Docker Compose for multi Containers

Training Material

TABLE OF CONTENTS

Chapter 1: Introduction to Docker Compose

1.1 What is Docker Compose?

Docker Compose is a powerful tool designed to define and run multi-container Docker applications. It simplifies the process of managing complex applications by allowing you to define and run multiple services in a single configuration file. This eliminates the need for manual container management, making it easier to deploy and scale applications.

1.2 Why use Docker Compose?

- **Simplified Management:** Docker Compose allows you to manage multiple containers with a single command.
- **Consistent Environments:** Ensures consistent environments across development, testing, and production.
- **Faster Development:** Streamlines the development process by quickly building, starting, and stopping containers.
- **Scalability:** Easily scale your application by adding or removing containers.
- **Improved Collaboration:** Facilitates collaboration among team members by providing a clear and concise way to define and manage applications.

1.3 Basic Docker Compose file structure

A Docker Compose file, typically named docker-compose.yml, is a YAML file that defines the services, networks, and volumes for your application. A basic structure looks like this:

YAML:

```
version: '3.8'

services:
  web:
    build: .
    ports:
      - "5000:5000"
  db:
    image: mysql:latest
    environment:
      MYSQL_ROOT_PASSWORD: my-secret-pw
```

1.4 Defining services in a Docker Compose file

Each service in a Docker Compose file represents a Docker container. You can define the following properties for each service:

- **Image:** The Docker image to use for the container.
- **Build:** The path to the Dockerfile to build the image.

- **Ports:** Port mappings between the container and the host.
- **Volumes:** Volumes to mount to the container.
- **Environment variables:** Environment variables to set within the container.
- **Depends_on:** Services that the current service depends on.

Chapter 2: Running Multi-Container Applications

2.1 Starting and Stopping Services

To start all the services defined in your docker-compose.yml file, use the following command:

docker-compose up -d
The -d flag runs the containers in detached mode, allowing you to continue working in your terminal.

To stop all running containers, use:

docker-compose down

2.2 Scaling Services

You can scale the number of instances of a service using the scale command:

docker-compose scale web=2 db=3
This command will create two instances of the web service and three instances of the db service.

2.3 Viewing Logs

To view the logs of a specific service, use:

docker-compose logs <service_name>
To view the logs of all services, use:

docker-compose logs

Chapter 3: Networking in Docker Compose

3.1 Default Network

By default, Docker Compose creates a network for each project. This network allows services within the same project to communicate with each other without exposing ports to the host machine. This is a convenient way to set up basic networking for your application.

3.2 Creating Custom Networks

You can create custom networks to isolate services or to establish more complex networking topologies. To define a custom network, add a networks section to your docker-compose.yml file:

version: '3.8'

```
services:
  web:
    build: .
    networks:
      - my-network
  db:
    image: mysql:latest
    networks:
      - my-network
networks:
  my-network:
```

3.3 Connecting Services to Networks

To connect a service to a specific network, use the networks key in the service definition:

```
services:
  web:
    build: .
    networks:
      - my-network
```

By defining custom networks, you can create more flexible and isolated network configurations for your multi-container applications.

Chapter 4: Advanced Docker Compose Features

4.1 Volumes and Data Persistence

Volumes provide a way to persist data generated by containers. They are not part of the container's file system and can be mounted to multiple containers.

```
version: '3.8'
services:
  web:
    build: .
    volumes:
      - ./data:/app/data
```

4.2 Environment Variables

Environment variables can be used to configure services without hardcoding values in the Dockerfile.

```
version: '3.8'

services:

  web:

    build: .

    environment:

      APP_ENV: production

      DB_HOST: db
```

4.3 Secrets Management

Docker Compose allows you to securely manage sensitive information like passwords and API keys.

```
version: '3.8'

services:

  web:

    build: .

    environment:

      DB_PASSWORD: ${DB_PASSWORD}

secrets:

  DB_PASSWORD:

    file: ./secrets/db_password
```

4.4 Health Checks

Health checks allow you to monitor the health of your services and automatically restart unhealthy containers.

```
version: '3.8'

services:

  web:

    build: .

    healthcheck:

      test: ["CMD", "curl", "-f", "http://localhost:5000/health"]

      interval: 5s

      timeout: 5s

      retries: 3
```

4.5 Conditional Configuration

You can use profiles to conditionally configure services based on environment variables.

```
version: '3.8'

profiles:
  dev
  prod
services:
  web:
    build: .
    ports:
      - "5000:5000"
    environment:
      APP_ENV: ${APP_ENV:-dev}
```

Chapter 5: Best Practices for Docker Compose

5.1 Organizing Your Docker Compose Files

- Use multiple docker-compose.yml files for different environments.
- Organize services into logical groups.
- Use clear and concise naming conventions.

5.2 Using Multi-Stage Builds

Multi-stage builds can help reduce image size and improve build time.

Dockerfile:

```
# Stage 1: Build
FROM python:3.9-slim-buster AS builder
WORKDIR /app
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt

COPY . .

# Stage 2: Production Image
```

```
FROM python:3.9-slim-buster

WORKDIR /app

COPY --from=builder /app .

CMD ["python", "app.py"]
```

5.3 Optimizing Image Sizes

- Use a minimal base image.
- Minimize the number of layers in your Dockerfile.
- Clean up unnecessary files.
- Use multi-stage builds.

5.4 Security Considerations

- Use official images from trusted sources.
- Keep images up-to-date.
- Limit container privileges.
- Scan images for vulnerabilities.
- Use secrets to store sensitive information.

Chapter 6: Real-world Use Cases and Demos

6.1 Building a Web Application with a Database Backend

- Create a Docker Compose file to define the web and database services.
- Configure networking and data persistence.
- Set up environment variables for database credentials.

6.2 Deploying a Microservices Architecture

- Break down your application into smaller, independent services.
- Define each service in a Docker Compose file.
- Use Docker Compose to manage the deployment and scaling of services.

6.3 Creating a CI/CD Pipeline with Docker Compose

- Integrate Docker Compose with CI/CD tools like Jenkins or GitLab CI/CD.
- Automate the build, test, and deployment processes.

Hands-on Exercises

1. **Setting up a simple multi-container application:**
    - Create a Docker Compose file for a simple web application and a database.

- Configure the services, networks, and volumes.
- Start and stop the application.

2. **Configuring networks and volumes:**
    - Create a custom network for your application.
    - Mount a volume to persist data.
3. **Using environment variables and secrets:**
    - Set environment variables for database credentials.
    - Use secrets to store sensitive information securely.
4. **Implementing health checks:**
    - Define health checks for your services.
    - Monitor the health of your application.
5. **Deploying a multi-container application to a production environment:**
    - Use Docker Compose to deploy the application to a production server.
    - Configure production-specific settings.