DEVOPS – TASK

Project Overview

- This project involves containerizing a full-stack web application with React frontend, Express backend, and MongoDB database using Docker.
- The goal is to simplify deployment and scalability by isolating each component into separate containers.
- The setup includes **creating Dockerfiles for both frontend and backend**, configuring a **Docker Compose file to orchestrate the services**, and writing an **automation shell script to build, tag, push, and deploy the application**.
- A volume is added for **MongoDB data persistence**, ensuring data continuity. By automating the setup, the project enables seamless testing and deployment across environments, making the application highly portable and manageable.

Prerequisites

Ensure the following are set up before starting:

- **Docker** installed (installation guide)
- **Docker Hub** account (sign up here)
- **Git** for cloning repositories

Project Components

- **Frontend Dockerfile** - Defines the setup for the React frontend.
- **Backend Dockerfile** - Defines the setup for the Express backend.
- **Docker Compose File** - Manages services for frontend, backend, and MongoDB.
- **Shell Script** - Automates the build, push, and deployment process

Steps to Complete the Project

Clone GitHub Repositories

Clone the provided GitHub repository for both frontend and backend:
Open git in your system and type:

git clone https://github.com/fliprlab/devops-task.git

Create Dockerfiles

FRONTEND DOCKERFILE

Navigate to the frontend directory and create a Dockerfile with the following content:

- **node:18**: A lightweight Node.js image.
- **WORKDIR**: Sets the working directory in the container.
- **COPY and RUN**: Copies dependencies and installs them, then builds the React app.

- **EXPOSE 3000**: Opens the port for the frontend.

```
# Frontend dockerfile

# Base Image
FROM node:18

# Working Directory
WORKDIR /app

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

# Install dependencies
RUN npm ci

# Copy source code
COPY . .

# Build the application
RUN npm run build

# Install lightweight HTTP server globally
RUN npm install -g serve

# Port
EXPOSE 3000

# Health checking
HEALTHCHECK --interval=30s --timeout=10s --retries=3 CMD curl -f http://localhost:3000 // exit 1

# Run the application
CMD ["serve", "-s", "build"]
```

BACKEND DOCKERFILE

Navigate to the backend directory and create a Dockerfile with the following content:

- **Node:18**: A lightweight Node.js image.
- **WORKDIR**: Sets the working directory in the container.
- **COPY and RUN**: Copies dependencies and installs them, then builds the React app.
- **EXPOSE 3000**: Opens the port for the frontend.

```
# Bakcend Dockerfile

# Base Image
FROM node:18

# Working Directory
WORKDIR /app

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

# Install dependencies
RUN npm ci

# Copy source code
COPY . .

# Port
EXPOSE 5000

# Health checking
HEALTHCHECK --interval=30s --timeout=10s --retries=3 CMD curl -f http://localhost:5000 // exit 1

# Run the application
CMD ["npm", "start"]
```

After creating backend and frontend docker file successfully

- Deploy image on container separately for frontend and backend
- And also push to docker Hub (from where any one can download image and use it)

Docker Compose file

In the root directory, create a **docker-compose.yml** file with the following configuration:

- **frontend**: Maps to the frontend Dockerfile, exposing port 3000.
- **backend**: Maps to the backend Dockerfile, exposing port 5000 and connecting to MongoDB via MONGO_URI.
- **mongo**: Uses the official MongoDB image, exposing port 27017 with a volume for data persistence

```yaml
version: '3'

services:
  mongo:
    image: mongo
    container_name: mongo-container
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db

    healthcheck:
      test: ["CMD", "mongo", "--eval", "db.adminCommand('ping')"]
      interval: 30s
      timeout: 10s
      retries: 5

  backend:
    build: ./backend
    container_name: backend-container
    ports:
      - "5000:5000"
    environment:
      - MONGO_URI=mongodb://mongo:27017/mydatabase
    depends_on:
      - mongo
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:5000/ || exit 1"]
      interval: 30s
      timeout: 10s
      retries: 5

  frontend:
    build: ./frontend
    container_name: frontend-container
    ports:
      - "3000:3000"
    depends_on:
      - backend
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:3000/ || exit 1"]
      interval: 30s
      timeout: 10s
      retries: 5

volumes:
  mongo-data:
```

- Docker Compose simplifies the management of a multi-container application consisting of a React frontend, an Express backend, and a MongoDB database.
- By defining services in a **docker-compose.yml** file, it specifies build contexts, port mappings, environment variables, and service dependencies.
- When executed with **docker-compose up**, it automatically builds and starts all containers, ensuring they can communicate over a shared network.
- This streamlines development, testing, and deployment, allowing for easy scaling and consistent environments.
- With Docker Compose, the entire application stack can be managed with a single command, enhancing efficiency and collaboration.

Automation Shell Script

Create a deploy.sh shell script in the root directory to automate the build, tag, push, and deploy steps.

```bash
#!/bin/bash

# Variables
FRONTEND_IMAGE="ys5sh/frontend"
BACKEND_IMAGE="ys5sh/backend"
TAG="02"

# Build frontend and backend images
echo "Building frontend and backend images..."
docker build -t $FRONTEND_IMAGE:$TAG ./frontend
docker build -t $BACKEND_IMAGE:$TAG ./backend

# Push images to Docker Hub
echo "Pushing images to Docker Hub..."
docker push $FRONTEND_IMAGE:$TAG
docker push $BACKEND_IMAGE:$TAG

# Run docker-compose
echo "Starting the application with Docker Compose..."
docker-compose up -d

echo "Application deployed successfully."

```

- The automation script, typically a shell script, streamlines the process of building, tagging, and deploying the Docker containers for the application.
- It begins by building Docker images for the frontend and backend using the specified Dockerfiles. Next, it tags these images with version numbers for better management.

- The script may then push the images to Docker Hub if configured. Finally, it updates the **docker-compose.yml** file with the new image versions and executes **docker-compose up** to start all services.
- This automation reduces manual steps, ensuring consistency and efficiency in deploying the application across environments.

Testing and Validation

- Run the application locally to confirm it's working:
  - Command: ***docker-compose up***
- Verify that each service is accessible:
  - **Frontend**: http://localhost:3000
  - **Backend**: http://localhost:5000
  - **MongoDB**: Confirm MongoDB is running on port 27017.
- **Expected Result**: The frontend should communicate with the backend, and the backend should successfully connect to MongoDB.
- Check the **logs**
  - Command: docker-compose logs

Conclusion

This project successfully demonstrated the process of containerizing a full-stack web application consisting of a React frontend, an Express backend, and a MongoDB database. By utilizing Docker and Docker Compose, each component of the application was isolated into its own container, making it more modular, scalable, and easier to deploy across diverse environments. Key objectives included setting up individual Dockerfiles for the frontend and backend, creating a Docker Compose file to manage multiple services, and automating the deployment pipeline with a shell script.

The containerization approach brings significant benefits:

- **Simplified Deployment**: Docker enables the application to run consistently across various platforms, eliminating the typical "works on my machine" issues.
- **Modularity**: Each component of the application is independently deployed in a separate container, allowing for greater flexibility in scaling and managing individual services.
- **Data Persistence**: Using volumes with Docker Compose ensures that MongoDB data is retained even if containers are restarted or redeployed.
- **Automation**: The shell script simplifies repetitive tasks by automating the build, tag, push, and deployment processes, saving time and reducing the risk of manual errors.

Through this project, we have created a reproducible and portable environment for the application, laying a solid foundation for continuous integration and deployment workflows. This setup not only supports current development needs but also makes it easier to extend the application in the future by adding more services or scaling individual components as needed.

Overall, this project highlights the advantages of containerization in modern web development, showcasing how Docker can streamline deployment, improve consistency, and enhance scalability for full-stack applications.

GitHub: ys5sh/DevOps-project (All project file, document and video)

Demo video: https://youtu.be/f_0St9MY2y4?si=T38l7KDVYxLpZi87