Experiment No 10

Aim : Containerizing App with Docker

Theory :

1. Introduction

This report outlines the process of containerizing a full-stack application using **Docker**. The goal is to create a lightweight, portable, and reproducible environment for both frontend and backend services.

By leveraging Docker, the application can run consistently across different environments without compatibility issues. The backend is containerized using a custom **Dockerfile**, while the frontend and backend Docker images can be built and deployed as needed.

This approach ensures seamless collaboration, simplified deployment, and scalability for modern application development.

2. Core Components and Technologies

2.1 Docker

Docker is an open-source platform designed to automate the deployment of applications inside lightweight, portable containers. Containers package the application code along with its dependencies, ensuring the application runs identically across various environments such as development, testing, and production.

In this project, Docker is used to:

- Create container images for both frontend and backend.
- Simplify dependency management and environment setup.
- Provide scalability and portability for deployment.

2.2 Dockerfile

The Dockerfile is a blueprint for creating Docker images. It contains step-by-step instructions for setting up the environment, installing dependencies, and running the application.

For the backend:

- **Base Image:** Uses a lightweight Node.js image (node:18-alpine).
- **Set Working Directory:** Configures /app as the working directory.
- **Install Dependencies:** Copies package.json and runs npm install.

- **Copy Source Code:** Adds the remaining backend code into the container.
- **Expose Port:** Opens port **3000** to run the backend service.
- **Startup Command:** Uses CMD ["node", "server.js"] to start the application.

2.3 Docker Hub

Docker Hub is a cloud-based registry for storing and distributing Docker images.
In this project:

- Separate repositories can be maintained for **frontend** and **backend** images.
- Images can be tagged with latest or version numbers to track updates.
- These images can then be pulled and deployed on any server or cloud provider.

3. Docker Workflow

3.1 Backend Containerization Steps

The process of containerizing the backend includes:

1) **Create Dockerfile**
    - Define the base image.
    - Install dependencies using npm install.
    - Copy application files into the container.
    - Expose the application port.
    - Define the startup command.

2. Build the Docker Image

```
docker build -t backend-image:latest ./backend
```

3.Run the Container

```
docker run -p 3000:3000 backend-image:latest
```

This runs the backend service and makes it accessible on port 3000.

4. Ports and Networking

- **Backend Service:** Configured to run on port 3000 (exposed in the Dockerfile).
- **Frontend Service:** Runs on a separate port, defined during container runtime.
- Docker provides a default network so that containers can communicate with each other by their container names.

5. Important Docker Commands

Here are some commonly used Docker commands:

Command	Description
docker --version	Check Docker version installed.
docker build -t <image-name> .	Build a Docker image from the Dockerfile.
docker images	List all available Docker images.
docker run -p <host-port>:<container-port> <image-name>	Run a container and map ports.
docker ps	List all running containers.
docker stop <container-id>	Stop a running container.
docker rm <container-id>	Remove a stopped container.
docker rmi <image-id>	Remove an image.
docker logs <container-id>	View logs of a running container.
docker exec -it <container-id> /bin/sh	Access the container's shell.
docker pull <image-name>	Pull an image from Docker Hub.
docker push <image-name>	Push an image to Docker Hub.

6. Benefits of Containerizing the Application

6.1 Consistency Across Environments

Containers ensure the application behaves the same way in development, testing, and production.

6.2 Portability

Docker images can be deployed to any system that has Docker installed.

6.3 Isolation

Frontend and backend services run in separate containers, preventing conflicts and improving security.

6.4 Simplified Deployment

Containerized services can be easily updated or scaled up with minimal downtime.

30% Extra Work – Automation with GitHub Actions

7.1 Overview

GitHub Actions is used to automate the process of building and pushing Docker images to Docker Hub. This eliminates the need for manual steps, speeding up the deployment cycle.

7.2 Workflow Steps

1. **Checkout Code:** Retrieves the latest code from the repository.
2. **Set Up Docker Buildx:** Enables advanced build capabilities.
3. **Log In to Docker Hub:** Uses credentials stored in GitHub Secrets.
4. **Build and Push Docker Images:**
    - Frontend and backend images are built separately.
    - Tagged as latest before being pushed to Docker Hub.

7.3 Configuration Details

Frontend Build:

```
context: .
file: ./Dockerfile
tags: web-image:latest
```

Backend Build:

```
context: ./backend
file: ./backend/Dockerfile
tags: backend-image:latest
```

This automation ensures a streamlined and repeatable build and deployment pipeline.

Codes :

Server.js in backend

docker-sid > backend > JS server.js > ...

```
const express = require('express');
const fetch = require('node-fetch');
const cors = require('cors');
const rateLimit = require('express-rate-limit');
const app = express();

app.use(cors({
  origin: ['http://localhost:5173', 'http://localhost']
}));
app.use(rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100 // 100 requests per window
}));

app.get('/joke', async (req, res) => {
  try {
    const response = await fetch('https://official-joke-api.appspot.com/random_joke');
    const data = await response.json();
    res.json({ setup: data.setup, punchline: data.punchline });
  } catch (error) {
    res.status(500).json({ setup: 'Error', punchline: 'Failed to fetch joke' });
  }
});

app.listen(3000, () => console.log('API running on port 3000'));
```

Docker file

docker-sid > backend > Dockerfile > ...

```
# Use an official Node.js runtime
FROM node:18-alpine

# Set working directory
WORKDIR /app

# Copy package.json and install dependencies
COPY package.json .
RUN npm install

# Copy the rest of the backend code
COPY . .

# Expose port 3000
EXPOSE 3000

# Start the server
CMD ["node", "server.js"]
```

DockerBuild in github workflows for github actions

docker-sid > .github > workflows > ! docker-build.yml

```yaml
name: Build and Push Docker Images

on:
  push:
    branches:
      - main

jobs:
  build-frontend:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Log in to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build and push frontend image
        uses: docker/build-push-action@v4
        with:
          context: .
          file: ./Dockerfile
          push: true
          tags: siddharthjogi/docker-sid-web:latest
```

```yaml
build-backend:
  runs-on: ubuntu-latest
  steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v2

    - name: Log in to Docker Hub
      uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}

    - name: Build and push backend image
      uses: docker/build-push-action@v4
      with:
        context: ./backend
        file: ./backend/Dockerfile
        push: true
        tags: siddharthjogi/docker-sid-api:latest
```

app.jsx

```jsx
import { useState, useEffect } from 'react';
import './index.css';

function App() {
  const [count, setCount] = useState(0);
  const [joke, setJoke] = useState({ setup: '', punchline: '' });
  const [isLoading, setIsLoading] = useState(false);

  const apiUrl = import.meta.env.VITE_API_URL || '/api/joke';

  const fetchJoke = async () => {
    setIsLoading(true);
    try {
      const response = await fetch(apiUrl);
      const data = await response.json();
      setJoke({ setup: data.setup, punchline: data.punchline });
    } catch (error) {
      setJoke({ setup: 'Oops!', punchline: 'Something went wrong, try again!' });
    }
    setIsLoading(false);
  };

  useEffect(() => {
    fetchJoke();
  }, []);

  return (
    <div className="min-h-screen bg-gradient-to-br from-indigo-500 via-purple-500 to-pink-500 flex items-center justify-center p-4">
      <div className="bg-white rounded-2xl shadow-2xl p-8 max-w-md w-full transform transition-all duration-500 hover:scale-105">
        <h1 className="text-3xl font-bold text-gray-800 mb-6 text-center">Random Joke Fetcher</h1>
        <div className="flex flex-col items-center space-y-6">
          <div className="text-center">
            <p className="text-lg text-gray-600 mb-2">Click to increment</p>
            <button
              className="bg-indigo-600 text-white font-semibold px-6 py-3 rounded-lg hover:bg-indigo-700 transition-colors duration-300"
              onClick={() => setCount(count + 1)}
            >
              Count: {count}
            </button>
          </div>
          <div className="bg-gray-50 rounded-lg p-6 w-full animate-fadeIn">
            <h2 className="text-xl font-semibold text-gray-700 mb-3">Random Joke</h2>
            {isLoading ? (
              <p className="text-gray-500 italic">Loading joke...</p>
            ) : (
              <>
                <p className="text-gray-600">{joke.setup}</p>
                <p className="text-indigo-600 font-medium mt-2">{joke.punchline}</p>
              </>
            )}
            <button
              className="mt-4 bg-purple-600 text-white px-4 py-2 rounded-lg hover:bg-purple-700 transition-colors duration-300"
              onClick={fetchJoke}
              disabled={isLoading}
            >
              {isLoading ? 'Fetching...' : 'Get New Joke'}
            </button>
          </div>
        </div>
        <p className="mt-6 text-center text-gray-500 text-sm">
          Built with Vite, React, Docker, and Tailwind CSS
        </p>
      </div>
    </div>
  );
}

export default App;
```

Index css

```
@import 'tailwindcss';

@layer utilities {
  .animate-fadeIn {
    animation: fadeIn 0.5s ease-in-out;
  }

  @keyframes fadeIn {
    0% { opacity: 0; transform: translateY(10px); }
    100% { opacity: 1; transform: translateY(0); }
  }
}
```

Dockercompose yml

```
Run All Services
services:
  Run Service
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "80:80"
    environment:
      - NODE_ENV=production
      - VITE_API_URL=/api/joke
    networks:
      - app-network
    depends_on:
      - api

  Run Service
  api:
    build:
      context: ./backend
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
    networks:
      - app-network

networks:
  app-network:
    driver: bridge
```

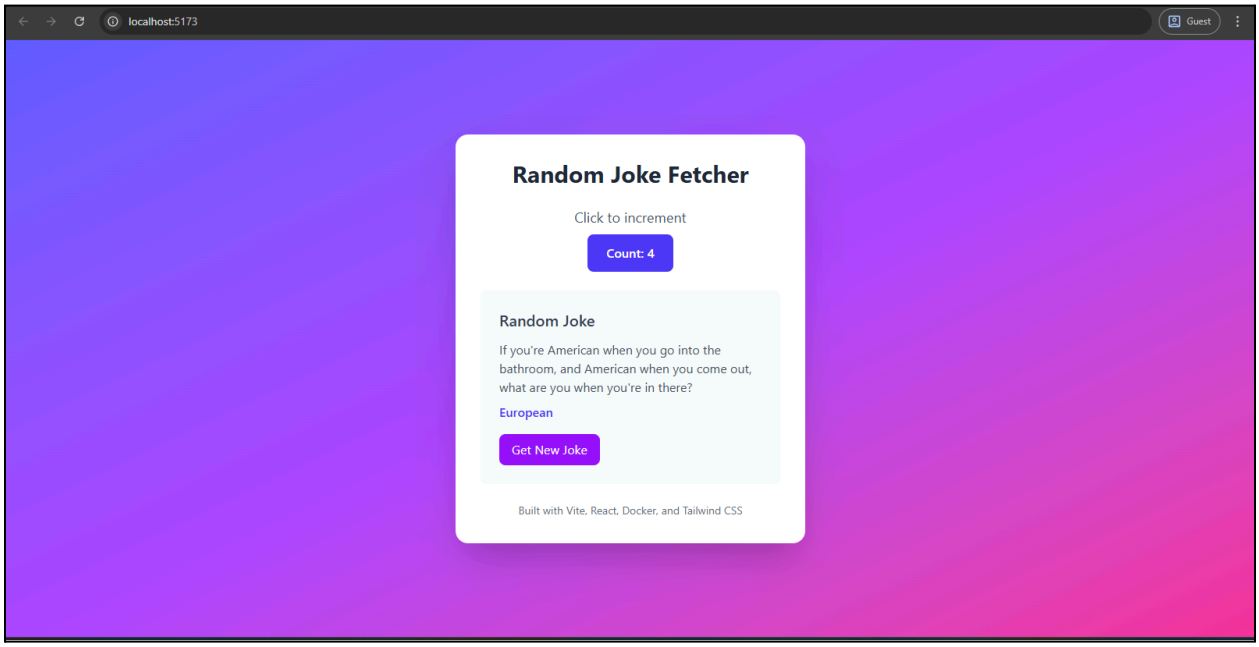
Main dockerfile

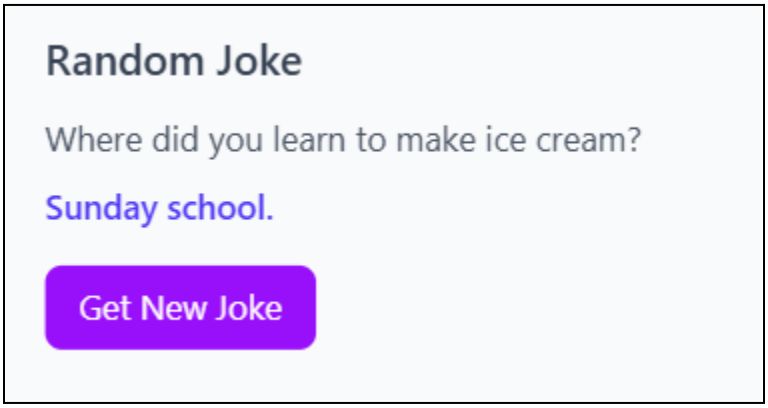
ocker-sid > Dockerfile > ...

```
# Use an official Node.js runtime as the base image
FROM node:18-alpine AS builder

# Set working directory
WORKDIR /app

# Copy package.json and package-lock.json
COPY package.json package-lock.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application code
COPY . .

# Build the Vite app for production
RUN npm run build

# Use a lightweight web server to serve the built app
FROM nginx:alpine

# Copy the build output to the Nginx html directory
COPY --from=builder /app/dist /usr/share/nginx/html

# Ensure permissions are correct
RUN chmod -R 755 /usr/share/nginx/html

# Copy custom Nginx configuration
COPY nginx.conf /etc/nginx/conf.d/default.conf

# Expose port 80
EXPOSE 80

# Start Nginx
CMD ["nginx", "-g", "daemon off;"]
```

Nginx config

Outputs :





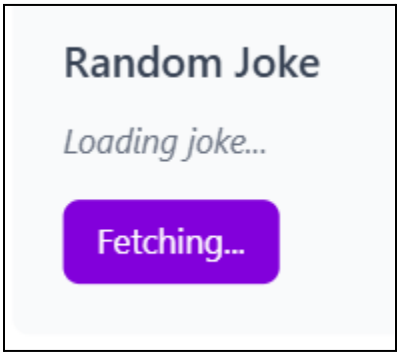


Figure 10.1, 10.2 , 10.3 - Getting a new joke through the joke API

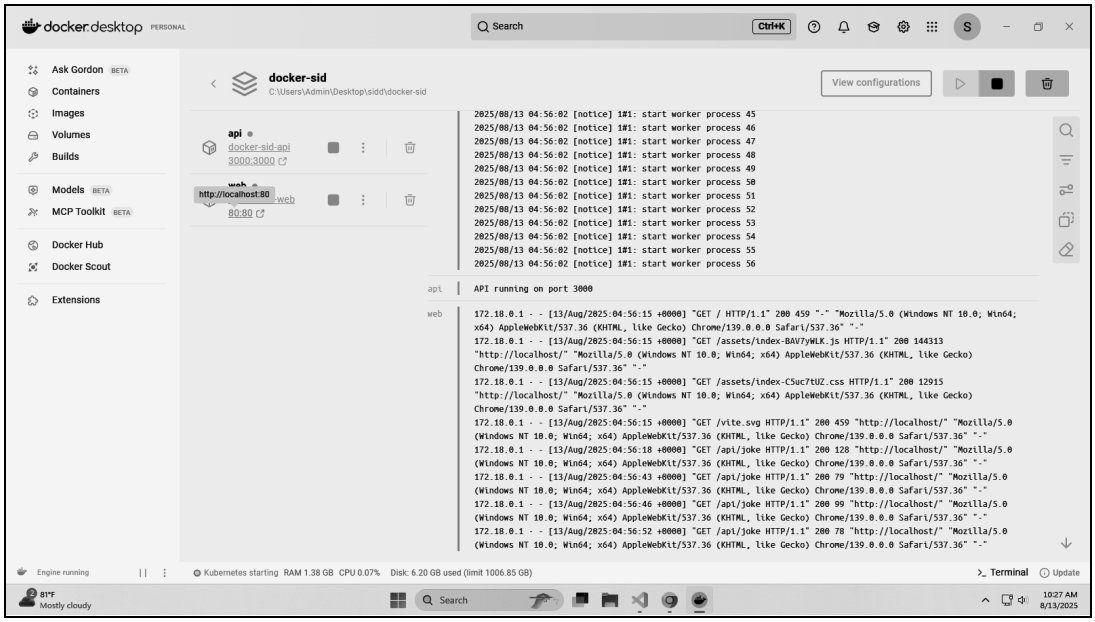


Figure 10.4 - Docker Container Running

Figure 10.5 - Docker Api and Web Containers Working

Figure 10.6 - Docker Compose Command

Conclusion :

Containerizing the application using Docker simplifies development, deployment, and scaling by creating a consistent and portable environment.

The use of Dockerfiles ensures reproducibility, while containerized frontend and backend services improve reliability and isolation. Adding automation with GitHub Actions further enhances productivity by streamlining image builds and deployments, laying the foundation for a modern DevOps workflow.