Docker

What is Docker?

Docker is a **containerization platform** that allows you to package applications and their dependencies into **containers**.

Key Concepts

- **Image**: A blueprint/template for a container.
- **Container**: A running instance of an image.
- **Dockerfile**: A script to create images.
- **Docker Compose**: A tool for managing multi-container applications.
- **Volume**: A way to persist data in containers.

Run Your First Container

Try running a simple **hello-world** container:

```
docker run hello-world
```

It should output a welcome message.

Step 3: Learn Docker Commands

1. **List running containers**:

   ```
   docker ps
   ```

2. **List all containers (including stopped ones)**:

   ```
   docker ps -a
   ```

3. **Stop a container**:

```
docker stop <container_id>
```

4. **Remove a container**:

```
docker rm <container_id>
```

5. **List all images**:

```
docker images
```

6. **Remove an image**:

```
docker rmi <image_id>
```

Understanding Dockerfiles and Building Images

A **Dockerfile** is a script containing instructions to build a Docker image.

Create a Simple Dockerfile

1. Create a new folder for your project:

```
mkdir MyDockerApp
cd MyDockerApp
```

2. Inside this folder, create a file named `Dockerfile` (without any extension).
3. Open `Dockerfile` in a text editor (like VS Code or Notepad++) and add the following content:

```
# Use an official Python image as the base image
FROM python:3.9

# Set the working directory inside the container
```

```
WORKDIR /app

# Copy files from the local machine to the container
COPY . .

# Run a Python script when the container starts
CMD ["python", "app.py"]
```

4. Now, create a simple Python script named `app.py` in the same folder:

```
print("Hello from Docker!")
```

Step 5: Building and Running a Docker Image

5.1 Build the Docker Image

Run the following command in PowerShell inside the `MyDockerApp` folder:

```
docker build -t my-python-app .
```

- `t my-python-app`: Assigns a name (`my-python-app`) to the image.
- `.` : Tells Docker to look for the `Dockerfile` in the current directory.

After running this, Docker will pull the required **Python image** and create your custom image.

5.2 Run the Docker Container

Once the image is built, run a container from it:

```
docker run my-python-app
```

Expected output:

```
Hello from Docker!
```

5.3 List All Images

To see the built images, run:

```
docker images
```

5.4 Stop and Remove Containers

To remove all stopped containers:

```
docker container prune
```

To remove an image:

```
docker rmi my-python-app
```

Step 6: Understanding Docker Volumes (Persistent Data)

By default, when a container stops, all its data is lost. **Docker Volumes** allow data to persist.

6.1 Run a Container with a Volume

```
docker run -v my_volume:/app/data my-python-app
```

- `v my_volume:/app/data` → Creates a volume named `my_volume` that maps to `/app/data` inside the container.

6.2 List Volumes

```
docker volume ls
```

6.3 Remove a Volume

```
docker volume rm my_volume-
```

Understanding Docker Compose

7.1 What is Docker Compose?

Docker Compose allows you to define and run multi-container applications using a `docker-compose.yml` file.

It is useful when you need to run multiple services together, like a **web app** and a **database**.

Step 8: Install Docker Compose

Docker Compose is included with **Docker Desktop**, so you already have it.

To verify, run:

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

Step 9: Create a Multi-Container Application

We'll create a simple **Flask web application** with a **PostgreSQL database**.

9.1 Set Up the Project Structure

```
docker-compose-tutorial/
│-- app/
│    ├── Dockerfile
│    ├── app.py
│    ├── requirements.txt
│-- docker-compose.yml
```

9.2 Create a Flask App

1️⃣ **Create the project folder**:

```
mkdir docker-compose-tutorial
cd docker-compose-tutorial
mkdir app
```

2️⃣ **Create the** `app.py` **file** in the `app` folder:

```
from flask import Flask
import psycopg2

app = Flask(__name__)

@app.route('/')
def home():
    return "Hello from Flask with PostgreSQL!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

3️⃣ **Create the** `requirements.txt` **file**:

```
flask
psycopg2
```

9.3 Create the `Dockerfile` for Flask

Inside the `app/` folder, create a `Dockerfile`:

```
# Use Python as the base image
FROM python:3.9

# Set the working directory
WORKDIR /app
```

```
# Copy the project files into the container
COPY . .

# Install dependencies
RUN pip install -r requirements.txt

# Run the app
CMD ["python", "app.py"]
```

9.4 Define the `docker-compose.yml` File

Go back to the **root folder** (`docker-compose-tutorial/`) and create `docker-compose.yml`:

```
version: '3.8'

services:
  web:
    build: ./app
    ports:
      - "5000:5000"
    depends_on:
      - db

  db:
    image: postgres:13
    environment:
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: mypassword
      POSTGRES_DB: mydatabase
    ports:
      - "5432:5432"
    volumes:
      - db_data:/var/lib/postgresql/data
```

```
volumes:
  db_data:
```

Explanation

- `web` → Builds the Flask app from the `app/` folder.
- `db` → Uses a PostgreSQL image with a database.
- **Volumes** → Stores database data persistently.

9.5 Run the Multi-Container App

From the **docker-compose-tutorial/** folder, run:

```
docker-compose up
```

🚀 The Flask app and PostgreSQL will start.

Go to **http://localhost:5000** in your browser. You should see:

```
Hello from Flask with PostgreSQL!
```

9.6 Stopping and Cleaning Up

Stop the services:

```
docker-compose down
```

Remove all containers, networks, and volumes:

```
docker-compose down --volumes
```

Deploying Docker Containers to the Cloud

10.1 What is Docker Hub?

Docker Hub is a public registry for storing and sharing Docker images.

10.2 Push Your Image to Docker Hub

1️⃣ Log in to Docker Hub

If you don't have an account, create one at https://hub.docker.com.

Then, log in via terminal:

```
docker login
```

Enter your Docker Hub **username** and **password**.

2️⃣ Tag Your Image

If you have a local image named `my-python-app`, tag it with your Docker Hub username:

```
docker tag my-python-app your_dockerhub_username/my-python-app:latest
```

3️⃣ Push the Image to Docker Hub

```
docker push your_dockerhub_username/my-python-app:latest
```

Now, your image is available online!

Step 11: Deploying to AWS EC2

11.1 Create an AWS EC2 Instance

1. Go to the **AWS Management Console**.
2. Navigate to **EC2 > Launch Instance**.
3. Choose an **Ubuntu 22.04** AMI.
4. Select a t2.micro instance (Free Tier eligible).
5. Create or select an **existing key pair**.
6. Allow inbound **port 22 (SSH)** and **port 80 (for web apps)**.

7. Launch the instance.

11.2 Install Docker on EC2

1. Connect to EC2 via SSH:

```
ssh -i your-key.pem ubuntu@your-ec2-public-ip
```

2. Install Docker:

```
sudo apt update
sudo apt install docker.io -y
```

3. Start and enable Docker:

```
sudo systemctl start docker
sudo systemctl enable docker
```

4. Verify Docker installation:

```
docker --version
```

11.3 Run Your Docker Container on EC2

1. Pull your Docker image from Docker Hub:

```
docker pull your_dockerhub_username/my-python-app:latest
```

2. Run the container:

```
docker run -d -p 80:5000 your_dockerhub_username/my-python-app
```

3. Open your EC2 **public IP** in a browser (`http://your-ec2-public-ip`) to see your app running!

Additional Info

Step 13: What is Kubernetes (K8s)?

Kubernetes is a **container orchestration platform** that automates:

- Deployment
- Scaling
- Load balancing
- Management of containers

Think of it as a **control center** for your Docker containers.

Step 14: Set Up Kubernetes Locally (Using Minikube)

We'll use **Minikube**, which creates a local Kubernetes cluster on your machine.

14.1 Install Minikube and kubectl (Kubernetes CLI)

1. **Install kubectl**:
   Download from https://kubernetes.io/docs/tasks/tools/
2. **Install Minikube for Windows** (if using Docker Desktop, you can skip VM requirements):

   ```
   choco install minikube
   ```

3. Start Minikube:

   ```
   minikube start --driver=docker
   ```

4. Check if your cluster is running:

   ```
   kubectl get nodes
   ```

Step 15: Deploy an App to Kubernetes

Let's deploy the same `my-python-app` image.

15.1 Create a Deployment

Create a file named `deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: python-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: python-app
  template:
    metadata:
      labels:
        app: python-app
    spec:
      containers:
      - name: python-app
        image: your_dockerhub_username/my-python-app:latest
        ports:
        - containerPort: 5000
```

Apply the deployment:

```
kubectl apply -f deployment.yaml
```

Check running pods:

```
kubectl get pods
```

15.2 Expose the App with a Service

Create `service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: python-service
spec:
  type: NodePort
  selector:
    app: python-app
  ports:
    - port: 80
      targetPort: 5000
      nodePort: 30007
```

Apply the service:

```
kubectl apply -f service.yaml
```

Now access the app:

```
minikube service python-service
```

This opens the app in your browser via the **NodePort (30007)**.

Step 16: Scale Your App

Kubernetes makes it super easy to scale:

```
kubectl scale deployment python-app --replicas=5
kubectl get pods
```

Now your app has 5 containers running in parallel! 🚀

Step 17: Clean Up

```
kubectl delete -f deployment.yaml
kubectl delete -f service.yaml
minikube stop
```