Continuous Deployment

Dealing with Containerization and Docker

Contents –

Contents –

1

Introduction

What is a Virtual Machine (VM)?

A **Virtual Machine (VM)** is a software-based simulation of a physical computer that runs an operating system (OS) and applications just like a physical machine.

It operates within a host machine but is completely isolated, meaning it has its own virtual CPU, memory, storage, and network resources.

Why is a Virtual Machine Required in SDLC?

- Development and Testing Environments –
  - Consistency: VMs allow developers to set up identical environments across different machines, preventing the "works on my machine" issue.
  - Isolation: Developers can experiment with new tools and dependencies without affecting the host system.

Why is a Virtual Machine Required in SDLC?

- Development and Testing Environments –
  - Multiple OS Support: Teams can develop and test applications across different operating systems without needing multiple physical machines.
  - Example: A developer working on a Windows machine can set up a VM with Linux to test an application that will be deployed on Linux servers.

Why is a Virtual Machine Required in SDLC?

- Software Testing and QA –
  - Safe Testing: VMs allow testers to run applications in isolated environments, ensuring that bugs and crashes do not affect the main system.
  - Snapshot and Rollback: Testers can take snapshots of a VM before testing. If a test fails, they can restore the snapshot and retry without reinstalling everything.

Why is a Virtual Machine Required in SDLC?

- Software Testing and QA –
  - Parallel Testing: Multiple VMs can run different versions of an application simultaneously to test compatibility.
  - Example: A QA engineer runs a web application in different VMs (Windows, macOS, Linux) to ensure cross-platform compatibility.

Why is a Virtual Machine Required in SDLC?

- Deployment and Production –
  - Cloud Integration: Many cloud services (AWS, Azure, GCP) use VMs to host applications in a scalable and secure manner.
  - Scalability: Businesses can create and destroy VMs based on demand, optimizing resource usage.

Why is a Virtual Machine Required in SDLC?

- Deployment and Production –
  - Disaster Recovery: VM snapshots help quickly recover from system failures.
  - Example: A company deploys its web application on AWS EC2 VMs to ensure high availability and scalability.

Why is a Virtual Machine Required in SDLC?

- Security and Isolation –
  - Sandboxing: Security teams use VMs to analyze malware and cyber threats without affecting real systems.
  - Isolated Environments: Sensitive applications can run in VMs to prevent unauthorized access.

Why is a Virtual Machine Required in SDLC?

- Security and Isolation –
  - Example: A cybersecurity team runs a suspected malicious file inside a VM to analyze its behavior safely.

Why is a Virtual Machine Required in SDLC?

- Legacy Software Support –
  - Many organizations still use old applications that require outdated operating systems.
  - VMs allow running legacy software without needing to maintain outdated hardware.

Why is a Virtual Machine Required in SDLC?

- Legacy Software Support –
  - Example: A company still using a Windows XP-based application can run it in a VM on a modern machine.

Challenges with Traditional Infrastructure (VM-Based Approach)

1. To host our apps, we need Infrastructure
   - Applications need hardware or cloud-based infrastructure to run.
2. We Use VM's/Cloud Computing to Set Up Infra
   - Virtual Machines (VMs) allow multiple applications to run on a single physical machine but require an OS for each instance. This adds OS cost, OS maintenance cost etc.

Challenges with Traditional Infrastructure (VM-Based Approach)

3. We Isolate Our Service in OS of VM

    - Each service gets its own VM to ensure isolation, avoiding conflicts between applications.

4. Because of Isolation, We End Up Setting Up Multiple VM's/Instances

    - Running separate VMs for different services leads to increased resource consumption.

Challenges with Traditional Infrastructure (VM-Based Approach)

5. VM's/Instances Will Be Over-Provisioned

   - To ensure performance and handle peak loads, we allocate more resources than needed, leading to inefficiency.
   - Also, VM itself is Expensive and if has OS then that add cost of OS Licensing, OS Nurturing, time to boot.

Challenges with Traditional Infrastructure (VM-Based Approach)

6. Results in High CapEx and OpEx

   - Capital Expenditure (CapEx): High initial costs for servers and infrastructure.
   - Operational Expenditure (OpEx): Ongoing costs for maintenance, power, and scaling resources.

So, how to isolate without OS?

Imagine Multiple Services running in same OS but isolated…….

Containers

2

Understanding Images and Containers

What are Containers?

- Container: A running instance of an image that encapsulates an application and its dependencies.

OR

- A container is a runtime instance of an image. When you run an image, it becomes a container. Containers are isolated environments that run applications with their own filesystem, networking, and processes, but share the host system's kernel.

What is Image?

- Image: A lightweight, stand-alone, executable software package that includes everything needed to run a piece of software, including code, runtime, system tools, libraries, and settings.
- An image for a Node.js application includes the Node.js runtime and app code.
- When you run *docker run my-node-app*, it creates a container from the image.

Understanding Images and Containers

- Images are like templates for creating containers or they are the building blocks of containers.
- Images are created using a Dockerfile, which defines the steps to build the image.
- Containers are instances of images that run in isolated environment, or they are the running instances of images.

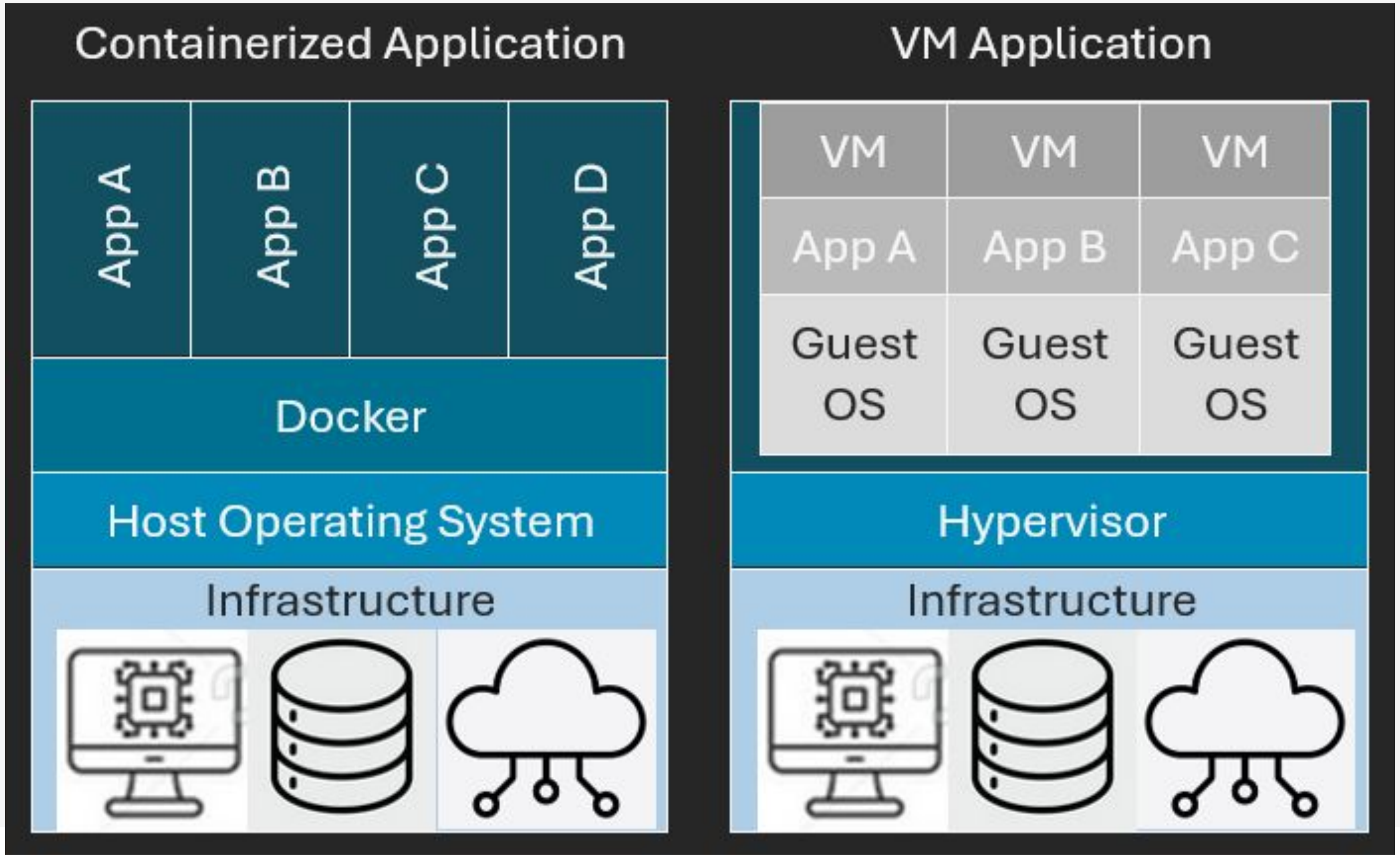
Understanding Images and Containers

- Containers are lightweight & portable, allowing applications to run consistently across different environments.
- A Python image (python:3.9) can be used to create multiple containers, each running a separate Python script.
- An image for a web server (e.g., Nginx) can be used to create multiple containers, each running an instance of the Nginx server.

Benefits of Containers Over VMs

VM-Based Approach	**Container-Based Approach (Docker)**
Each VM has a full OS, consuming resources.	Containers share the host OS, reducing overhead.
Boot time is slow (minutes).	Containers start in seconds.
Requires OS maintenance & licensing.	No separate OS, reducing costs.
High resource consumption per VM.	Containers are lightweight and efficient.
Heavy and bulky to move VMs.	Containers are portable and small in size.
Expensive (High CapEx & OpEx).	Cost-effective by optimizing infrastructure.

Benefits of Containers Over VMs



What is Containerization?

- Containerization is a lightweight alternative to full machine virtualization. It involves encapsulating an application and its dependencies into a container that can run on any system with a compatible container runtime (e.g., Docker).
- OR Containerization is the process of packaging an application and its dependencies into a container, ensuring it runs consistently across different environments.

What is Containerization?

- It provides isolation, portability, and consistency across different environments.
- Example: A developer creates a containerized web app that runs the same way on a laptop, server, or cloud.
- Example: A Python application containerized with its dependencies (e.g., Flask, NumPy) can run on any system with Docker installed, without worrying about compatibility issues.

What is Docker?

- Docker is a platform that enables developers to build, package, and deploy applications in containers.
- OR Docker is an open-source platform that automates the deployment, scaling, and management of applications using containerization.
- It provides tools to build, ship, and run containers efficiently.

What is Docker?

- Thus, 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.

What is Docker?

- 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.

- Example: Docker allows developers to package a Node.js application into a container and deploy it to any server running Docker.

3

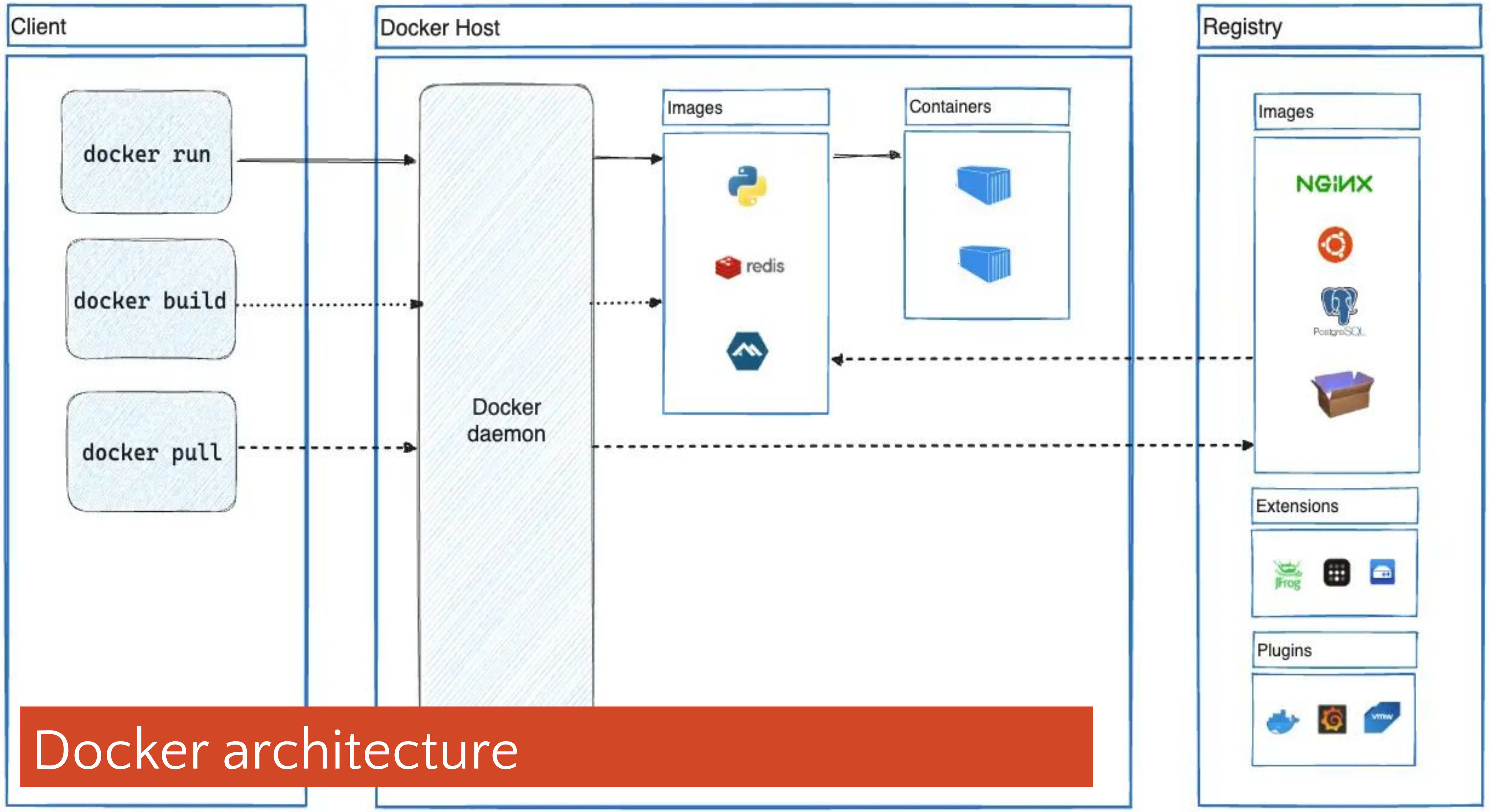
Docker Architecture

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.

Docker architecture

- 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

Docker architecture

- 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.

Docker architecture

- The Docker client
  - The Docker client (docker) is the primary way that many Docker users interact with Docker.
  - It is a command-line interface (CLI) or tool for user interaction with the Docker Daemon.

Docker architecture

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

Docker architecture

- The 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 architecture

- The Docker Desktop
  - Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

Docker architecture

- 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.

Docker architecture

- Docker Registries
  - 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 architecture

- Docker Objects
  - These are Images, containers, networks, and volumes.
  - When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects.

Docker architecture

- Docker 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.

Docker architecture

- Docker Objects – Images
    - 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.

Docker architecture

- Docker Objects – Images
  - 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 makes images so lightweight, small, and fast, when compared to other virtualization technologies.

Docker architecture

- Docker Objects – 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.

Docker architecture

- Docker Objects – Containers
  - 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.

Docker architecture

- Docker Objects – Containers
  - 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.

4

Docker Components

Docker Components

- Docker CLI – Command-line interface to interact with Docker.
- Docker Daemon – Background service managing containers.
- Dockerfile – A script/text file with instructions to build an image automatically.
- Docker Compose – Tool to define and run multi-container applications.
- Docker Hub – A cloud-based repository for sharing Docker images.

Docker Components

- Images: The read-only templates used to create containers.
- Containers: The running instances of images.
- Volumes: Persistent storage for containers.
- Networks: Isolated networks for communication between containers.

Benefits of Docker

- Portability: Containers can run on any system with Docker, ensuring consistency across environments.
  - Example: A containerized app developed on a Mac will run the same way on a Linux server.
- Isolation: Containers isolate applications and their dependencies, preventing conflicts.
  - Example: Two containers running different versions of Python won’t interfere with each other.

Benefits of Docker

- Efficiency: Containers share the host OS kernel, making them lightweight and fast.
  - Example: Running 10 containers on a single server uses fewer resources than running 10 virtual machines.
- Scalability: Containers can be easily scaled up or down using orchestration tools like Kubernetes.
  - Example: A web app can handle increased traffic by spinning up additional containers.

Benefits of Docker

- Consistency: Developers can avoid the "it works on my machine" problem.
  - Example: A team can share a Docker image to ensure everyone uses the same environment.
- Version Control: Docker images can be versioned and stored in registries.
  - Example: Roll back to a previous version of an app by running an older image.
- Speed: Containers start in seconds, compared to minutes for virtual machines.
  - Example: A developer can quickly test changes by restarting a container.

5

Docker Commands

Docker Commands

- build Command –
  - Docker build command is used to build an image using a Dockerfile.
  - Basic build command is as follows which will execute the Dockerfile in the project directory and do whatever instructions from the current directory files.
  - `docker build .`

Docker Commands

- build Command –
  - If a directory have multiple docker files then, which file is used for build process will be specified by "–f" flag followed by file name. e.g.

```
docker build -f Dockerfile.dev .
```

Docker Commands

- build Command –
  - The output image file have a name which consists of alphanumeric characters and in all the cases it is hard to remember.
  - Then there should be way to name the image which is tagging.

Docker Commands

- build Command –
  - There is a convention to tag an image as follows –

```
<docker_username>/<application_name>:version
```

  - Here, the version is optional to add and if not defined it takes as the latest.
  - Example – `docker build -t dockeruser/myapp:beta .`

Docker Commands

- run Command –
  - After building the image there are several options we have to add when we running an image and basic command is as follows –

```
Docker run <image>
```

  - Image can be identified from the id or form the name if we tagged it.

Docker Commands

- run Command –
  - When we run an image, it will create an instance of an image which is known as a container.
  - Container runs and terminate when the process is completed but it remained even after termination.

Docker Commands

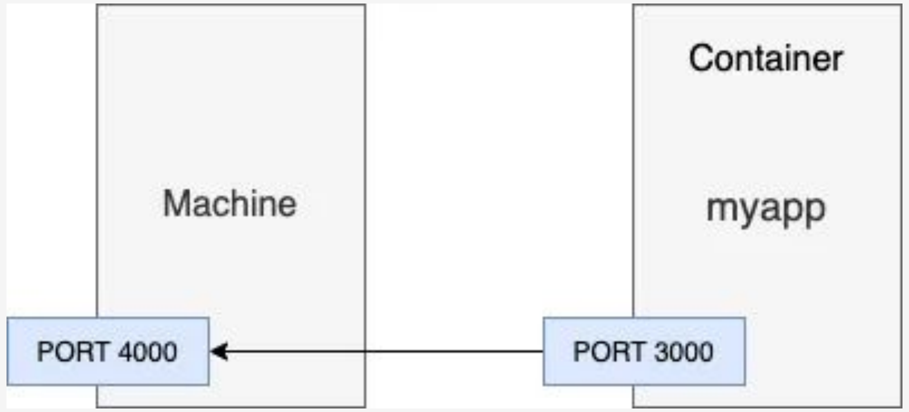
- run Command –
  - We can remove the container after terminate using “–rm” flag.

```
docker run -rm <image>
```

Docker Commands

- run Command –
  - If some application run in a container exposing a port, we must map those ports to local ports to access as follows using "-p" flag.

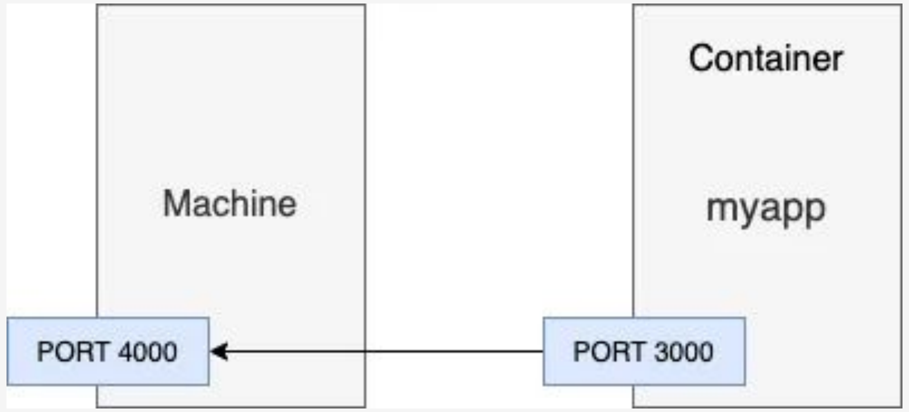
```
docker run -p <local_machine_port>:<container_port>
```



Docker Commands

- run Command –
  - If some application run in a container exposing a port, we must map those ports to local ports to access as follows using "-p" flag.

```
docker run -p <local_machine_port>:<container_port>
```



Docker Commands

- For more about various docker command use –
    - https://www.cherryservers.com/blog/docker-commands-cheat-sheet

The "dockerfile"

- A "dockerfile" consists of the following details:
- **FROM:** This parameter enables you to specify the base image that will be used to build the container.

  For example, if you are building a Node.js application your base image can be `node:19-alpine3.1`. It is very important to ensure that you specify the image version to prevent compatibility issues.

The "dockerfile"

- **RUN:** This command is used to install dependencies that are needed by your application in your container as specified by the package.js file.

- **COPY:** This command replicates files from the current directory on your local machine into the Docker image.

  You set the current directory using the `WORKDIR` field. All commands will be executed from the directory specified by the `WORKDIR` field.

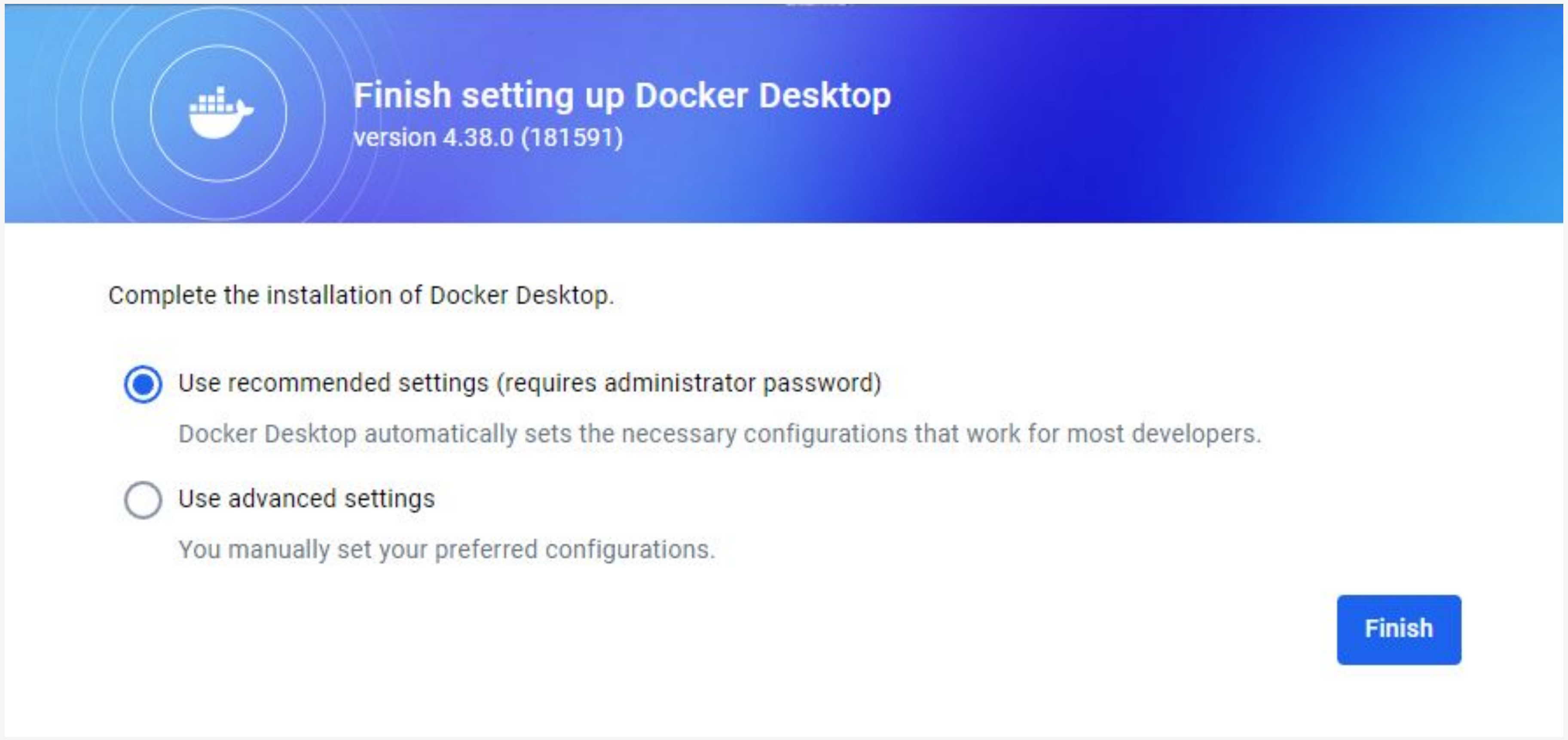
The "dockerfile"

- **CMD:** This is short for command. This parameter is used to state the command that will be run when the container starts running.
- **ENV:** This is used for setting environment variables.

5

Docker Installation

Benefits of Docker



Finish setting up Docker Desktop

version 4.38.0 (181591)

Complete the installation of Docker Desktop.

[✓ Use recommended settings (requires administrator password)]

Docker Desktop automatically sets the necessary configurations that work for most developers.

Use advanced settings

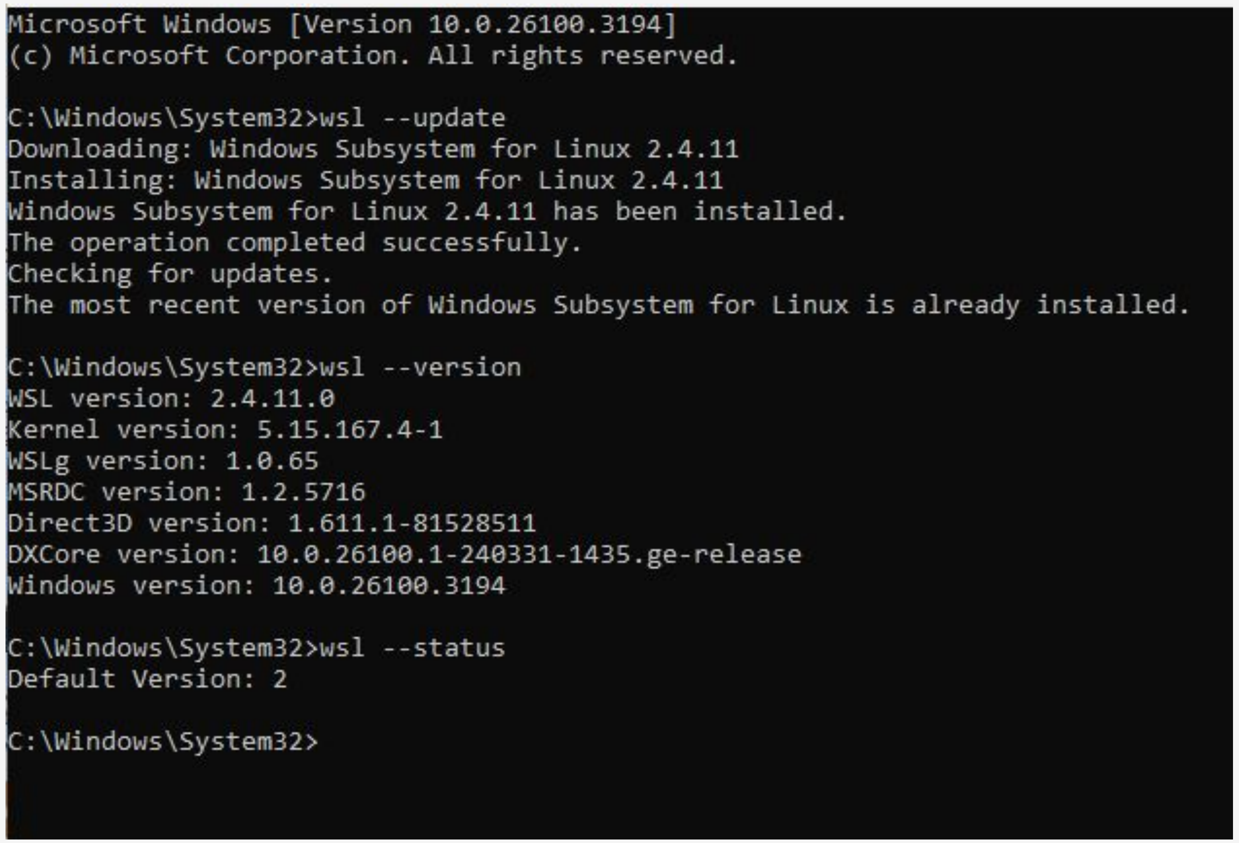
You manually set your preferred configurations.

Benefits of Docker

- If you see following error –

  wsl update failed: update failed: updating wsl: exit code: 4294967295: running WSL command wsl.exe C:\WINDOWS\System32\wsl.exe --update --web-download

- Follow the steps shown in command prompt.



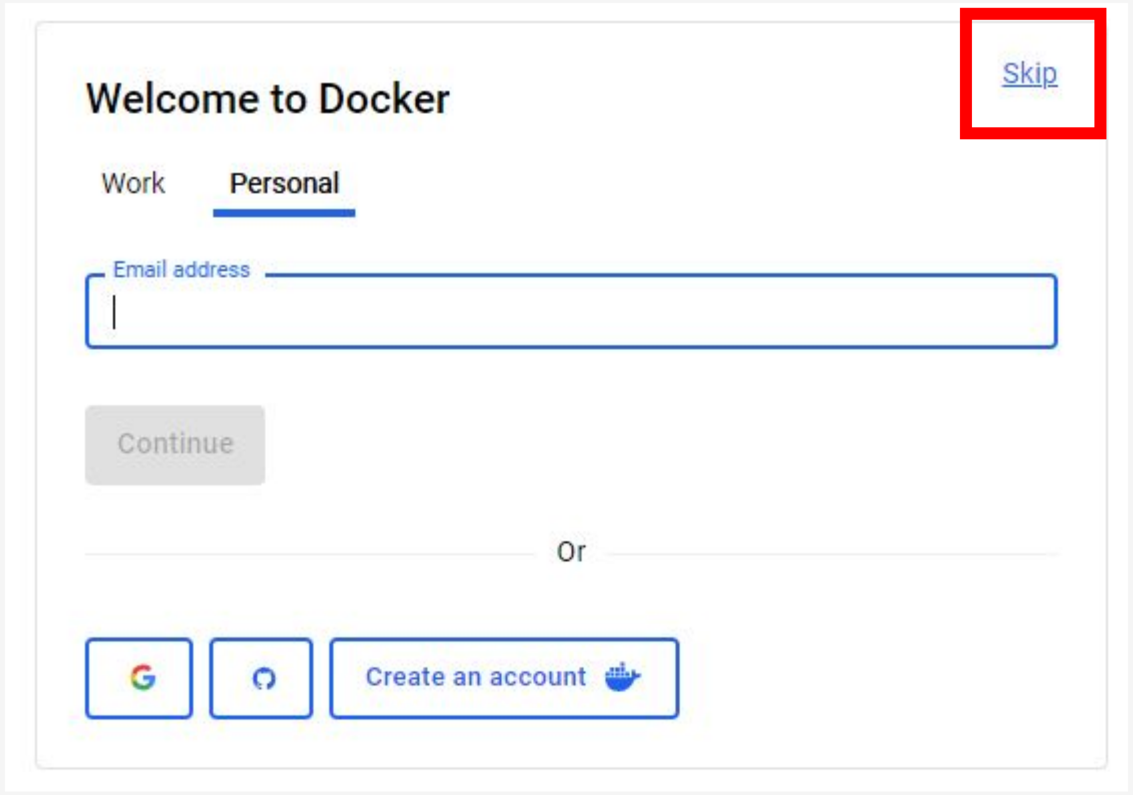
```
Microsoft Windows [Version 10.0.26100.3194]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\System32>wsl --update
Downloading: Windows Subsystem for Linux 2.4.11
Installing: Windows Subsystem for Linux 2.4.11
Windows Subsystem for Linux 2.4.11 has been installed.
The operation completed successfully.
Checking for updates.
The most recent version of Windows Subsystem for Linux is already installed.

C:\Windows\System32>wsl --version
WSL version: 2.4.11.0
Kernel version: 5.15.167.4-1
WSLg version: 1.0.65
MSRDC version: 1.2.5716
Direct3D version: 1.611.1-81528511
DXCore version: 10.0.26100.1-240331-1435.ge-release
Windows version: 10.0.26100.3194

C:\Windows\System32>wsl --status
Default Version: 2

C:\Windows\System32>
```

Benefits of Docker

- If you see following error –

  wsl update failed: update failed: updating wsl: exit code: 4294967295: running WSL command wsl.exe C:\WINDOWS\System32\wsl.exe --update --web-download

- Follow the steps shown in command prompt.



6

Build, deploy and manage web application on Docker Engine

6.1

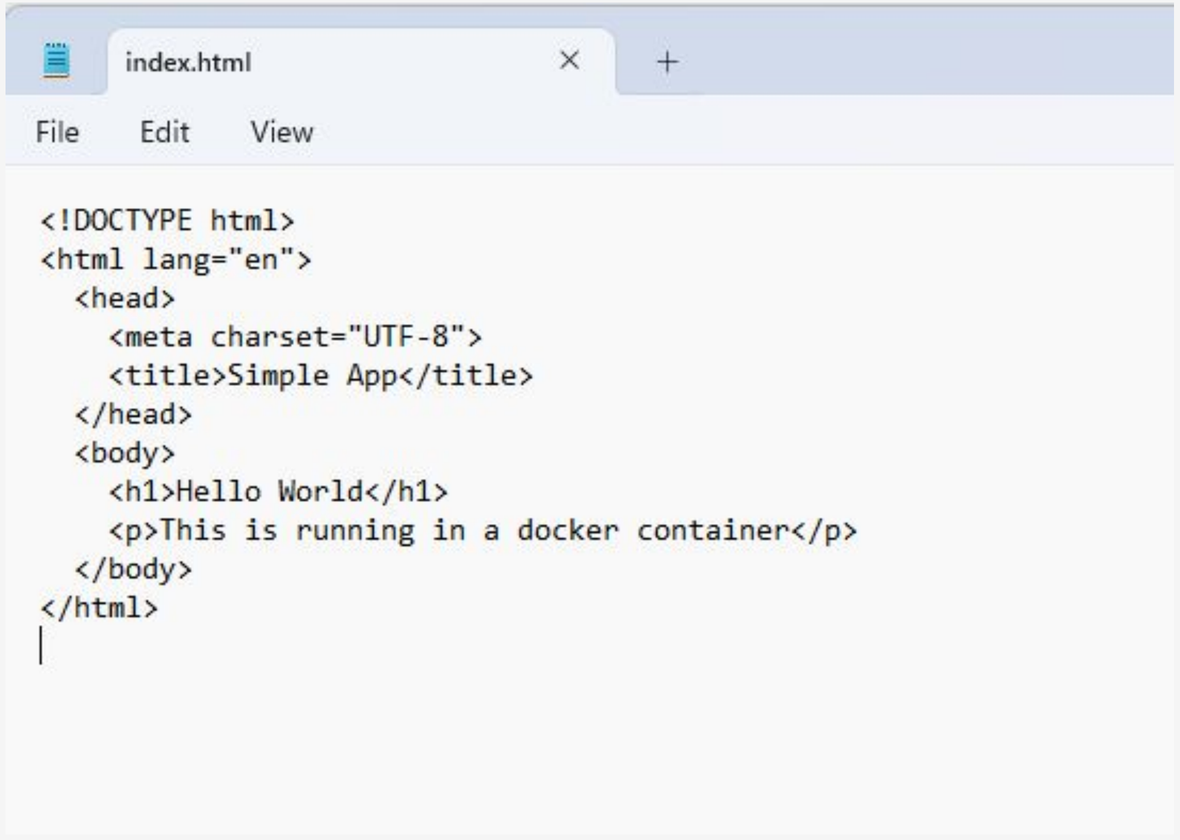
Dealing with static web page

Build, deploy and manage web application on Docker Engine

- Create a working directory and index.html in it.
- Select a base image and write a "dockerfile" file
- Build the image using the Docker build command
- Verify and Run the Docker image
- Access the application

Create a working directory and index.html in it.

- Create a directory to use for the demo application and navigate to that directory.
- Create a file called "index.html" in the directory and add the content shown in it.



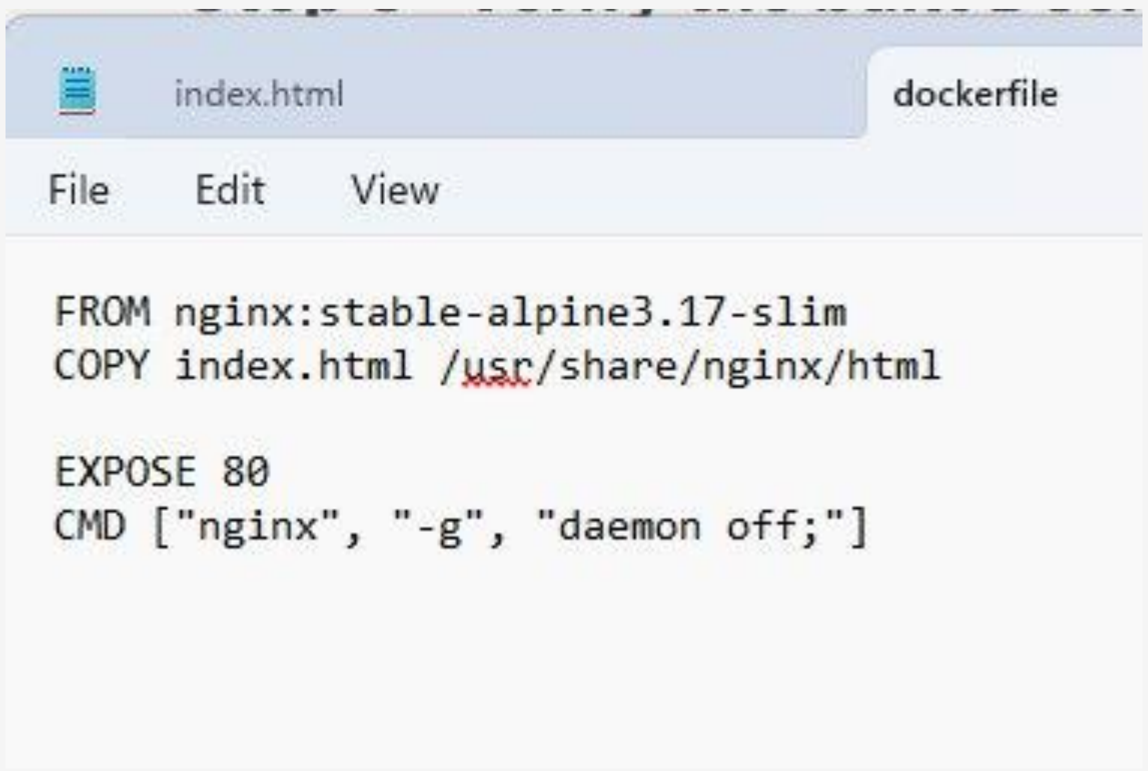
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Simple App</title>
  </head>
  <body>
    <h1>Hello World</h1>
    <p>This is running in a docker container</p>
  </body>
</html>
```

Select a base image and write a "dockerfile" file

- Next, select a suitable base image from Docker Hub or a local repository.
- The base image forms the foundation of your custom image and contains the operating system and essential dependencies.
- Almost every single image for Docker is based on another image.
- For this demonstration, you'd be using nginx:stable-alpine3.17-slim as the base image.

Select a base image and write a "dockerfile" file

- Now create a file named "Dockerfile".
- This file will define the build instructions for your image.
- By default, when you run the docker build command, docker searches for the Dockerfile.



```
FROM nginx:stable-alpine3.17-slim
COPY index.html /usr/share/nginx/html

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Select a base image and write a "dockerfile" file

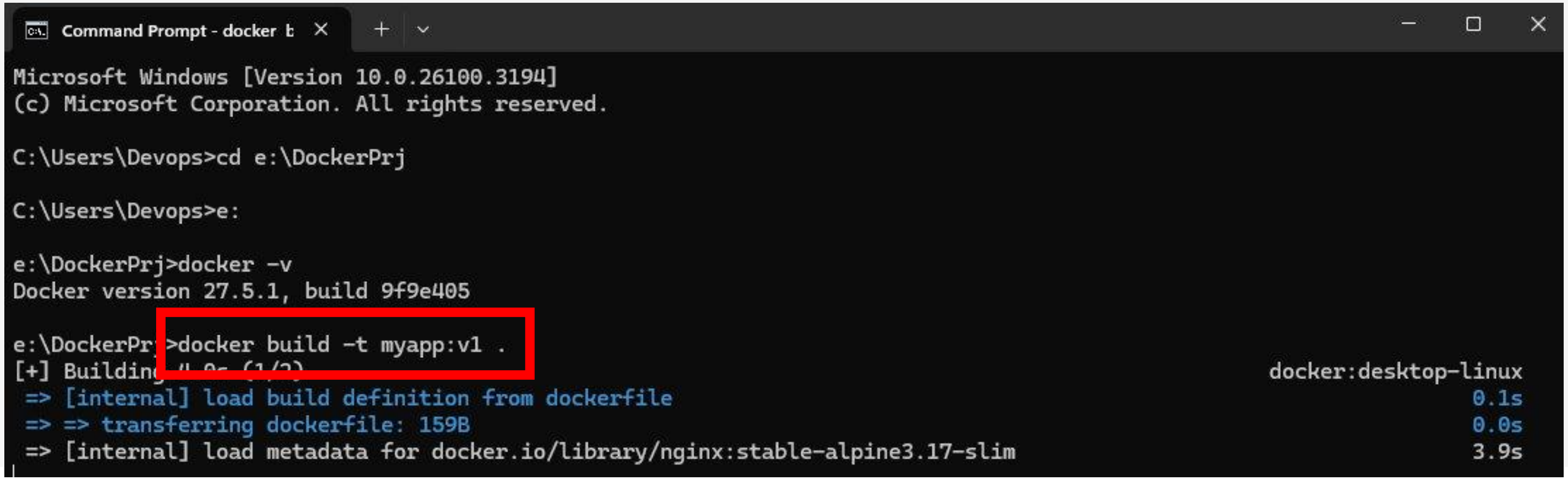
- The **FROM** instruction initializes a new build stage and sets the base image for subsequent instructions.
- The **COPY** instruction copies files or directories (usually the source code) from the source to the specified location inside the image.
- It copies the file to the "/usr/share/nginx/html" directory, which is the default location for serving static files in the Nginx web server.

Select a base image and write a "dockerfile" file

- The main purpose of the **CMD** instruction is to provide defaults for executing containers.
- The instructions defined in Dockerfiles differ based on the kind of image you're trying to build.

Build the image using the Docker build command

- To build your container, make sure you're in the folder where you have your Dockerfile and run the following command in the terminal:

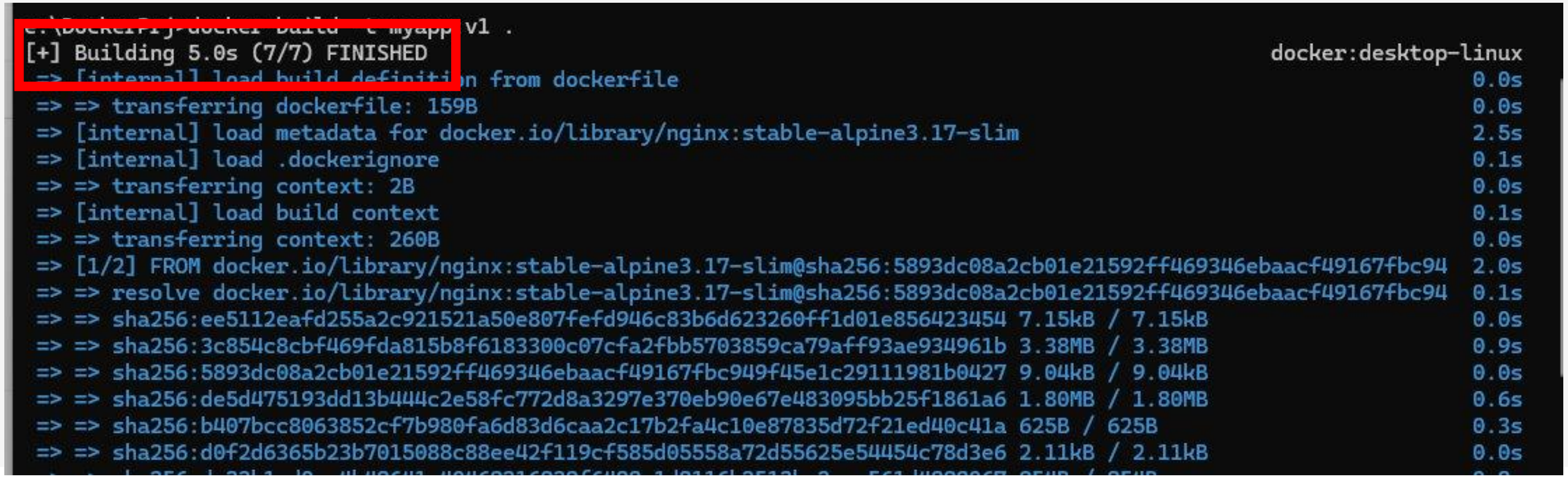


Command Prompt - docker b

```
Microsoft Windows [Version 10.0.26100.3194]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Devops>cd e:\DockerPrj

C:\Users\Devops>e:

e:\DockerPrj>docker -v
Docker version 27.5.1, build 9f9e405

e:\DockerPrj>docker build -t myapp:v1 .
[+] Building                                                              docker:desktop-linux
 => [internal] load build definition from dockerfile                                     0.1s
 => => transferring dockerfile: 159B                                                     0.0s
 => [internal] load metadata for docker.io/library/nginx:stable-alpine3.17-slim          3.9s
```

Build the image using the Docker build command

- You should see the build process start and an output indicating that it has finished when it is done.

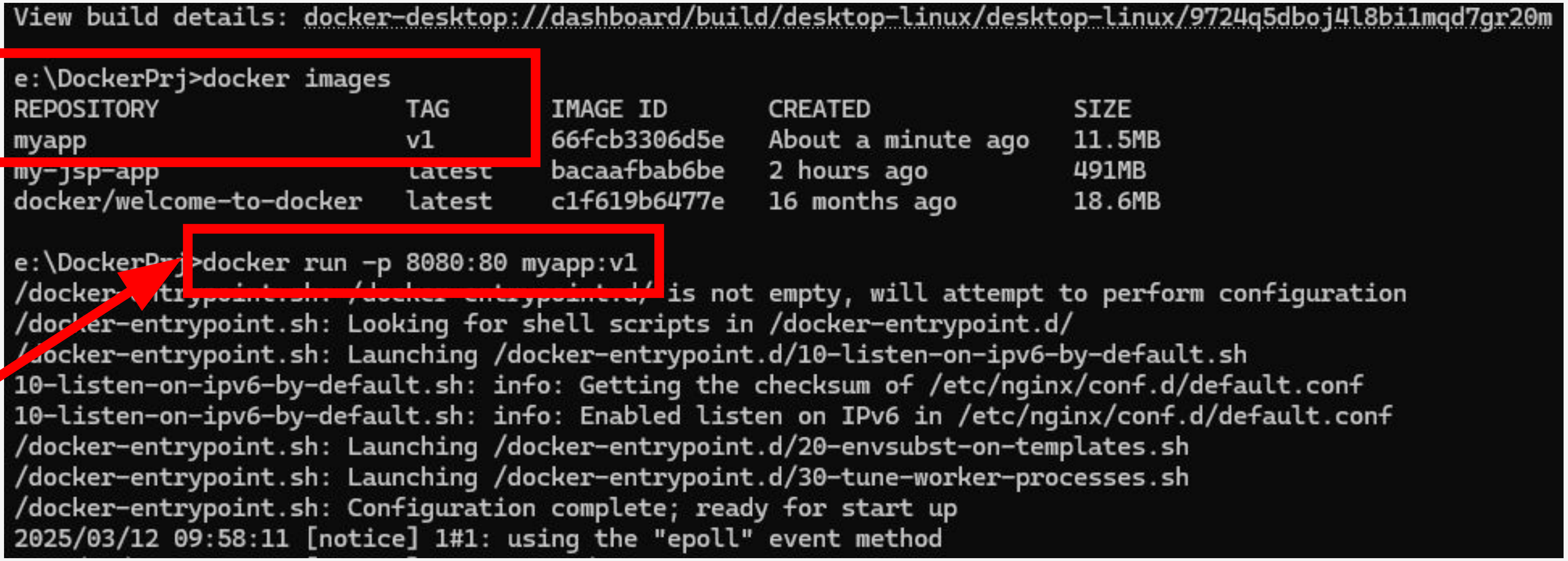


```
c:\DockerPrj>docker build -t myapp:v1 .
[+] Building 5.0s (7/7) FINISHED                                                         docker:desktop-linux
 => [internal] load build definition from dockerfile                                                    0.0s
 => => transferring dockerfile: 159B                                                                    0.0s
 => [internal] load metadata for docker.io/library/nginx:stable-alpine3.17-slim                         2.5s
 => [internal] load .dockerignore                                                                       0.1s
 => => transferring context: 2B                                                                         0.0s
 => [internal] load build context                                                                       0.1s
 => => transferring context: 260B                                                                       0.0s
 => [1/2] FROM docker.io/library/nginx:stable-alpine3.17-slim@sha256:5893dc08a2cb01e21592ff469346ebaacf49167fbc94  2.0s
 => => resolve docker.io/library/nginx:stable-alpine3.17-slim@sha256:5893dc08a2cb01e21592ff469346ebaacf49167fbc94  0.1s
 => => sha256:ee5112eafd255a2c921521a50e807fefd946c83b6d623260ff1d01e856423454 7.15kB / 7.15kB           0.0s
 => => sha256:3c854c8cbf469fda815b8f6183300c07cfa2fbb5703859ca79aff93ae934961b 3.38MB / 3.38MB           0.9s
 => => sha256:5893dc08a2cb01e21592ff469346ebaacf49167fbc949f45e1c29111981b0427 9.04kB / 9.04kB           0.0s
 => => sha256:de5d475193dd13b444c2e58fc772d8a3297e370eb90e67e483095bb25f1861a6 1.80MB / 1.80MB           0.6s
 => => sha256:b407bcc8063852cf7b980fa6d83d6caa2c17b2fa4c10e87835d72f21ed40c41a 625B / 625B               0.3s
 => => sha256:d0f2d6365b23b7015088c88ee42f119cf585d05558a72d55625e54454c78d3e6 2.11kB / 2.11kB           0.0s
```

Verify and Run the Docker image

Verifying the Docker image.

Running the Docker image as a container

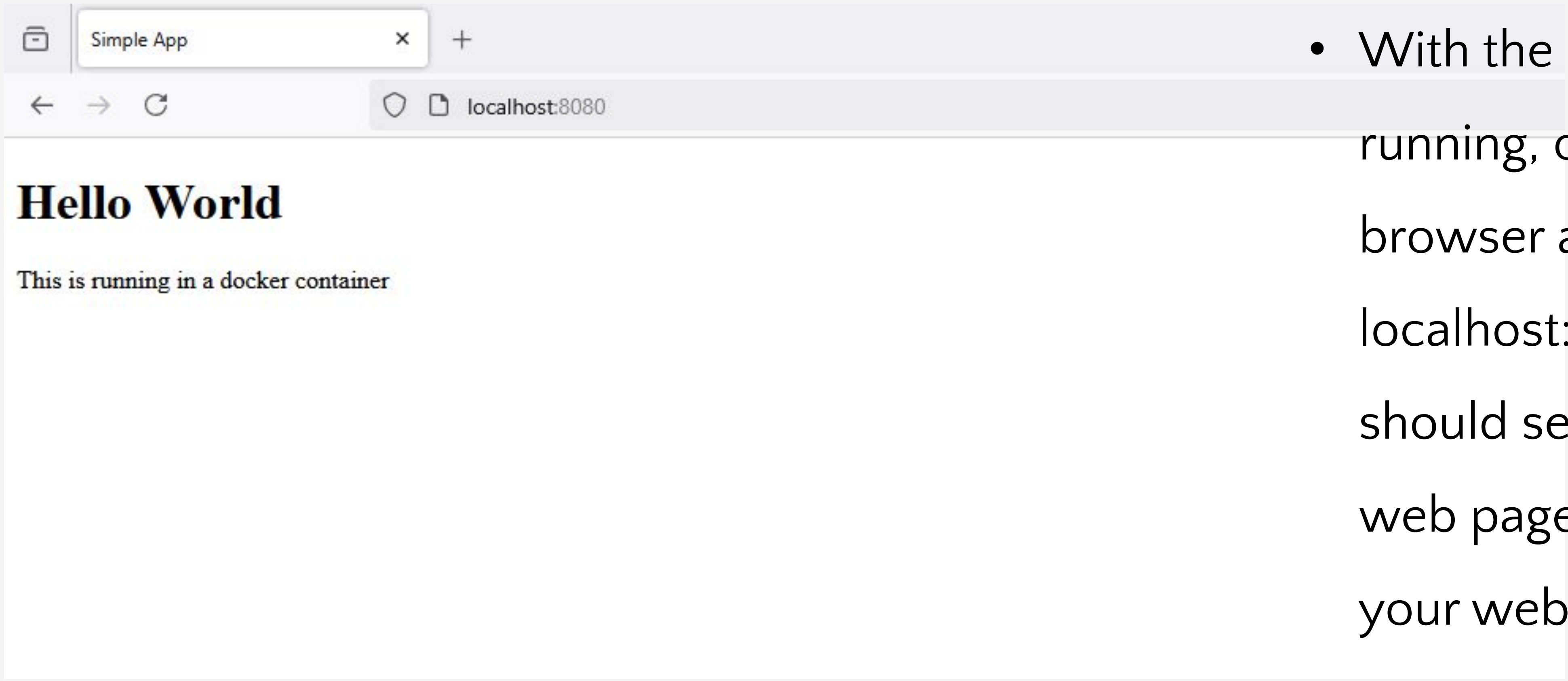


```
View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/9724q5dboj4l8bi1mqd7gr20m

e:\DockerPrj>docker images
REPOSITORY                TAG       IMAGE ID       CREATED              SIZE
myapp                     v1        66fcb3306d5e   About a minute ago   11.5MB
my-jsp-app                latest    bacaafbab6be   2 hours ago          491MB
docker/welcome-to-docker  latest    c1f619b6477e   16 months ago        18.6MB

e:\DockerPrj>docker run -p 8080:80 myapp:v1
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2025/03/12 09:58:11 [notice] 1#1: using the "epoll" event method
```

Verify and Run the Docker image

- The RUN command tells Docker to run the "myapp" container.
- The -p flag specifies the port mapping, which maps a port from the host machine to a port inside the container.
- Here, you are mapping port 8080 of the host machine to port 80 of the container.
- You can modify the host port as per your preference.

Access the application



Hello World

This is running in a docker container

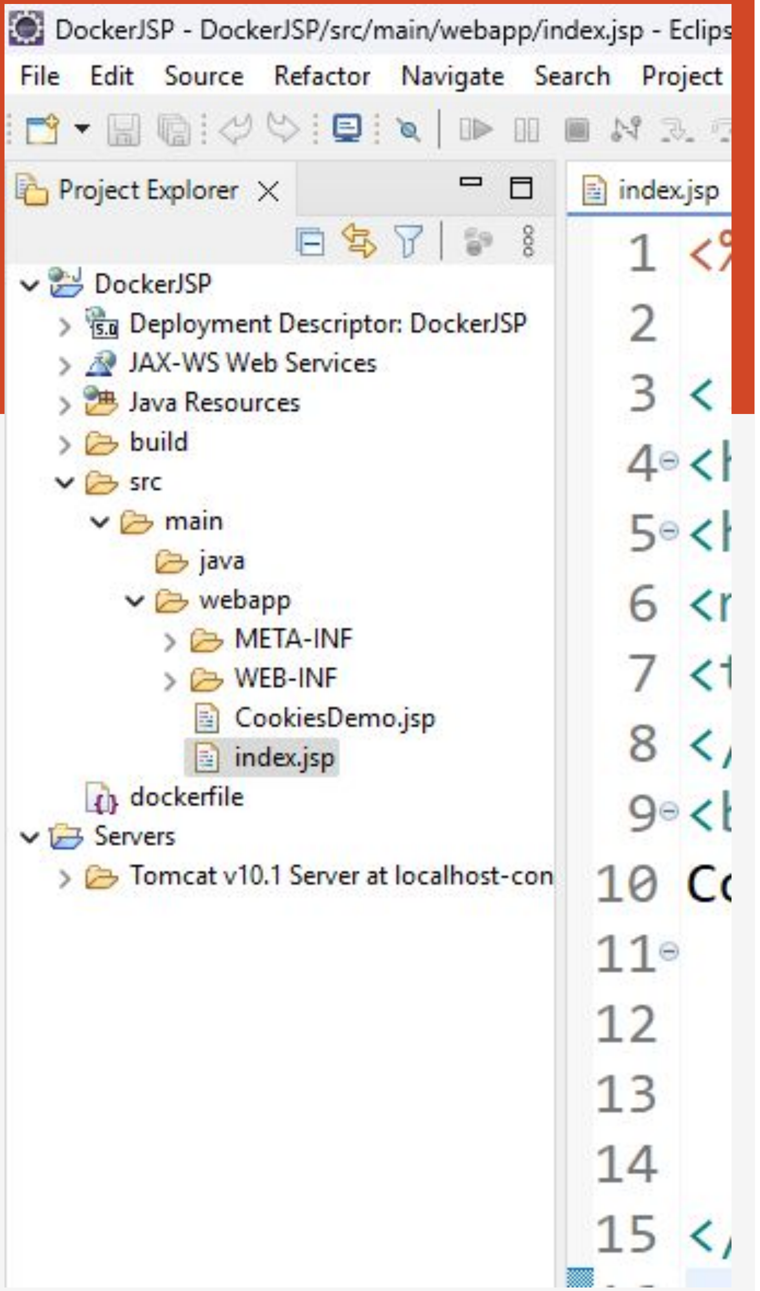
- With the container running, open a web browser and navigate to localhost:8080 and you should see the sample web page displayed on your web browser.

6.2

Dealing with JSP application

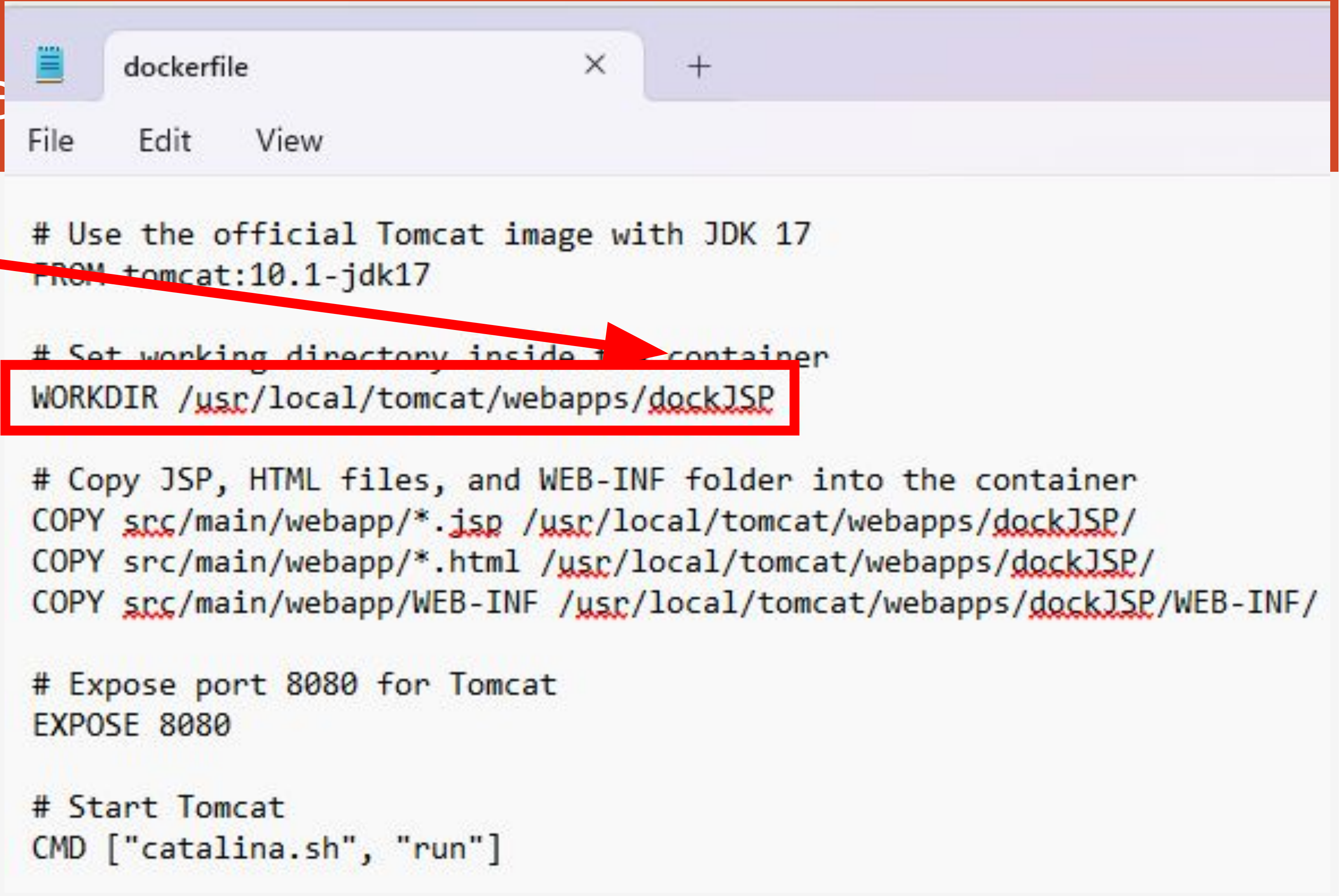
Creating JSP Project

- Create a JSP project with the help of Eclipse.
- The project structure is shown here.
- Create a "dockerfile" in the Project folder and add the commands shown on next slide.


DockerJSP - DockerJSP/src/main/webapp/index.jsp - Eclips
File Edit Source Refactor Navigate Search Project
Project Explorer
index.jsp
DockerJSP
Deployment Descriptor: DockerJSP
JAX-WS Web Services
Java Resources
build
src
main
java
webapp
META-INF
WEB-INF
CookiesDemo.jsp
index.jsp
dockerfile
Servers
Tomcat v10.1 Server at localhost-con

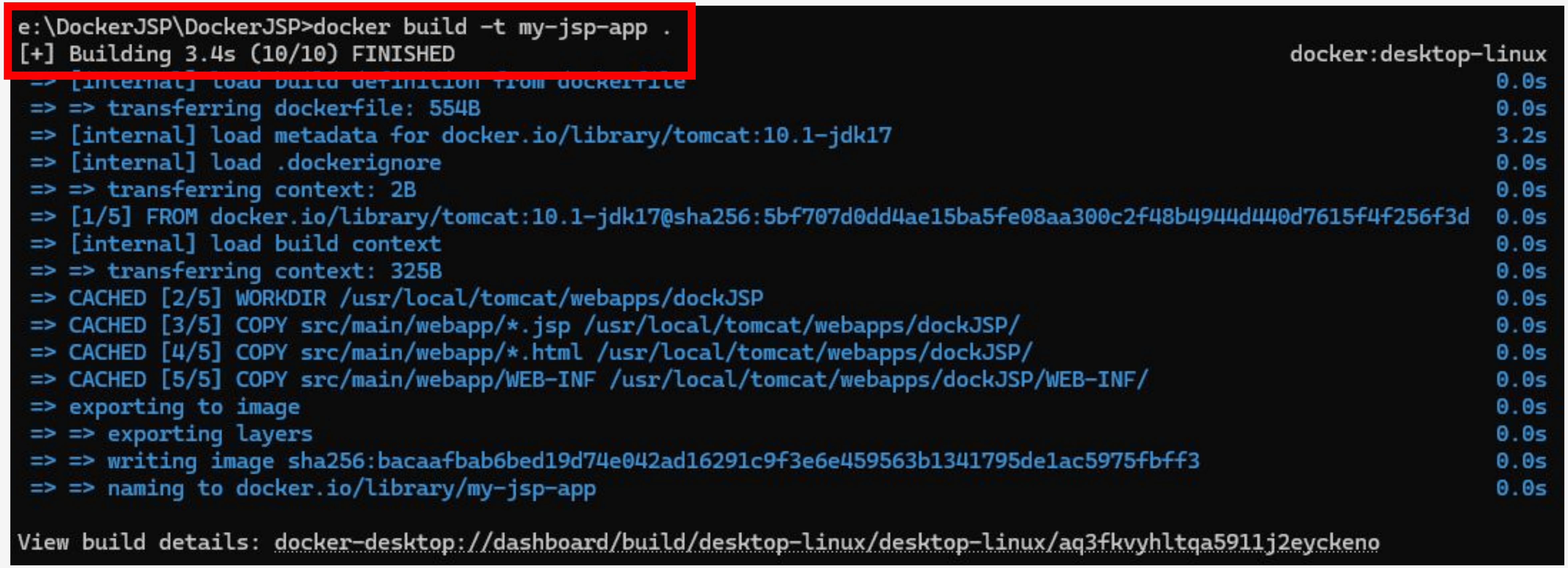
Creating dockerfile

The JSP project is copies in dockJSP directory of tomcat hence, while running the project in browser, use this app name.



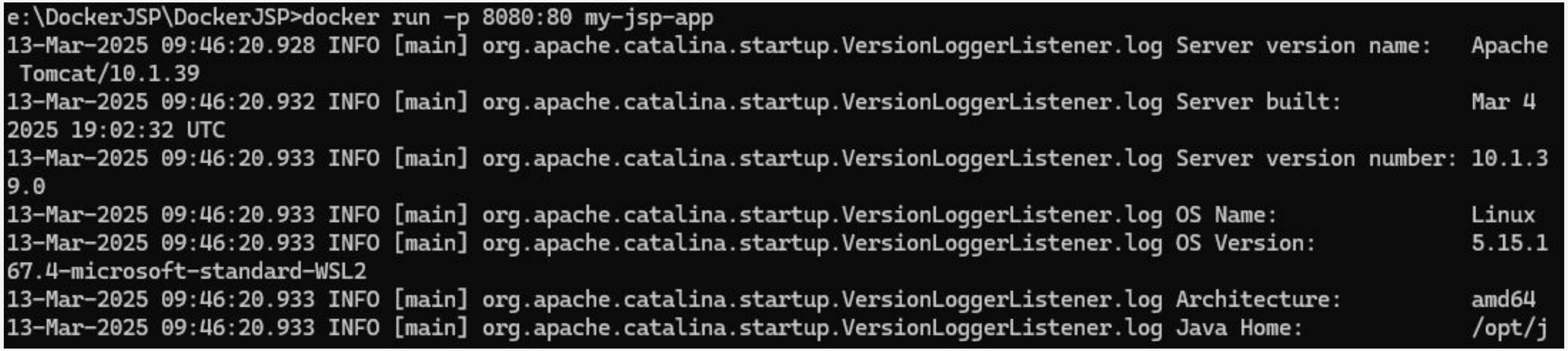
```
# Use the official Tomcat image with JDK 17
FROM tomcat:10.1-jdk17

# Set working directory inside the container
WORKDIR /usr/local/tomcat/webapps/dockJSP

# Copy JSP, HTML files, and WEB-INF folder into the container
COPY src/main/webapp/*.jsp /usr/local/tomcat/webapps/dockJSP/
COPY src/main/webapp/*.html /usr/local/tomcat/webapps/dockJSP/
COPY src/main/webapp/WEB-INF /usr/local/tomcat/webapps/dockJSP/WEB-INF/

# Expose port 8080 for Tomcat
EXPOSE 8080

# Start Tomcat
CMD ["catalina.sh", "run"]
```

Build the image using the Docker build command



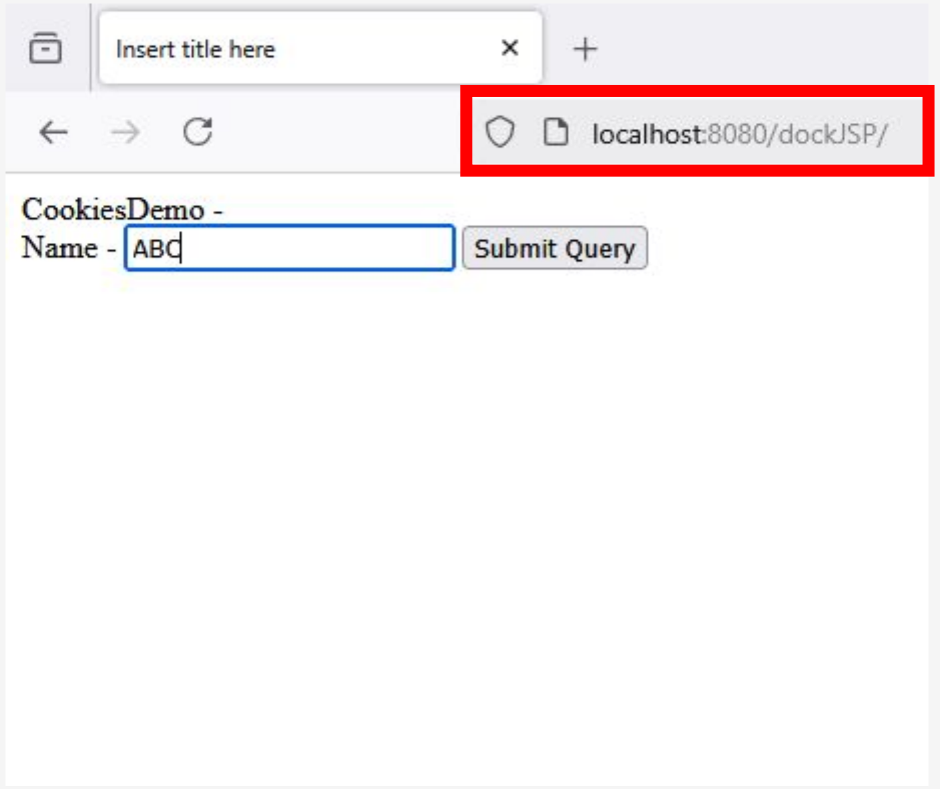
```
e:\DockerJSP\DockerJSP>docker build -t my-jsp-app .
[+] Building 3.4s (10/10) FINISHED                                                         docker:desktop-linux
 => [internal] load build definition from dockerfile                                                      0.0s
 => => transferring dockerfile: 554B                                                                      0.0s
 => [internal] load metadata for docker.io/library/tomcat:10.1-jdk17                                      3.2s
 => [internal] load .dockerignore                                                                         0.0s
 => => transferring context: 2B                                                                           0.0s
 => [1/5] FROM docker.io/library/tomcat:10.1-jdk17@sha256:5bf707d0dd4ae15ba5fe08aa300c2f48b4944d440d7615f4f256f3d 0.0s
 => [internal] load build context                                                                         0.0s
 => => transferring context: 325B                                                                         0.0s
 => CACHED [2/5] WORKDIR /usr/local/tomcat/webapps/dockJSP                                                0.0s
 => CACHED [3/5] COPY src/main/webapp/*.jsp /usr/local/tomcat/webapps/dockJSP/                            0.0s
 => CACHED [4/5] COPY src/main/webapp/*.html /usr/local/tomcat/webapps/dockJSP/                           0.0s
 => CACHED [5/5] COPY src/main/webapp/WEB-INF /usr/local/tomcat/webapps/dockJSP/WEB-INF/                  0.0s
 => exporting to image                                                                                    0.0s
 => => exporting layers                                                                                   0.0s
 => => writing image sha256:bacaafbab6bed19d74e042ad16291c9f3e6e459563b1341795de1ac5975fbff3               0.0s
 => => naming to docker.io/library/my-jsp-app                                                             0.0s

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/aq3fkvyhltqa5911j2eyckeno
```

Verify and Run the Docker image



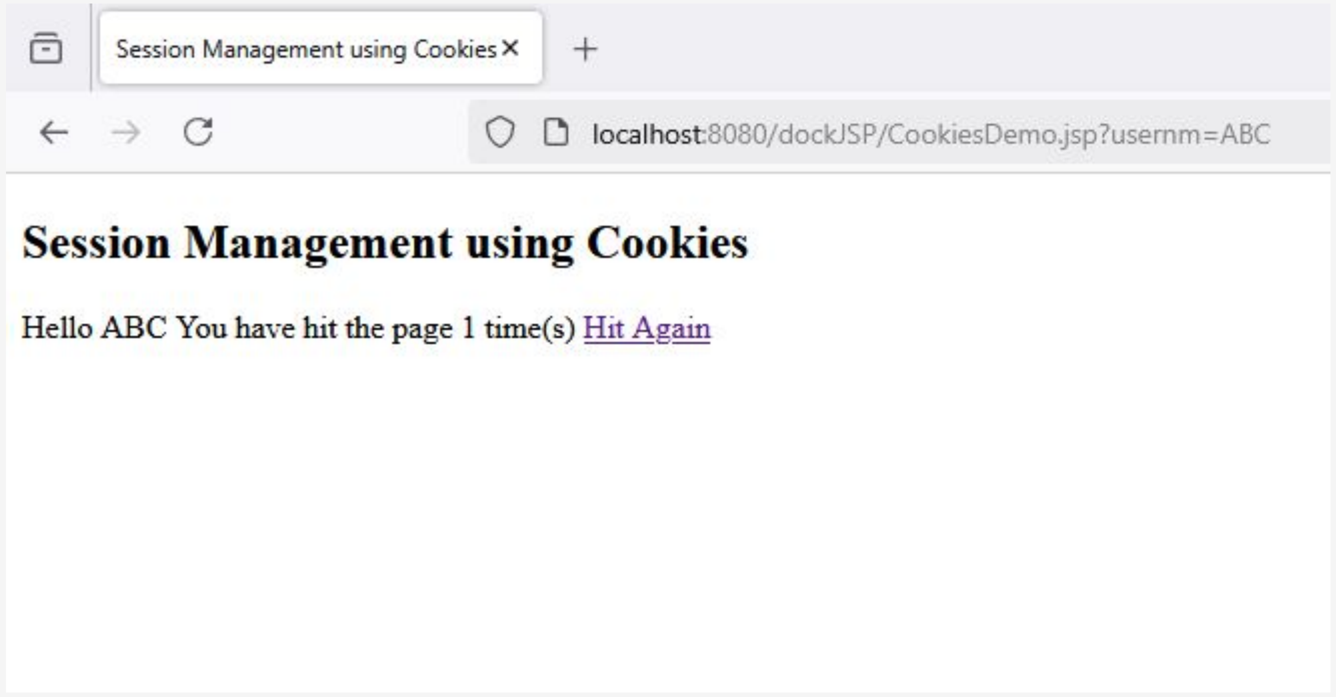
```
e:\DockerJSP\DockerJSP>docker run -p 8080:80 my-jsp-app
13-Mar-2025 09:46:20.928 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server version name:   Apache
 Tomcat/10.1.39
13-Mar-2025 09:46:20.932 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server built:          Mar 4
2025 19:02:32 UTC
13-Mar-2025 09:46:20.933 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server version number: 10.1.3
9.0
13-Mar-2025 09:46:20.933 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log OS Name:               Linux
13-Mar-2025 09:46:20.933 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log OS Version:            5.15.1
67.4-microsoft-standard-WSL2
13-Mar-2025 09:46:20.933 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Architecture:          amd64
13-Mar-2025 09:46:20.933 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Java Home:             /opt/j
```

Access the application


Insert title here

localhost:8080/dockJSP/

CookiesDemo -
Name - ABC


Session Management using Cookies

localhost:8080/dockJSP/CookiesDemo.jsp?usernm=ABC

Session Management using Cookies

Hello ABC You have hit the page 1 time(s) Hit Again

6.3

Jenkins Pipeline for Building & Deploying a Dockerized Java Application

Jenkins Pipeline for Building & Deploying a Dockerized Java Application

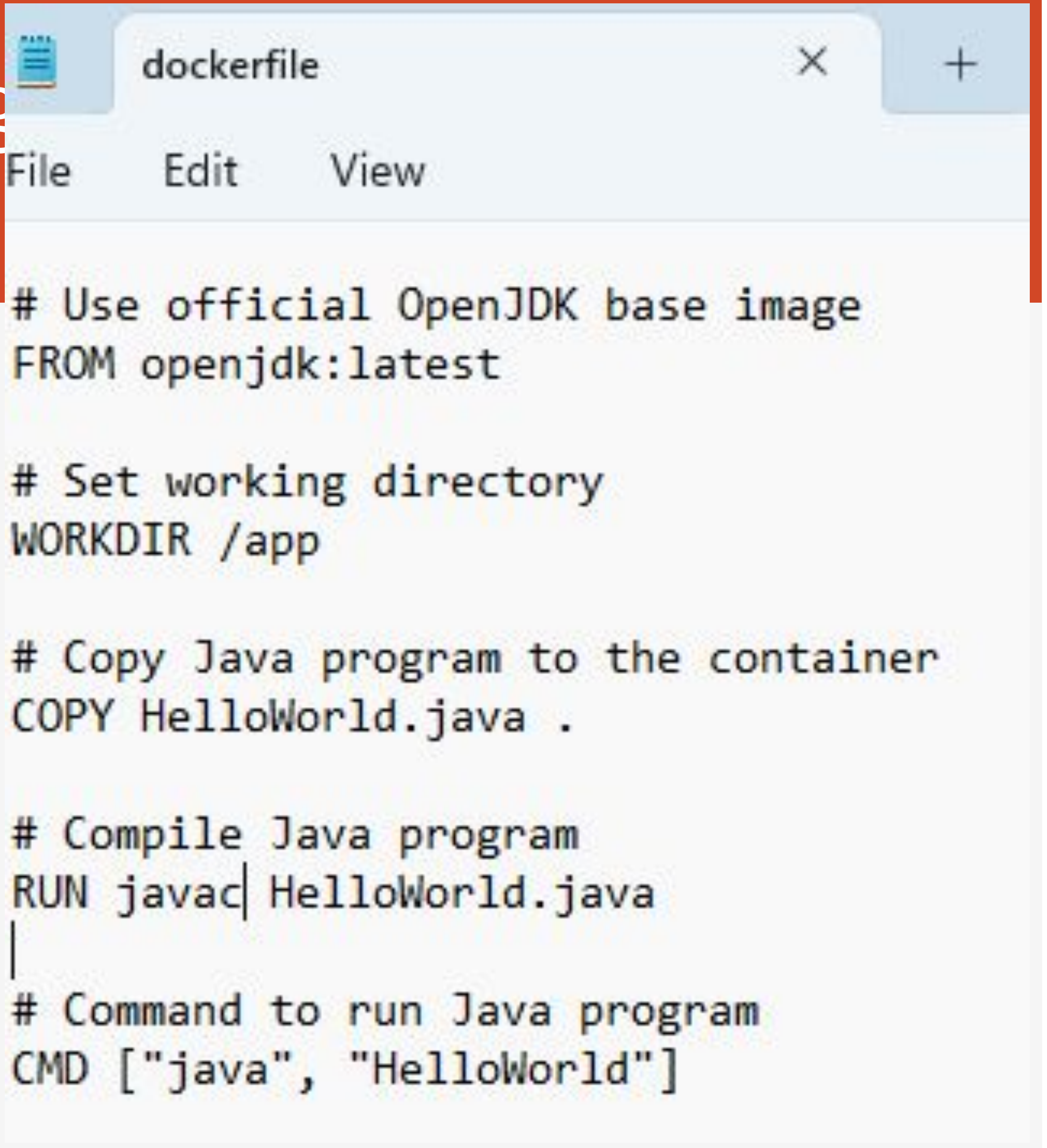
- Steps for the Jenkins Pipeline –
  - Create a working directory (like `E:\DockerJSP\exp2`)
  - Create a `HelloWorld.java` program in the working directory.
  - Create a `dockerfile` inside the working directory.
  - Create a `Jenkinsfile` inside the working directory.

Jenkins Pipeline for Building & Deploying a Dockerized Java Application

- Steps for the Jenkins Pipeline –
  - Add the working directory in the Git and push it to the remote repo.
  - Create a Jenkins pipeline and specify the Pipeline script for SCM.
  - Save and Build the pipeline to get the output.

Jenkins for Building & Deploying a Dockerized Java Application

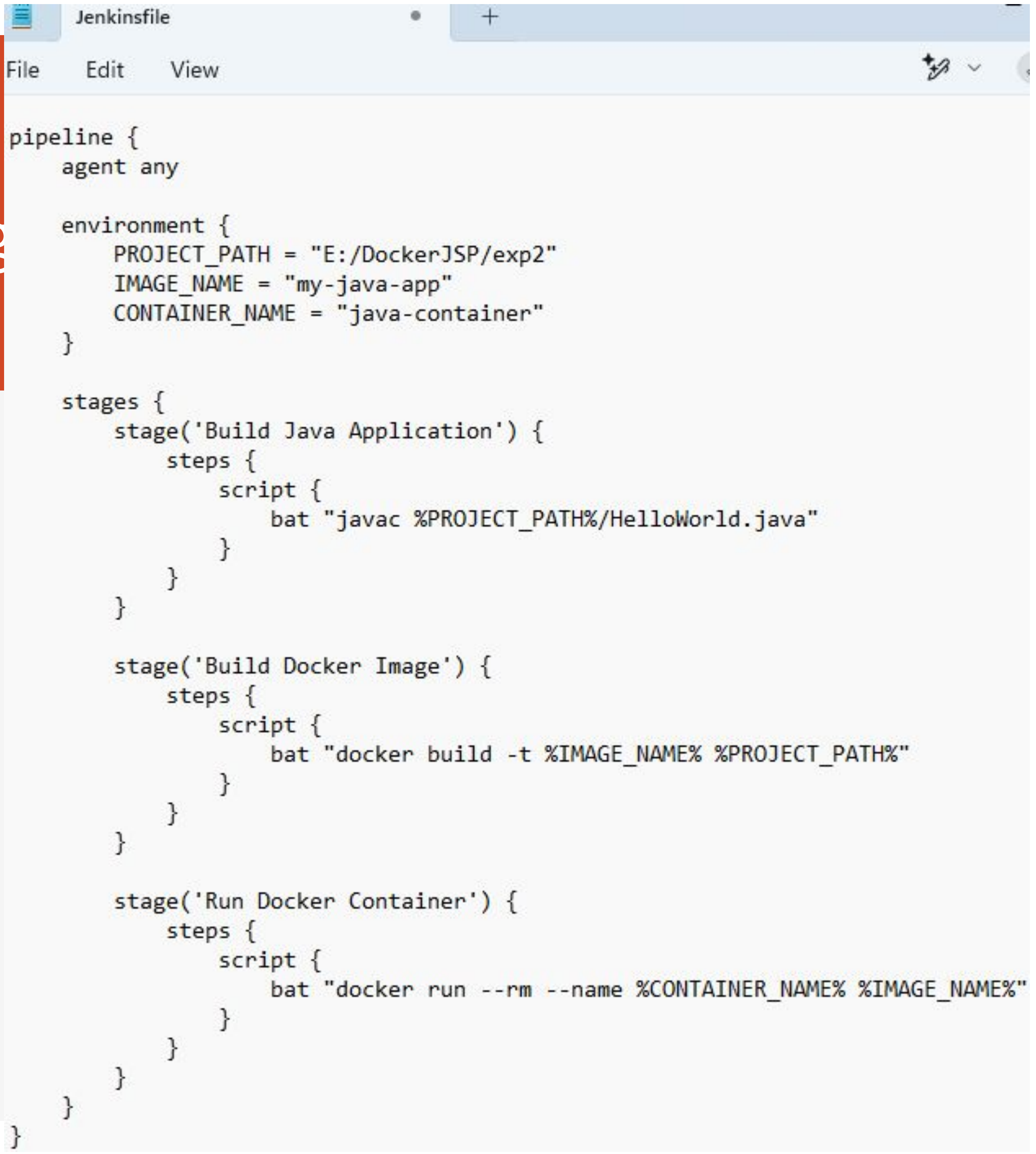
- Steps for the Jenkins Pipeline –
  - Create a `dockerfile` inside the working directory.



```
# Use official OpenJDK base image
FROM openjdk:latest

# Set working directory
WORKDIR /app

# Copy Java program to the container
COPY HelloWorld.java .

# Compile Java program
RUN javac HelloWorld.java

# Command to run Java program
CMD ["java", "HelloWorld"]
```

Jenkins for Building & Deploying a Dockerized Java Application

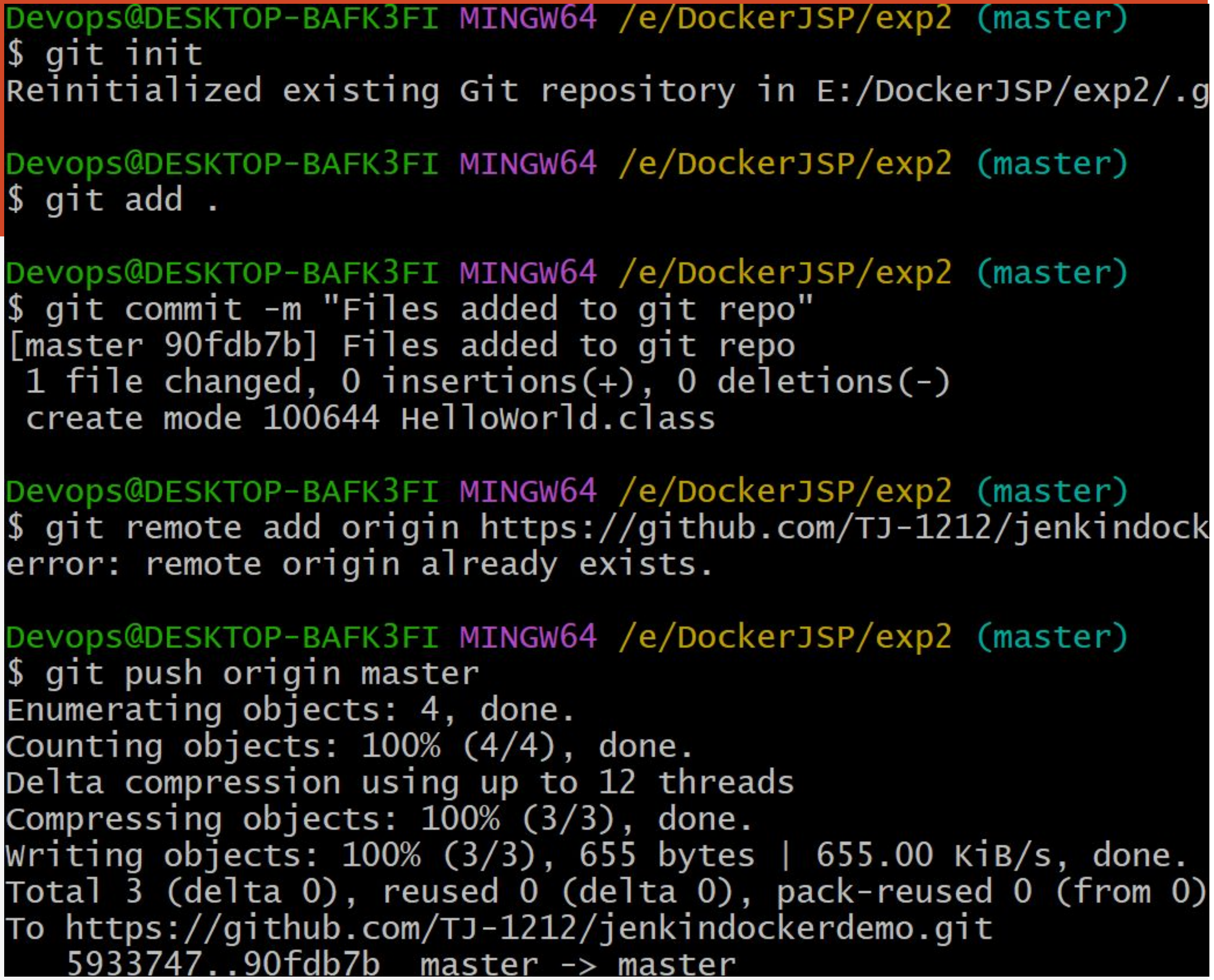
- Steps for the Jenkins Pipeline –
  - Create a `Jenkinsfile` inside the working directory.



```
pipeline {
    agent any

    environment {
        PROJECT_PATH = "E:/DockerJSP/exp2"
        IMAGE_NAME = "my-java-app"
        CONTAINER_NAME = "java-container"
    }

    stages {
        stage('Build Java Application') {
            steps {
                script {
                    bat "javac %PROJECT_PATH%/HelloWorld.java"
                }
            }
        }

        stage('Build Docker Image') {
            steps {
                script {
                    bat "docker build -t %IMAGE_NAME% %PROJECT_PATH%"
                }
            }
        }

        stage('Run Docker Container') {
            steps {
                script {
                    bat "docker run --rm --name %CONTAINER_NAME% %IMAGE_NAME%"
                }
            }
        }
    }
}
```

Jenkins for Building & Deploying a Dockerized Java Application

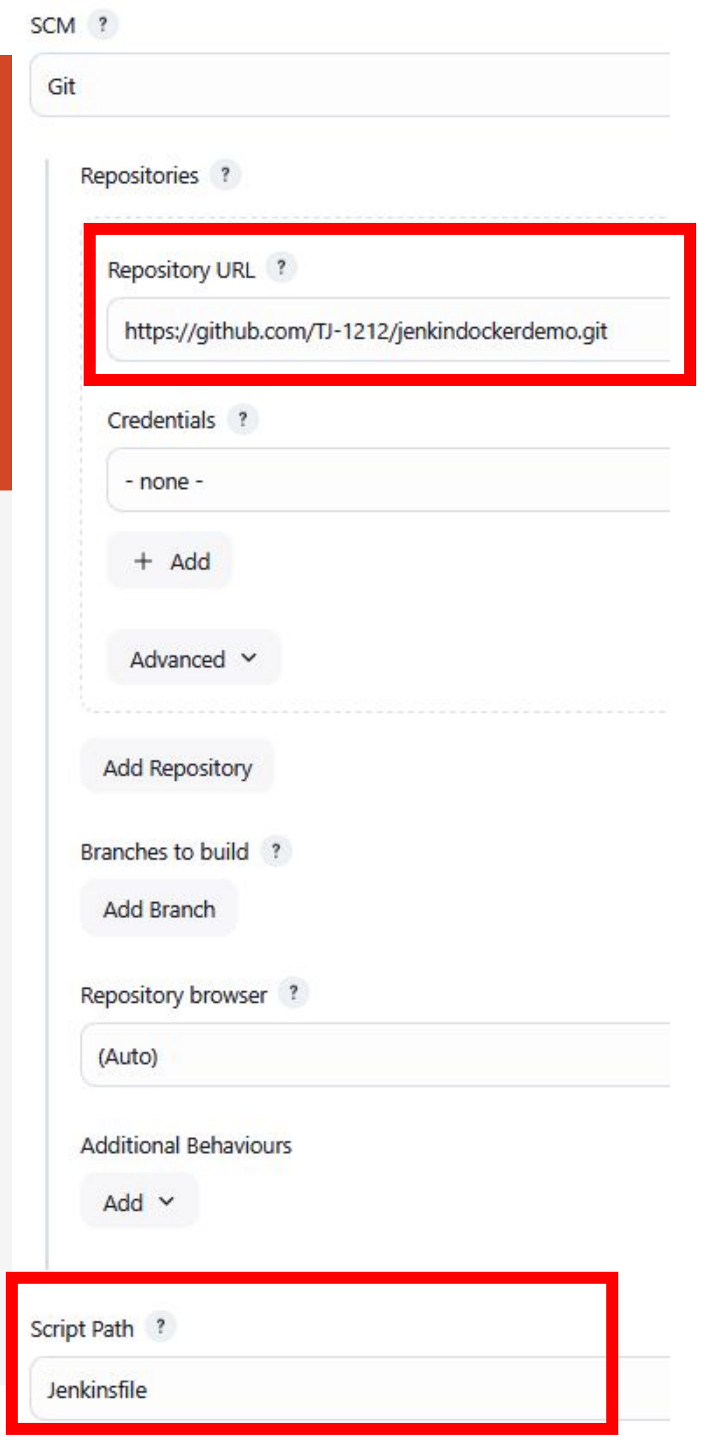
- Steps for the Jenkins Pipeline –
  - Add the working directory in the Git and push it to the remote repo.



```
Devops@DESKTOP-BAFK3FI MINGW64 /e/DockerJSP/exp2 (master)
$ git init
Reinitialized existing Git repository in E:/DockerJSP/exp2/.g

Devops@DESKTOP-BAFK3FI MINGW64 /e/DockerJSP/exp2 (master)
$ git add .

Devops@DESKTOP-BAFK3FI MINGW64 /e/DockerJSP/exp2 (master)
$ git commit -m "Files added to git repo"
[master 90fdb7b] Files added to git repo
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 HelloWorld.class

Devops@DESKTOP-BAFK3FI MINGW64 /e/DockerJSP/exp2 (master)
$ git remote add origin https://github.com/TJ-1212/jenkindock
error: remote origin already exists.

Devops@DESKTOP-BAFK3FI MINGW64 /e/DockerJSP/exp2 (master)
$ git push origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 655 bytes | 655.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/TJ-1212/jenkindockerdemo.git
   5933747..90fdb7b  master -> master
```

Jenkins Pipeline for Building & Deploying a Dockerized Java Application

- Steps for the Jenkins Pipeline –
  - Create a Jenkins pipeline and specify the Pipeline script for SCM.


SCM: Git

Repositories

Repository URL: https://github.com/TJ-1212/jenkindockerdemo.git

Credentials: - none -

Branches to build

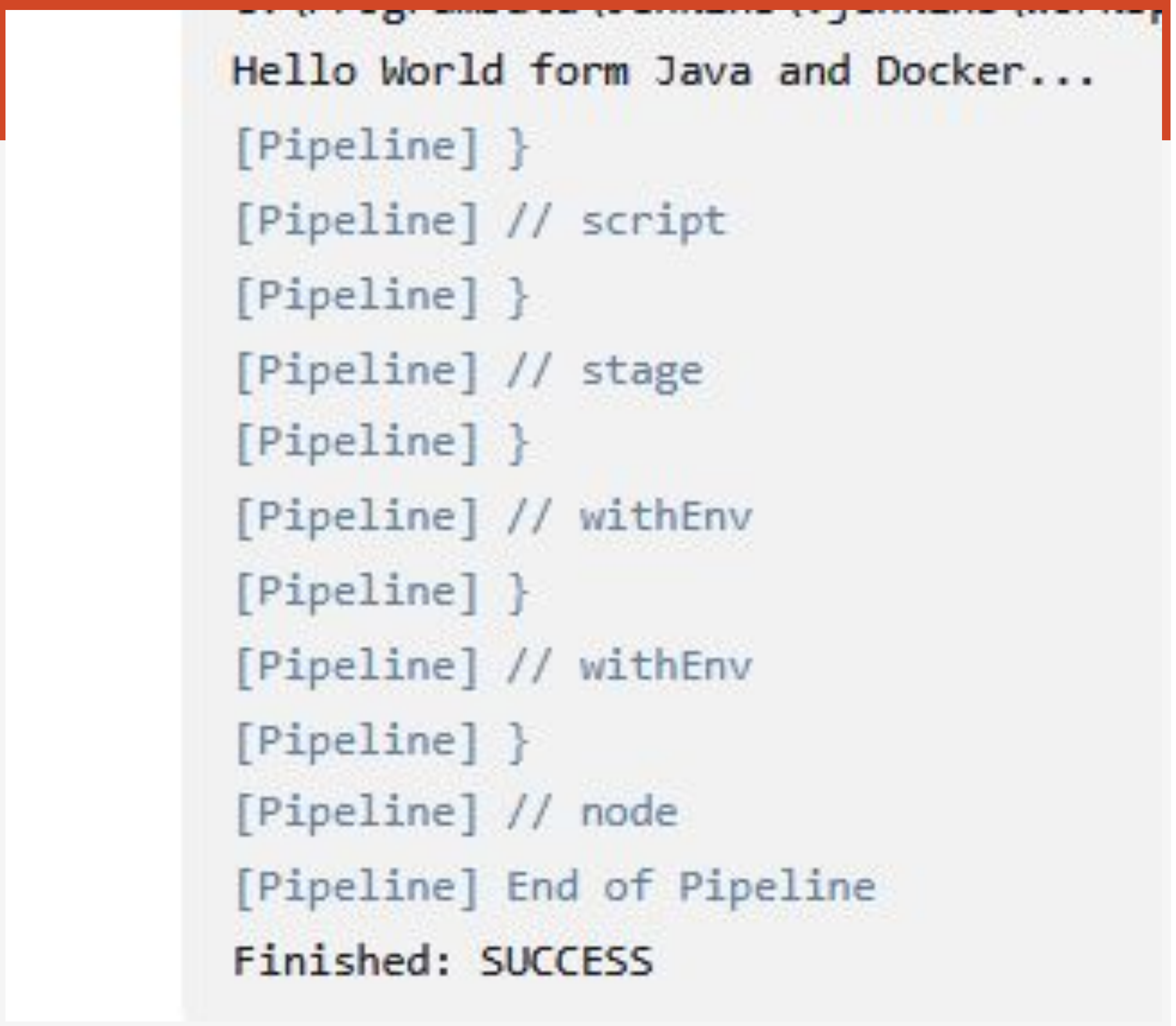
Repository browser: (Auto)

Additional Behaviours

Script Path: Jenkinsfile

Jenkins Pipeline for Building & Deploying a Dockerized Java Application

- Steps for the Jenkins Pipeline –
  - Save and Build the pipeline to get the output.



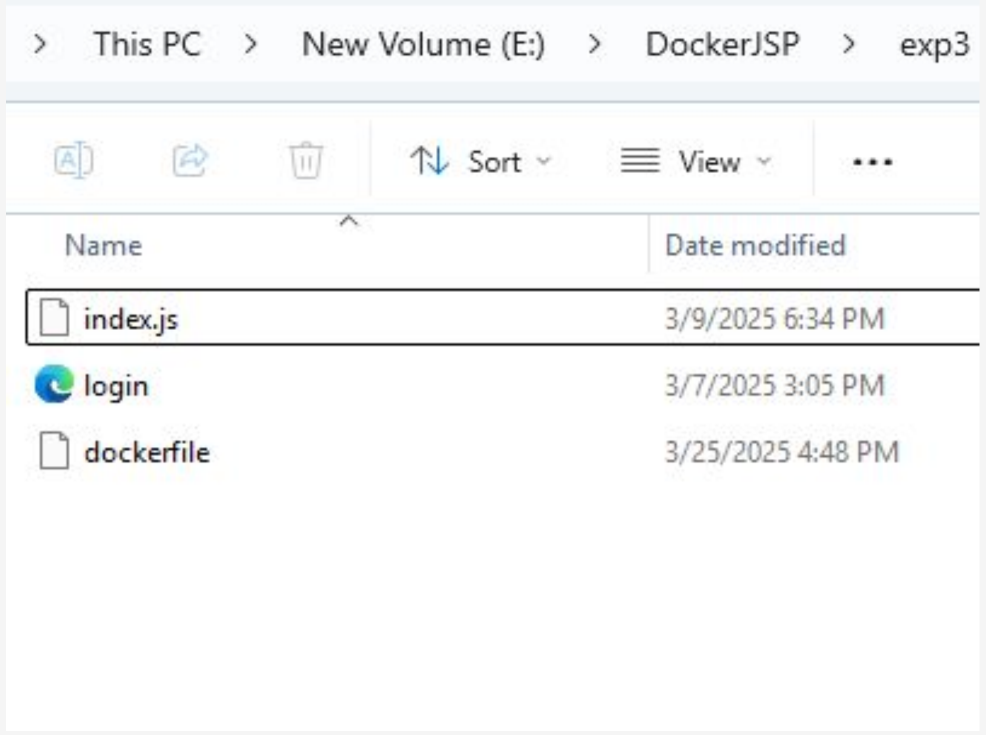
```
Hello World form Java and Docker...
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

6.4

Dealing with NodeJS application

Creating NodeJS Project

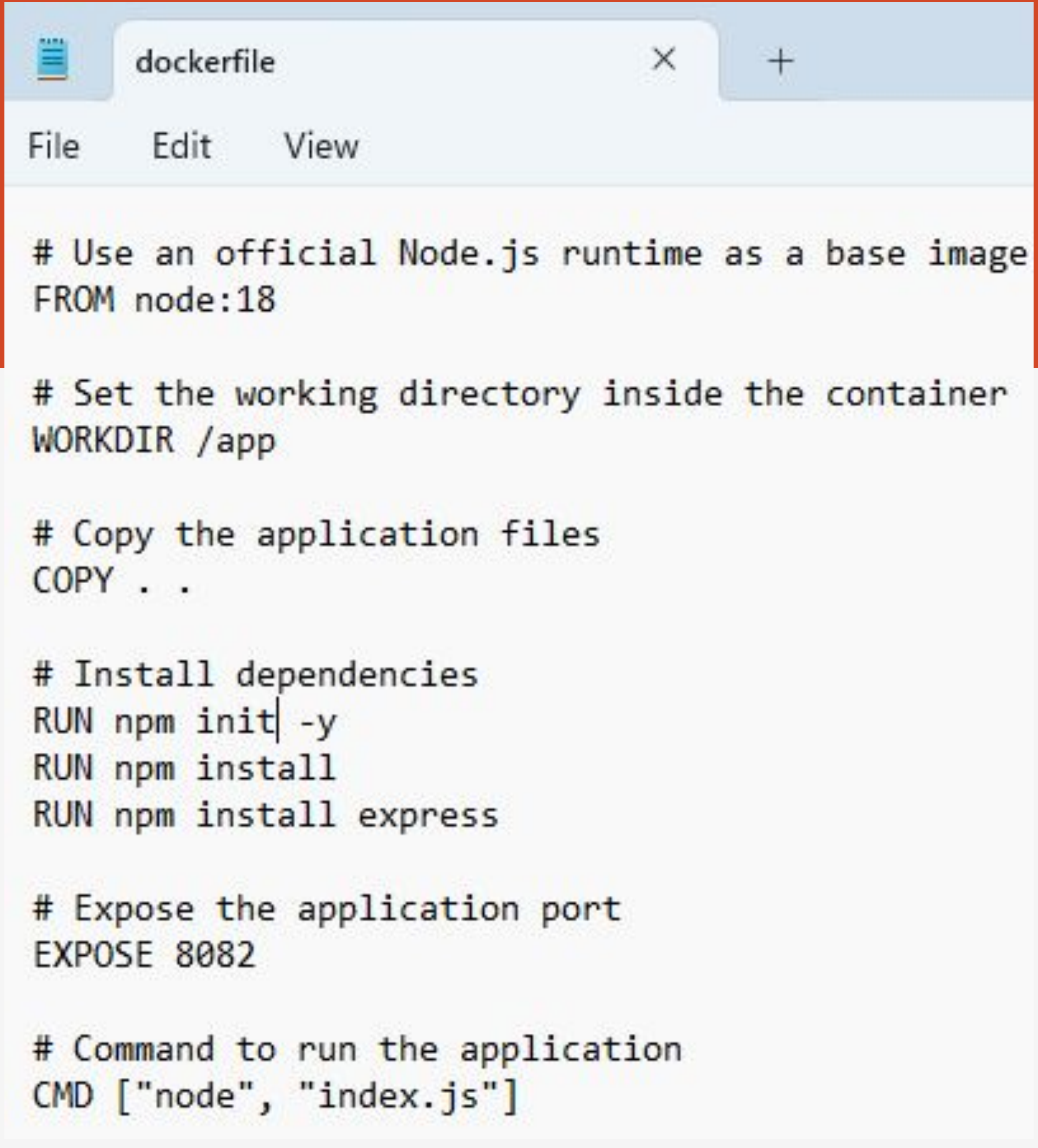
- Create a NodeJS project with the help of VS Code.
- The project structure is shown here.
- Create a "dockerfile" in the Project folder and add the commands shown on next slide.



Name	Date modified
index.js	3/9/2025 6:34 PM
login	3/7/2025 3:05 PM
dockerfile	3/25/2025 4:48 PM

Creating NodeJS Project

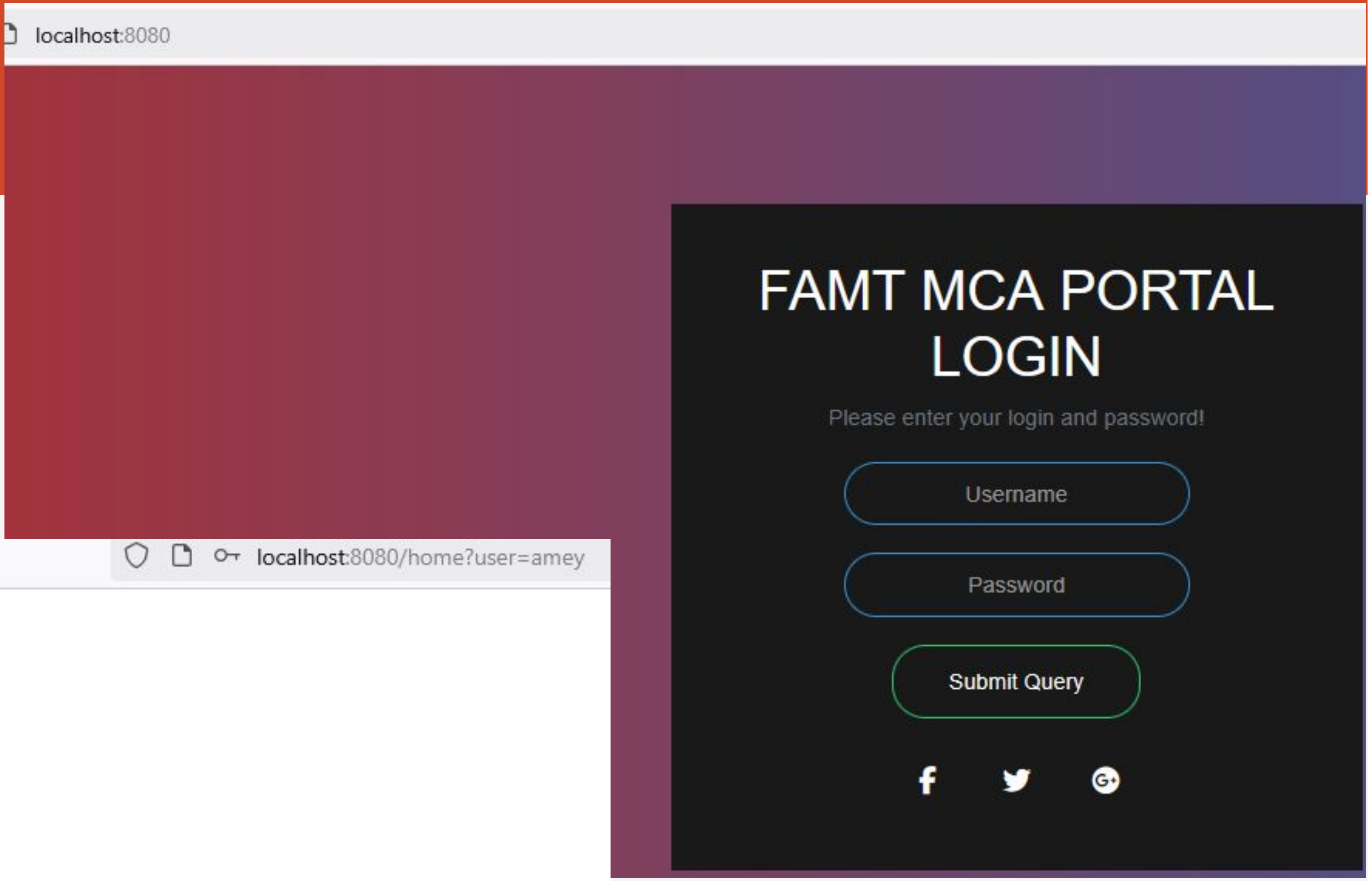
- After adding commands to the dockerfile, build and run the container.
- To build use –
  - docker build –t mynodeimg:v1 .
- To run use –
  - docker run –p 8080:8082 mynodeimg:v1



```
# Use an official Node.js runtime as a base image
FROM node:18

# Set the working directory inside the container
WORKDIR /app

# Copy the application files
COPY . .

# Install dependencies
RUN npm init -y
RUN npm install
RUN npm install express

# Expose the application port
EXPOSE 8082

# Command to run the application
CMD ["node", "index.js"]
```

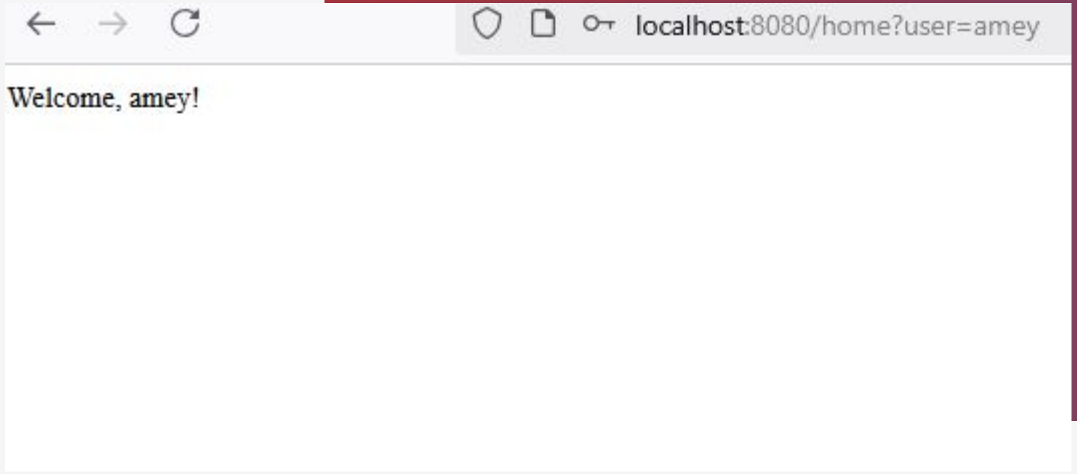
Creating NodeJS Project

- After running docker run command –


localhost:8080

FAMT MCA PORTAL LOGIN

Please enter your login and password!


localhost:8080/home?user=amey

Welcome, amey!