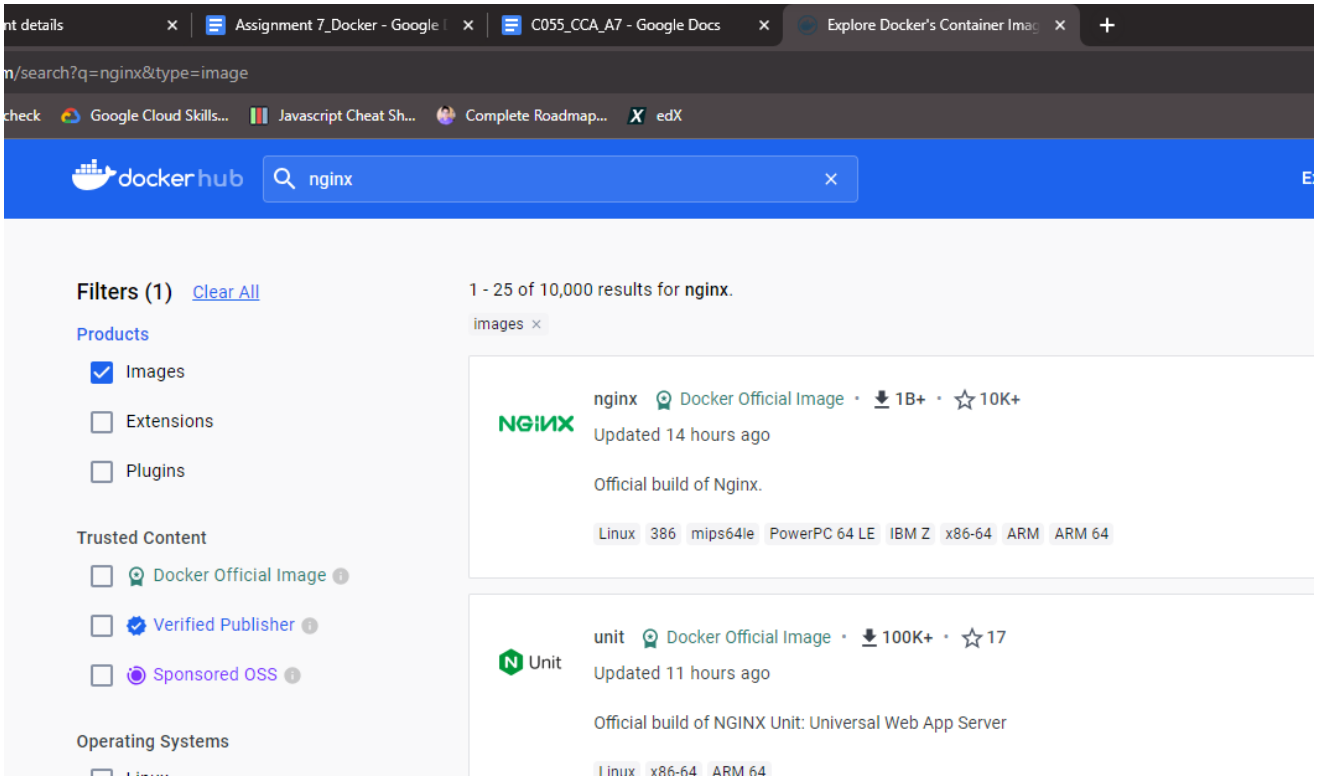
c. docker version --format '{{json.}} `

```
PS C:\Users\soham> docker version --format '{{json .}}'
{"Client":{"CloudIntegration":"v1.0.35+desktop.5","Version":"24.0.6","ApiVersion":"1.43","DefaultAPIVersion":"1.
ep  4 12:32:48 2023","Context":"default"},"Server":{"Platform":{"Name":"Docker Desktop 4.25.2 (129061)"},"Compon
me":"Mon Sep  4 12:32:16 2023","Experimental":"false","GitCommit":"1a79695","GoVersion":"go1.20.7","KernelVersio
","Version":"1.6.22","Details":{"GitCommit":"8165feabfdfe38c65b599c4993d227328c231fca"}},{"Name":"runc","Version
Details":{"GitCommit":"de40ad0"}}],"Version":"24.0.6","ApiVersion":"1.43","MinAPIVersion":"1.12","GitCommit":"1a
tandard-WSL2","BuildTime":"2023-09-04T12:32:16.000000000+00:00"}}
PS C:\Users\soham>
```

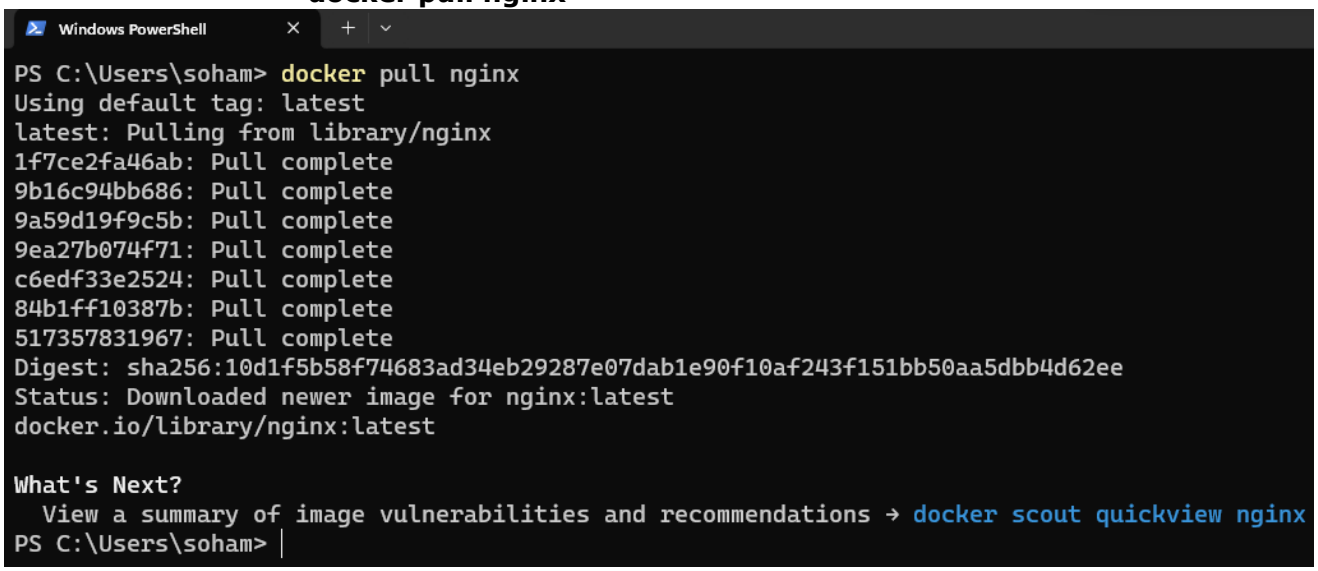
- # Steps to run the “Sample website” in Docker container.

Step 1) visit to Docker hub web site: https://hub.docker.com/

Step 2) search for “nginx” image on site



Step 3) pull the latest image of nginx using command "docker pull nginx"



Step 4) check the docker images on your desktop by using command:

"docker images"

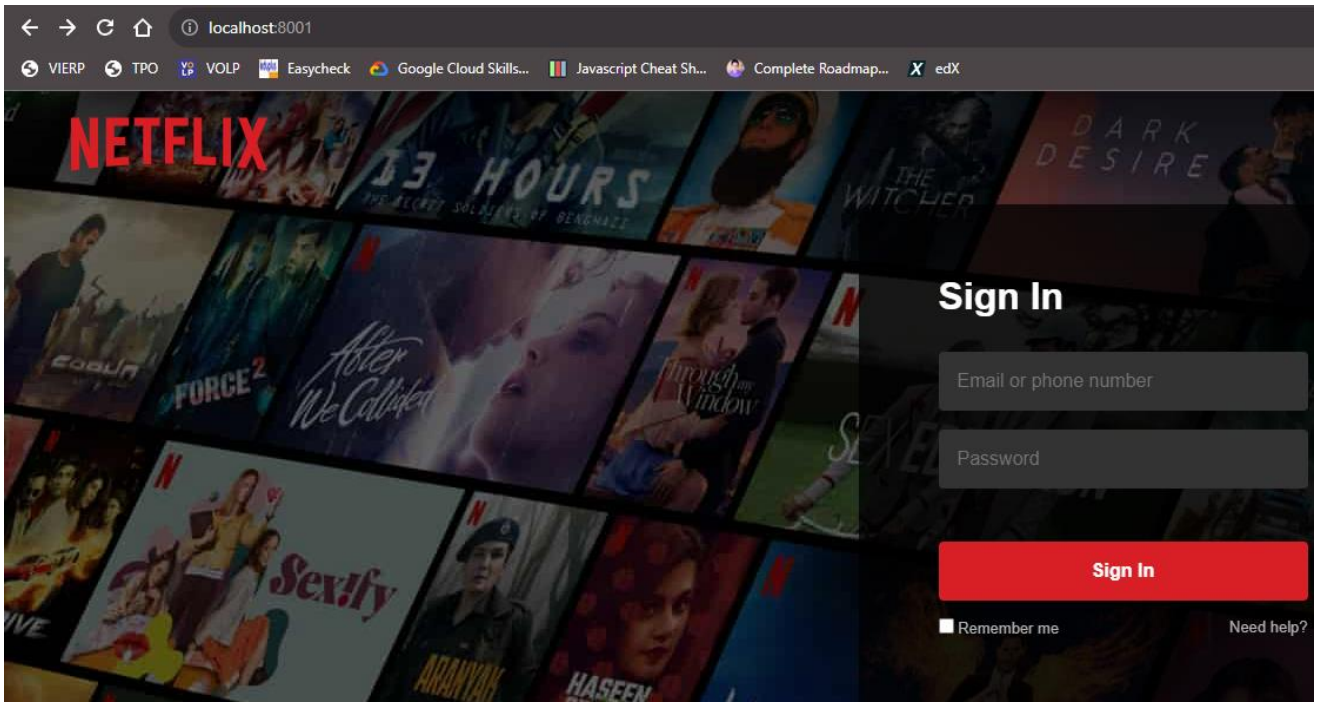
```
PS C:\Users\soham> docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
nginx        latest    a6bd71f48f68   31 hours ago    187MB
PS C:\Users\soham>
```

Step 5) go in the "SampleWebsite" folder and then Create a container using the docker command and sync the "SampleWebsite" folder with folder inside the container folder. (This is called Mount Bind")

"docker run -d -p 8001:80 -v ${PWD}:/usr/share/nginx/html --name web-site nginx"

```
PS D:\Downloads\MiscSetups\nginx-1.24.0\nginx-1.24.0> docker run -d -p 8001:80 -v ${PWD}:/html --name web-site nginx
42eaf493182679414f5059dce3589f1747da5538c9034c321fb16ffc8f0e75e2
PS D:\Downloads\MiscSetups\nginx-1.24.0\nginx-1.24.0> docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS                  NAMES
42eaf4931826   nginx     "/docker-entrypoint.…"   7 seconds ago   Up 6 seconds   0.0.0.0:8001->80/tcp   web-site
PS D:\Downloads\MiscSetups\nginx-1.24.0\nginx-1.24.0>
```

Step 6)verify the website open browser and chec "localhost:8001". Now this website is running inside your container.



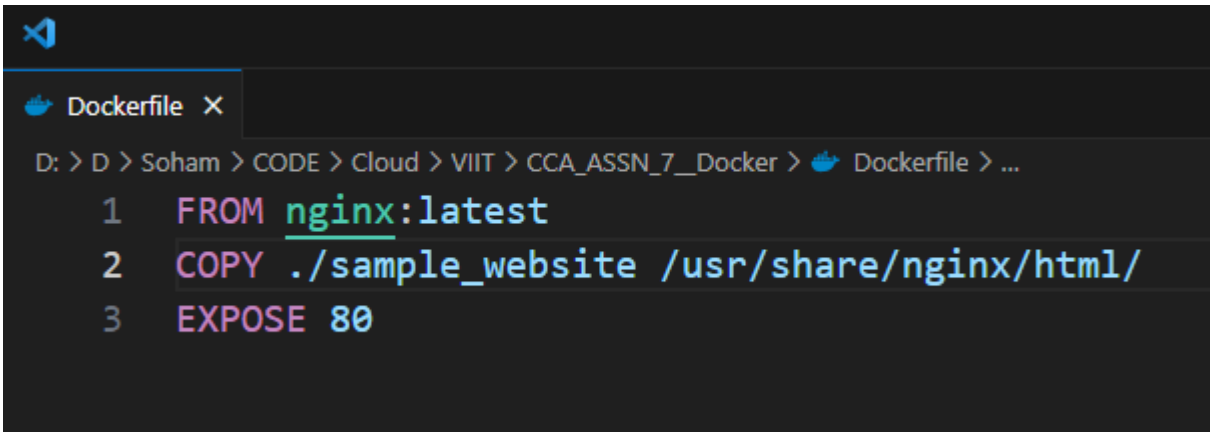
DockerFile

Step 1) Create a Directory structure like

App

- SampleWebSite
- Dockerfile

Step 2) Write a following script into "Dockerfile"



Step 3) build image from docker file using command

"docker build -t my-app:v1 . "

```
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker> docker build -t my-app:v1 .
[+] Building 0.4s (7/7) FINISHED                                                                    docker:default
 => [internal] load build definition from Dockerfile                                                        0.1s
 => => transferring dockerfile: 111B                                                                        0.0s
 => [internal] load .dockerignore                                                                           0.1s
 => => transferring context: 2B                                                                             0.0s
 => [internal] load metadata for docker.io/library/nginx:latest                                             0.0s
 => [1/2] FROM docker.io/library/nginx:latest                                                               0.2s
 => [internal] load build context                                                                           0.1s
 => => transferring context: 36B                                                                            0.0s
 => [2/2] COPY ./sample_website /usr/share/nginx/html/                                                      0.0s
 => exporting to image                                                                                      0.0s
 => => exporting layers                                                                                     0.0s
 => => writing image sha256:1b9602ec82eb655b662e6c8dc4b05dce38f6c1cd9b91a95fce67eaf4b5db7814                0.0s
 => => naming to docker.io/library/my-app:v1                                                                0.0s

What's Next?
  1. Sign in to your Docker account → docker login
  2. View a summary of image vulnerabilities and recommendations → docker scout quickview
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker>
```

Step 4) check images using command: docker images

```
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker> docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
my-app       v1        1b9602ec82eb   36 seconds ago   187MB
nginx        latest    a6bd71f48f68   32 hours ago     187MB
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker>
```

PUSH Image to "DockerHub"

Step 1) login to docker hub using command

1) **docker login -u soham12345**

```
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker> docker login -u soham12345
Password:
Login Succeeded
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker>
```

2) docker images

```
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker> docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
my-app       v1        1b9602ec82eb   4 minutes ago    187MB
nginx        latest    a6bd71f48f68   32 hours ago     187MB
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker>
```

3) docker tag (old image name) soham12345/my-app:v1

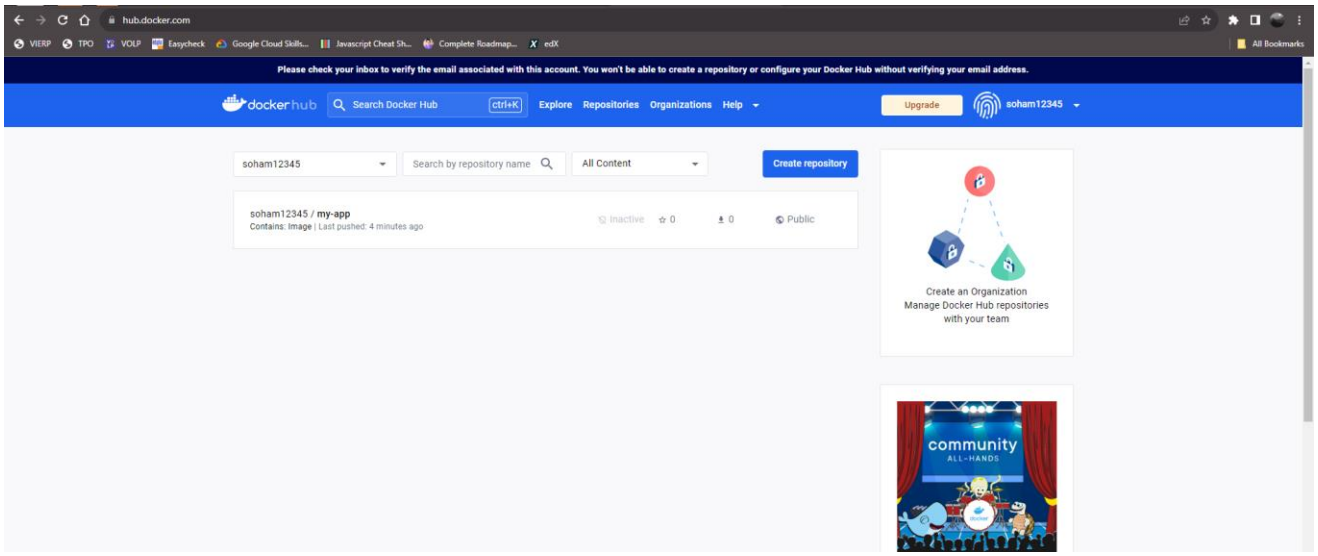
docker tag my-web:v1 soham12345/my-app:v1

```
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker> docker tag my-app:v1 soham12345/my-app:v1
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker>
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker> docker images
REPOSITORY          TAG       IMAGE ID       CREATED          SIZE
my-app              v1        1b9602ec82eb   5 minutes ago    187MB
soham12345/my-app   v1        1b9602ec82eb   5 minutes ago    187MB
nginx               latest    a6bd71f48f68   32 hours ago     187MB
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker>
```

4) docker push soham12345/my-app:v1

```
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker> docker push soham12345/my-app:v1
The push refers to repository [docker.io/soham12345/my-app]
418e0c6d9595: Pushed
0d0e9c83b6f7: Mounted from library/nginx
cddc309885a2: Mounted from library/nginx
c2d3ab485d1b: Mounted from library/nginx
66283570f41b: Mounted from library/nginx
f5525891d9e9: Mounted from library/nginx
8ae474e0cc8f: Mounted from library/nginx
92770f546e06: Mounted from library/nginx
v1: digest: sha256:4bee448a8c45be99ea3ad89c47a7e163ec71ea6ec8a2eafab4ea269178ad35dd size: 1985
PS D:\D\Soham\CODE\Cloud\VIIT\CCA_ASSN_7__Docker>
```

Step 5) Login to Docker Hub and check the repository



Conclusion: Thus, we have successfully Deploy a web application using Docker.