INTRODUCTION TO DOCKER

Docker is an **open-source platform** designed to simplify the process of building, deploying, and running applications using containerization.

Containerization, in the context of Docker, refers to the process of packaging an application and all its dependencies (code, runtime, system tools, libraries, and settings) into a single, standardized isolated unit called a container. This container can then be run consistently across various environments, regardless of the underlying infrastructure.

Why Docker is Needed

Before Docker, a common issue in software development was the "it works on my machine" problem. Moving applications between environments often cause inconsistencies due to differences in dependencies, configurations, and operating systems.

Docker solves this by packaging applications in a standardized and portable way, ensuring consistent performance across different environments.

Advantage: It enables developers to work faster, collaborate more effectively, and deliver software reliably and efficiently across various environments.

Key Advantages and Benefits of Using Docker

- **Consistency across Environments**
  - Docker ensures consistent application behavior across different environments, from local development to production servers.
- **Faster Deployment**
  - Containers are lightweight and start quickly, leading to faster deployments and more efficient resource use compared to virtual machines.
- **Efficient Resource Utilization**
  - Containers share the host's operating system kernel, reducing overhead. This allows more applications to run on a single machine, potentially saving on hardware and infrastructure costs.
- **Isolation and Security**
  - Docker containers provide process isolation, enhancing security by ensuring applications run in separate environments. This reduces the risk of vulnerabilities impacting other parts of the system.
- **Portability**
  - Docker containers are portable and can run on any system that supports Docker, including cloud platforms like AWS, Azure, and Google Cloud.
- **Scalability**
  - Docker simplifies application scaling by allowing quick creation of new container instances to handle increased traffic or scaling down during low usage periods, according to prescienceds.com.

- **Simplified Development and Testing**
    - Docker provides isolated development environments, eliminating the "it works on my machine" problem, allowing developers to work with consistent setups.
- **Collaboration**
    - Docker simplifies team collaboration by enabling developers to easily share containers and configurations, ensuring everyone works with the same software stack and dependencies.
- **DevOps Integration**
    - Docker integrates well with DevOps practices, streamlining CI/CD pipelines, automating deployment workflows, and enabling faster, more reliable software delivery

Docker is a platform and set of tools for building, deploying, and running applications in containers. At its heart is the **Docker Engine** – which manages the creation, execution, and lifecycle of these containers. Docker Engine utilizes features like Linux namespaces and control groups (cgroups) to isolate containers and manage their resources. So, Docker can run directly on Linux.

Docker Engine is Native to Linux:

- Docker containers leverage features of the Linux kernel (like namespaces and cgroups) to provide isolation and resource management.
- This means the core Docker functionality (Docker Engine) can run directly on a Linux operating system without the need for a virtual machine or other virtualization layers.

Running Docker on Linux

To use Docker directly on Linux, need to install Docker Engine:

1. Install Docker Engine. For Red Hat derivatives, install Docker Engine as follows:
   sudo dnf install docker-io
2. Start Docker Service
   sudo systemctl enable docker
   sudo systemctl start docker
3. Add User to Docker Group (Optional but Recommended)
4. Install Docker Compose (needed if application used for multi-container deployments)
    1. Install Docker Compose plugin (recommended for modern Docker versions)
       sudo dnf install docker-compose-plugin
    2. Use Docker Compose
       docker compose up

NOTE: Docker cannot run natively on operating systems like macOS and Windows that lack these specific Linux kernel functionalities.

Docker Desktop for Mac and Windows

Docker Desktop is a user-friendly application designed to bridge this gap, specifically for Windows and macOS environments. It provides a complete Docker environment on these systems by bundling Docker Engine within a lightweight Linux virtual machine (VM).

Docker Desktop Installation on Windows 11

1. Open a web browser and navigate to the official Docker website https://docs.docker.com/desktop/setup/install/windows-install/
2. Download **Docker Desktop for Windows-x86_64**
3. Locate the downloaded .exe file and double-click to run the installer.
4. During the installation, ensure that the option **"Use the WSL 2 based engine"** is checked, as this is the recommended and default setting for better performance.
    - During Docker Desktop installation, if you encounter a message suggesting you run **wsl --update**, it means your WSL 2 (Windows Subsystem for Linux 2) installation requires an update to its kernel component or other related components for optimal compatibility with Docker Desktop.
    - Run **wsl --update**
    - Restart your computer when prompted
5. After restarting, open Docker Desktop from the Start Menu or the hidden icons menu on your taskbar.
6. Follow the prompts to accept the service agreement and choose your settings. You can sign in to your Docker account or skip the process.
7. Open the Windows terminal or command prompt and type **docker --version** to check if Docker is installed correctly and running.
8. Alternatively, you can run **docker run hello-world** to pull and run a test container, confirming that Docker is functioning.

Week-6: Explore Docker commands for content management

open the command prompt in your working directory or Docker Desktop terminal and execute the following commands:

docker pull <image-name>

- Downloads the specified Docker image from Docker Hub (or another Docker registry) to the local machine.
- Example: **docker pull hello-world**
- Output:

docker run <image-name>

- Creates and starts a new container from the specified image.
- Runs the container immediately.
- Example: **docker run hello-world**
- Output:

docker ps

- Lists all currently running containers
- Output:

docker ps –a

- Lists all containers on your system, including running, stopped, and exited ones
- Output:

docker images

- Lists all Docker images stored locally on your machine
- Output:

docker rm <container-id or container-name>

- Deletes the specified container from your system.
- The container must be stopped before removal.
- Frees up resources and cleans up unused containers.
- Output:

docker rmi <image-name>

- Deletes the specified Docker image from your local storage
- You can't remove an image if containers based on it are still present (you must remove those containers first).

docker start <container-id or container-name>

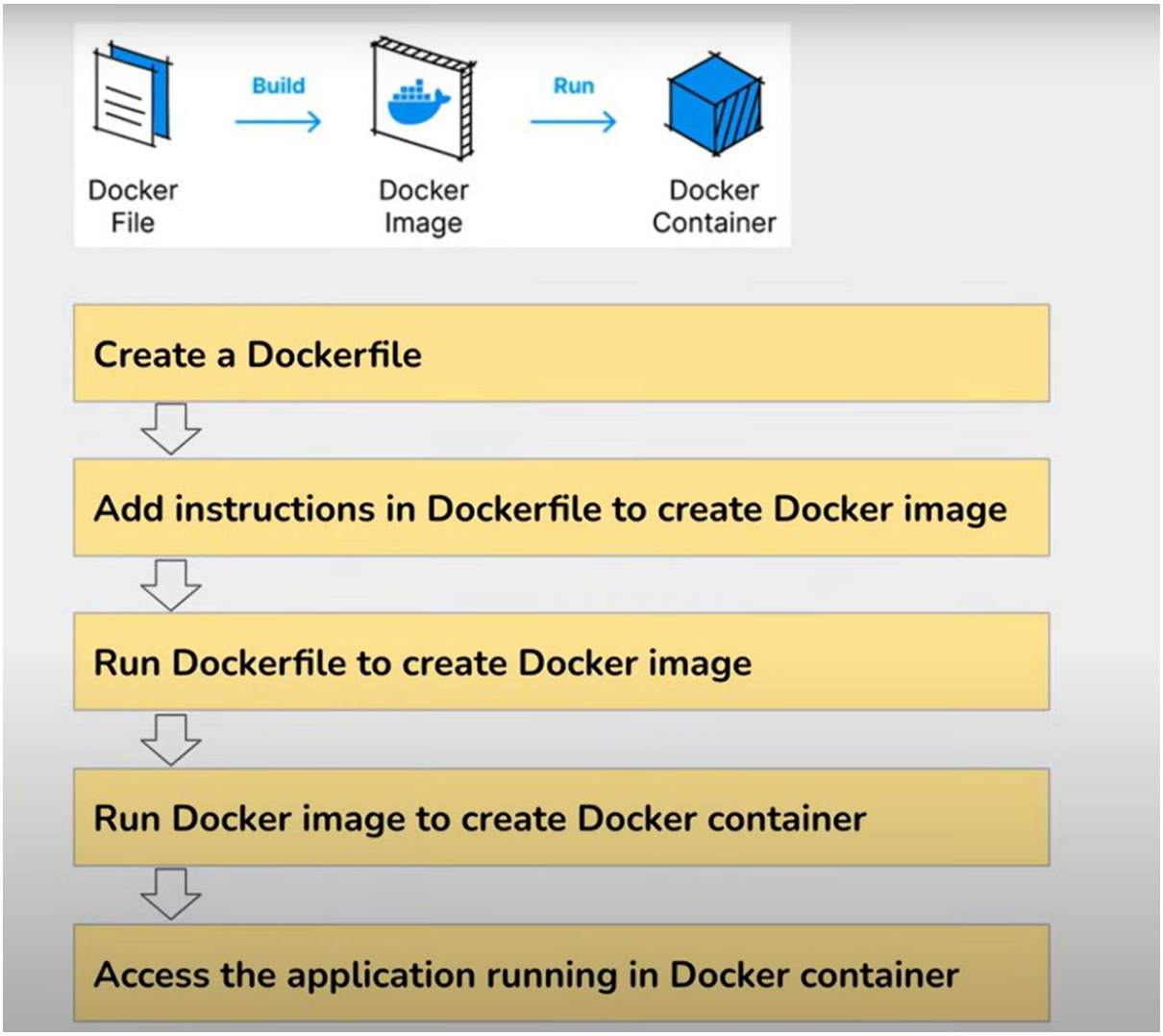
- Starts the specified container.

docker stop <container-id or container-name>

- Stops the specified container.

Output:

Steps to "Containerize/Dockerize" Applications (OR) Build and Run Applications In Docker Container:



A Dockerfile is a text file that contains a set of instructions used to build a Docker image.

Docker reads the Dockerfile line-by-line and:

- Creates a temporary container layer by layer.
- Installs the app and its dependencies.
- Packages everything into a Docker image.
- Then that image can then be run as a container.

A Docker image is like a blueprint for a container. It includes:

- The application code
- Runtime (like Python or Node.js)
- Dependencies (like Flask)
- Instructions on how to run the app

Dockerfile Instructions:

FROM → specifies the base image to use for the Docker image

WORKDIR → sets the working directory inside the container for subsequent commands in the Dockerfile

COPY → copies file from the local file system to the Docker image

RUN → runs a command inside the Docker container during the build process

EXPOSE <port> → tells Docker and other developers which port the application inside the container is using

CMD → specifies the command to run when the Docker container starts

Simple Python Application: Write simple python application in VS code in your workspace

app.py

```
import os
print("Hello world")
print("Build python image")
print("current directory is : ",os.getcwd())
print(os.listdir())
```

Dockerfile

```
FROM python
WORKDIR /app
COPY . /app
CMD ["python3", "app.py"]
```

Containerize (or) Build Docker Image for simple python applications: Go to terminal in vs code editor, navigate to your workspace directory and execute the following commands:

To build the image in docker

> **docker build -t <image-name> .**
> it creates docker image with default tagname as "**latest**"
> Example**:** docker build -t myfirstpythonapp .

To build the Docker Image with specified tag name

> **docker build -t <image-name>:<tag-name> .**
> it creates docker image with specified tagname
> Example: docker build -t myfirstpythonflaskapp:v1 .

To run the image in container

> **docker run <image-name>**
> Example: docker run myfirstpythonapp

To run the docker image in isolated container by specifying container name

> **docker run --name <container-name> <image-name>**
> Example: docker run --name mycontainer myfirstpythonapp

NOTE: The **docker run** command always creates a new container from the specified image.

- If you specify a container name using **--name**, Docker expects that name to be unique.

- If a container with the same name already exists (even if it's stopped), running the same command again will result in an error.
- Ensure to remove an existing container before re-running the docker run command
  docker rm mycontainer
  docker run --name mycontainer myfirstpythonapp

For Python Flask Application: Write python flask application in VS code in your workspace

app.py

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def firstApp():
   return "<h1>Hello World!! Welcome Docker</h1>"
if __name__ == "__main__":
   app.run(host='0.0.0.0', port= 5000, debug = True)
```

Dockerfile

```
FROM python:3.9-slim-buster
WORKDIR /app
COPY . /app
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 5000           // Documents that the container listens on port 5000.
CMD ["python","app.py"]
```

RUN pip install --no-cache-dir -r requirements.txt →Installs Python dependencies listed in requirements.txt.

--no-cache-dir → prevents pip from caching installation files, keeping the image smaller.

---

requirements.txt

```
flask
```

Containerize (or) Build Docker Image for Python Flask applications: Go to terminal in vs code editor, navigate to your workspace directory and execute the following commands:

To build the Docker Image

> **docker build -t <image-name> .**
> it creates docker image with default tagname as "**latest**"
> Example: docker build -t myfirstpythonflaskapp .

To build the Docker Image with specified tag name

> **docker build -t <image-name>:<tag-name> .**
> it creates docker image with specified tagname
> Example: docker build -t myfirstpythonflaskapp:v1 .

To run the docker image in isolated container

docker run -p <hostport>:<containerport> <image-name>
Example: docker run -p 5000:5000 myfirstpythonflaskapp

To run the docker image in isolated container by specifying container name

docker run -p <hostport>:<containerport> --name <container-name> <image-name>
Example: docker run -p 5000:5000 –name mycontainer myfirstpythonflaskapp

NOTE: If any error raises while building docker image, use the following command:

docker system prune It is used to clean up unused Docker data/resources like stopped containers, dangling images (untagged and unused), Unused networks, build cache etc., helping to free up disk space and tidy the Docker environment.

docker system prune -a --volumes used to perform a deep clean

Week-7: Develop a simple containerized application using Docker (through Jenkins CI/CD pipeline)

Pre-requisites:

- Install JDK and set jdk path in environment variables
- Install Git Client and set git path in environment variables
- Install Docker Desktop
- Install Docker plugin
  - open Jenkins Dashboard → Manage Jenkins →Plugins
  - Go to Available plugins and search for Docker
  - Select Docker
  - Click Install
- open Jenkins Dashboard → Manage Jenkins → Tools
  - scroll down, Docker Installations section is configured
  - If Docker Desktop already installed, no additional tool configuration of Docker needed and click Save.
  - Otherwise, click Add Docker → Give it a name (eg: Docker) → check Install automatically → Add Installer → select one of them → click Save

Steps:

1. Write a Python web application code for a simple user registration form.
2. Write/Create **Dockerfile**

```
FROM python:3.9-slim-buster
WORKDIR /app
COPY . /app
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 5000
CMD ["python","app.py"]
```

3. Write/Create **Jenkinsfile**

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        echo "Build Docker Image"
        bat "docker build -t mypythonflaskapp ."
      }
    }
```

```
        stage('Run') {
            steps {
                echo "Run application in Docker Container"
                bat "docker rm -f mycontainer || exit 0"
                //forcibly removes the Docker container named mycontainer
                    //If the container does not exist, this command will fail and return an
error
                    //if fails execute exit 0, tells the shell to exit with a success status

                bat "docker run -d -p 5000:5000 --name mycontainer
mypythonflaskapp"
                //with -d runs the container in detached mode,
                    //meaning it runs in the background, and you get your terminal back
immediately.
                //Without -d runs in the foreground,
                    //terminal shows container logs and is “blocked” by the container
process.
            }
        }
    }
    post {
        success {
            echo 'Pipeline completed successfully!'
        }
        failure {
            echo 'Pipeline failed. Please check the logs.'
        }
    }
}
```

4. Write **requirements.txt**

```
flask
```

5. Commit and Check in your project to the SCM repository using git commands.

6. Project/Application directory structure should be as follows:

SamplePythonFlaskApp/

├── app.py

├── requirements.txt

├── Dockerfile

├── Jenkinsfile

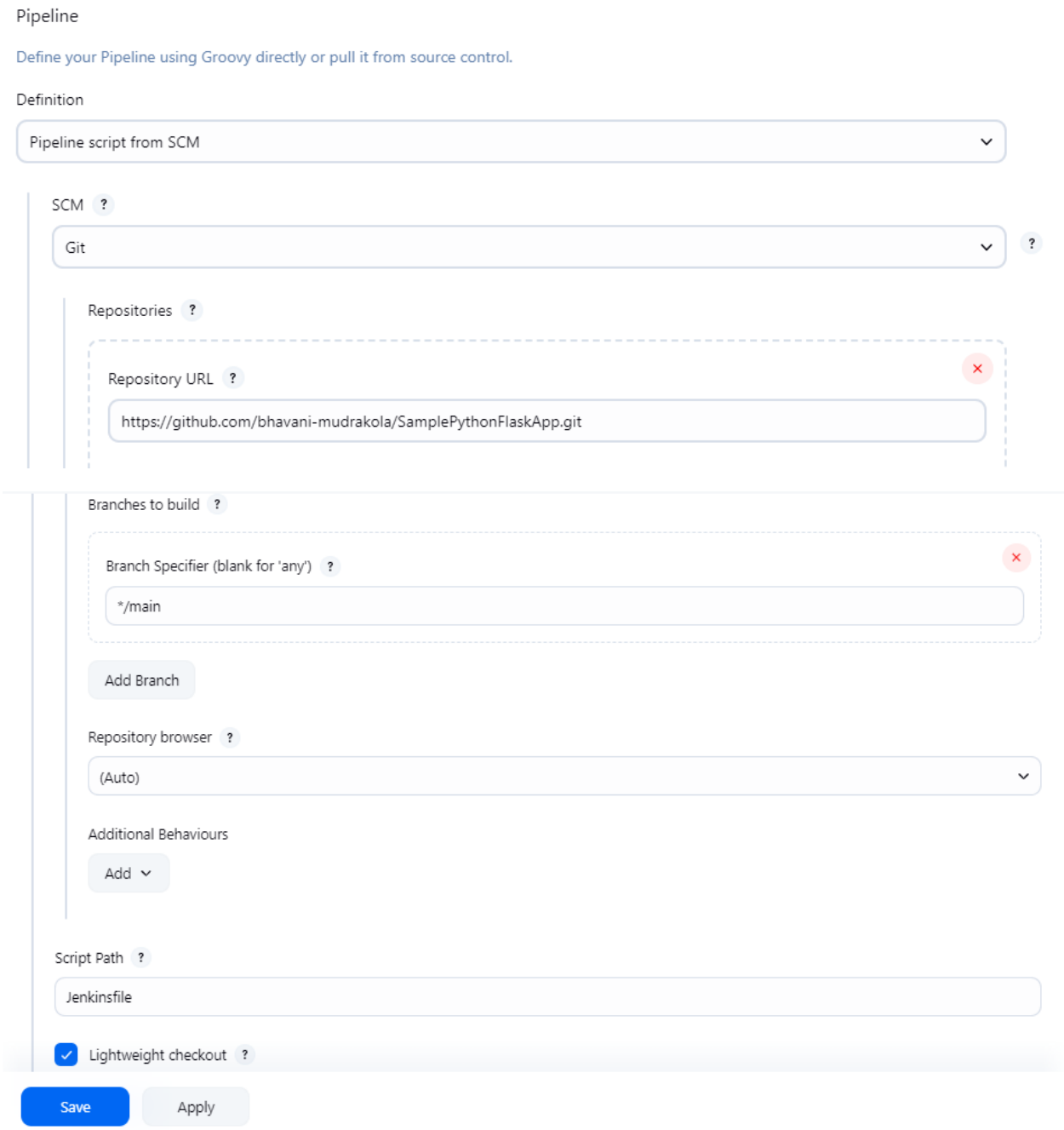
├── templates/

├── registration.html

├── greetings.html

7. Create CI/CD pipeline in Jenkins
    - Open Jenkins Dashboard → click New Item → select Pipeline → Ok
    - Configure the pipeline
        - Give description
        - Check Poll SCM and schedule polling for 5minutes – to automate the process of source code change management and build the application

        - Configure Pipeline section
            - Select Define Pipeline Script from SCM, as pipeline script is written in Jenkinsfile which is in SCM.
            - Select SCM as Git
            - Specify the <repository-url>
            - Specify the branch name
            - Specify the Script Path as Jenkinsfile
        - Click on Apply/Save – saves the pipeline job after configuration



8. Run and Monitor Pipeline Job
    - On the Pipeline dashboard, click **Build Now** to trigger the pipeline manually for the first time.
    - Jenkins will now execute stages defined inside the Pipeline Script
    - Click on the build number (e.g., #1)
        - → Console Output to monitor the pipeline logs in real time.
        - → Pipeline Overview to view the different stages executed in the pipeline

Output: