Docker Compose

Introduction

Docker Compose is a powerful tool that allows developers to define and manage multi-container Docker applications. It uses YAML files to configure application services, which simplifies the entire process of running complex applications with multiple interacting services.

Basic Concepts

Before we jump into Docker Compose, let's first understand the following basic concepts:

Docker Image: This is a read-only template that includes set of instructions for creating a container. Images are created from Dockerfile.

Docker Container: A runnable instance of a Docker image.

Docker Service: In the context of Docker Compose, a service is an application, which can be made up of one or more containers.

Docker Compose File (docker-compose.yml): This file describes all the services that compose your app. Services in Docker Compose are similar to Docker containers, but they have additional features.

YAML

Introduction

YAML, which stands for "YAML Ain't Markup Language," is a human-readable data serialization language. It's commonly used for configuration files and in applications where data is being stored or transmitted.

Basic Structure

YAML supports two basic data structures: maps (also known as dictionaries or hashes) and lists (also known as arrays or sequences). Here is an example:

```
# Map
person:
  name: John Doe
  age: 30

# List
fruits:
  - Apple
  - Banana
  - Orange
```

In the map, 'person' is a key and the value is another map with keys 'name' and 'age'. In the list, 'fruits' is a key and the value is a list of items.
Indentation: In YAML, indentation is used to denote structure. It's similar to the way Python uses indentation. This makes the file easy to read and understand.

Comments: You can include comments in your YAML files to explain what's going on. This is done using the # symbol.

Data Types: YAML supports common data types like strings, booleans, floating-point numbers, and integers. It also supports null values, which can be represented with a ~ or null.

Combining YAML and Docker Compose:

Now, let's use our understanding of YAML to define a multi-service Docker application. Here's an example of a docker-compose.yml file:

```
version: '3'
services:
  app:
    build: ./app
    volumes:
      - ./app:/usr/src/app
    ports:
      - "8080:8080"
    depends_on:
      - db
  db:
    image: postgres
    volumes:
      - ./data/db:/var/lib/postgresql/data
```

In the above example:

We define two services, **app** and **db**.

The **app** service is built from the Dockerfile located in the **./app** directory.
The **./app:/usr/src/app** volume for the **app** service mounts the **./app** directory on the host to **/usr/src/app** in the container. This is beneficial during development where you want changes in the host to be reflected in the container immediately.

The **"8080:8080"** port mapping for the **app** service maps port 8080 in the container to port 8080 on the host machine.

The **depends_on** directive ensures that the **db** service starts before the **app** service.

The **db** service uses the official **postgres** image from Docker Hub.

The **./data/db:/var/lib/postgresql/data** volume for the **db** service is a named volume and is useful for persisting database data across container restarts.

With a solid understanding of Docker Compose and YAML, you can define complex multi-service applications that are easy to run and manage, saving you from the hassle of starting, linking, and stopping containers manually. By leveraging the power of Docker Compose and the simplicity of YAML, you can streamline your Docker workflows and focus more on developing your application.

The Docker Compose file is written in YAML and consists of several components to define and configure your application's services. Here's an explanation of common keywords and their meanings in the Docker Compose file:

1. version: This specifies the version of Docker Compose. The version depends on the features you want to use and your Docker Engine version.

2. services: This is the main component where you define your application's services, essentially all of your app's containers. Each service runs in a separate container.

Inside each service, you can specify a variety of options:

- **build**: This option can take two forms. It can either be a string specifying the path to the build context (e.g., **build: .**), or it can be a map with keys representing different build parameters (e.g., **context, dockerfile, args**).
- **image:** This allows you to specify the Docker image to be used for the container. You can either use locally built images or pull images from a registry like Docker Hub (e.g., **image: nginx:latest**).
- **command:** This overrides the default command for the Docker image.
- **container_name:** You can set a specific name for the container instead of a randomly generated one.
- **depends_on:** This option allows you to specify which services need to be started before the current one. Docker Compose will ensure the services listed are started first.
- **environment:** Here you can specify any environment variables that the service will use.
- **volumes:** This option is used for mounting volumes for the service, and it supports several different syntaxes.
- **ports:** This is used to map the container's ports to the host, allowing external access to the services in your containers.
- **networks:** This specifies which networks the service should be connected to.

3. networks: This key is used at the top level to define custom networks. You can specify driver, driver options, and other network-specific settings here.

4. volumes: This key is also used at the top level to define named volumes. Named volumes are managed by Docker and can be persisted across container restarts.

5. secrets: This option is used to manage sensitive data like passwords and API keys. It can be defined at the service level to grant access to specific services, or at the root level to define the secrets.

For example, consider this sample Docker Compose file:

```
version: '3'
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    image: my_web_app:latest
    container_name: my_web_app
    command: python app.py
    volumes:
      - .:/code
    ports:
      - "5000:5000"
    depends_on:
      - db
    environment:
      FLASK_ENV: development
  db:
    image: postgres:latest
    volumes:
      - postgres_data:/var/lib/postgresql/data/
networks:
  default:
    external:
      name: my_network
volumes:
  postgres_data:
    driver: local
```

In the above example, we're defining a web service that builds from a Dockerfile in the current directory and a database (db) service that uses the latest postgres image. The web service depends on the db service, exposes port 5000, and has an environment variable set. The db service has a volume mounted for data persistence. The external network my_network is used and a named volume postgres_data is defined.

Understanding the various components and syntax in Docker Compose file is important as it allows you to effectively define, run and manage multi-container Docker applications.

Docker Compose Commands

Some of the common commands in Docker Compose are:

docker-compose up: Starts and runs your entire app. If the image does not exist, Docker Compose will attempt to build it for you.

docker-compose down: Stops and removes containers, networks, volumes, and images created by up.

docker-compose build: Builds (or rebuilds) services.

docker-compose ps: Lists all running services.

docker-compose exec SERVICE COMMAND: Execute command in running service.

Docker Compose in Action

Now let's take an example of a Node.js application with a MongoDB database, and see how we can use Docker Compose to run it.

First, we create a **Dockerfile** for our Node.js application. The **Dockerfile** might look like this:

```
FROM node:10

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 8080

CMD [ "node", "server.js" ]
```

This Dockerfile will create a Docker image with our Node.js application.

Now let's see the **docker-compose.yml** file for our application:

```
version: '3'
services:
  app:
    build: .
    ports:
      - "8080:8080"
    depends_on:
      - db
    links:
      - db
  db:
    image: mongo
    ports:
      - "27017:27017"
```

Here, we have two services, **app** and **db**. The **app** service is our Node.js application, which is built using the current directory Dockerfile. The **db** service is our MongoDB database, and we use the standard **mongo** image from Docker Hub.

Now we can start our application using Docker Compose:

```
docker-compose up
```

This command will start both the **app** and **db** services. The **app** service can connect to the MongoDB database with the hostname **db**.

Networking in Docker Compose

Docker Compose automatically sets up a network and allows each service to communicate with others using the service name as hostname. For example, the Node.js application can connect to MongoDB using the hostname **db** (as per our previous **docker-compose.yml** file).

Docker Compose and Environment Variables

Docker Compose supports environment variables in the Compose file. These can be useful for managing different configurations for different environments (e.g., development, testing, production). An environment variable in a Compose file might look like this:

```
version: '3'
services:
  web:
    image: "my-web-app:${TAG}"
    environment:
      NODE_ENV: production
```

In this example, `${TAG}` is an environment variable that would be substituted with its value when the Compose file is processed.

Scaling with Docker Compose

Docker Compose allows you to scale your applications by using the --scale flag with the docker-compose up command. For example, to start 3 instances of the web service, you would run:

```
docker-compose up --scale web=3
```

Docker Compose and CI/CD

Docker Compose can be very useful in a CI/CD (Continuous Integration/Continuous Deployment) environment. Since it allows you to define your entire application environment in a single file, you can use Docker Compose to create identical environments for development, testing, and production.

Conclusion

In summary, Docker Compose is a powerful tool that can simplify the process of working with multi-container applications. By defining your application environment in a Compose file, you can ensure consistency across different stages of the development lifecycle, and make it easier to manage complex application architectures. Practice working with Docker Compose, experiment with different configurations, and you'll quickly become proficient in using this valuable tool.

Real Time Group

RT Embedded Linux Solutions

Docker Copmose Exercises

Exercise 1: Creating and Running a Simple Python Web App with Docker Compose
Application Code:
Create a simple Python Flask app with the following code and save it as app.py:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, Docker!'

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

Now, your task is to:

Write a Dockerfile that will create an image for the above Python Flask app.
Write a docker-compose.yml file to build and run the Docker image using Docker Compose.
Test if the application is running properly by accessing it from your local machine.

Exercise 2: Running a MySQL Database with Docker Compose

Database Code:
For this exercise, there is no specific code. We'll be using the MySQL Docker image.

Your task:

Write a docker-compose.yml file that will pull the MySQL image and run a MySQL container.
Set the environment variables for **MYSQL_ROOT_PASSWORD, MYSQL_DATABASE, MYSQL_USER, and MYSQL_PASSWORD**.
Connect to the MySQL database from your local machine and test if it's working properly.

Exercise 3: Connecting Python Flask App with MySQL Database using Docker Compose

Application Code:
Let's expand on the first Python Flask app. This new Flask app will connect to a MySQL database and display whether the connection was successful. Save it as app.py:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
import os

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get('DB_URI')
db = SQLAlchemy(app)

@app.route('/')
def hello_world():
    try:
        db.session.query("1").from_statement("SELECT 1").all()
        return 'Hello, Docker! Database connection successful.'
    except Exception as e:
        return f'An error occurred: {str(e)}'

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

Now, your task is to:

Update the Dockerfile if necessary.
Write a docker-compose.yml file that will build and run both the Flask app and MySQL containers.
Set the environment variables for the MySQL service and also set the DB_URI environment variable in the Flask app service, which is the SQLAlchemy database URI.
Test if the application and database are running and communicating properly by accessing the Flask app from your local machine.

Exercise 4: Building and Running a Node.js Web App with Docker Compose

Application Code:
Let's create a simple Node.js Express app. Save the following code as app.js:

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello Docker!')
})

app.listen(port, () => {
  console.log(`App running at http://localhost:${port}`)
})
```

And a package.json:

```
{
  "name": "docker-node-app",
  "version": "1.0.0",
  "description": "Node.js on Docker",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.16.1"
  }
}
```

Your task is to:

Write a Dockerfile that will create an image for the above Node.js Express app.
Write a docker-compose.yml file to build and run the Docker image using Docker Compose.
Test if the application is running properly by accessing it from your local machine.

Exercise 5: Connecting Node.js App with MongoDB Database using Docker Compose

Application Code:
Let's expand on the previous Node.js Express app. This new Express app will connect to a MongoDB database and display whether the connection was successful. Update app.js as follows:

```
const express = require('express')
const mongoose = require('mongoose')
const app = express()
const port = 3000

mongoose.connect('mongodb://mongo:27017/test', { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log('MongoDB Connected...'))
  .catch(err => console.log(err))

app.get('/', (req, res) => {
  res.send('Hello Docker! MongoDB connection was successful.')
})

app.listen(port, () => {
  console.log(`App running at http://localhost:${port}`)
})
```

Also, add mongoose to the package.json file:

```
{
  "name": "docker-node-app",
  "version": "1.0.0",
  "description": "Node.js on Docker",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.16.1",
    "mongoose": "^5.13.7"
  }
}
```

Your task is to:
Update the Dockerfile if necessary.
Write a docker-compose.yml file that will build and run both the Node.js app and MongoDB containers.
Test if the application and database are running and communicating properly by accessing the Node.js app from your local machine.

Exercise 6: Docker Compose with a Multi-Container Flask and Redis Application

Application Code:
Let's create a simple Python Flask application that increments a counter stored in a Redis database. Here is the Python application code (save it as app.py):

```
from flask import Flask
from redis import Redis

app = Flask(__name__)
redis = Redis(host='redis', port=6379)

@app.route('/')
def hello():
    count = redis.incr('hits')
    return 'Hello Docker! I have been seen {} times.\n'.format(count)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

Your task is to:

Write a Dockerfile to containerize this Python application.
Write a docker-compose.yml file to create a multi-container application with Docker Compose. It should create and link two services: the Flask app and a Redis database.
Test the application to ensure that the counter increments each time you refresh the page.

Exercise 7: Multi-service Docker Compose Application with Networking

Application Code:
Now let's create a Node.js Express app that fetches data from a PostgreSQL database. Both services will be defined in Docker Compose and the Node.js app will connect to the PostgreSQL service over the network.

First, we'll need a Node.js Express application. Here is the app.js file:

```
const express = require('express');
const app = express();
const port = 3000;
const { Pool } = require('pg');
const pool = new Pool({
  user: 'postgres',
  host: 'db',
  database: 'test',
  password: 'password',
  port: 5432,
});

app.get('/', async (req, res) => {
  const client = await pool.connect();
  const result = await client.query('SELECT * FROM visits');
  const currentVisit = result.rows[0].count;
  client.query('UPDATE visits SET count = count + 1');
  client.release();

  res.send(`Hello Docker! I have been visited ${currentVisit} times.`);
});

app.listen(port, () => {
  console.log(`App running on http://localhost:${port}`)
});
```

Next, create an initialization script for PostgreSQL to create a table named 'visits'. This script will be placed in a directory named 'db-scripts'. Here is the init.sql file:

```
CREATE TABLE IF NOT EXISTS visits (
  count integer
);

INSERT INTO visits VALUES (0);
```

Your task is to:

Write a Dockerfile to containerize the Node.js application.
Write a docker-compose.yml file to create a multi-service application with Docker Compose. This should define and link the Node.js app and PostgreSQL services.
The PostgreSQL service should use an initialization script to create the necessary database and table.
Test the application to ensure that the visit count increments each time you refresh the page.

Exercise 8: Docker Compose with Frontend and Backend

Application Code:

Let's create a simple Python Flask backend that returns a message and a React.js frontend that displays this message.

Here is the Python backend code (save it as app.py):

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/')
def hello():
    return jsonify(message='Hello Docker!')

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

And here is the React.js frontend code (save it as App.js):

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  state = { message: "" };

  componentDidMount() {
    fetch('/api')
      .then(res => res.json())
      .then(data => this.setState({ message: data.message }));
  }

  render() {
    return (
      <div className="App">
        <header className="App-header">
          <p>{this.state.message}</p>
        </header>
      </div>
    );
  }
}

export default App;
```

Your task is to:

Write Dockerfiles to containerize both the Python Flask backend and the React.js frontend.
Write a docker-compose.yml file to create a multi-container application with Docker Compose. It should create and link both services.
Test the application to ensure that the React.js frontend displays the message from the Python Flask backend.

Exercise 9: Docker Compose with Custom Network

We'll create a custom network with Docker Compose and add a PostgreSQL database and an Express.js backend to this network.

Here is the Express.js backend code (save it as app.js):

```
const express = require('express');
const { Pool } = require('pg');
const app = express();

const pool = new Pool({
  user: 'postgres',
  host: 'db',
  database: 'test',
  password: 'password',
  port: 5432,
});

app.get('/', async (req, res) => {
  const client = await pool.connect();
  const result = await client.query('SELECT NOW()');
  const currentTime = result.rows[0].now;
  client.release();

  res.send(`Hello Docker! The current time is ${currentTime}.`);
});

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

Your task is to:
Write a Dockerfile to containerize the Express.js backend.
Write a docker-compose.yml file to create a custom network and add the Express.js backend and PostgreSQL database to this network. You should define and link both services.
Test the application to ensure that the Express.js backend can fetch the current time from the PostgreSQL database.

Exercise 10: Docker Compose with Environment Variables

In this exercise, you'll work with a Python Flask application that uses environment variables.

Here is the Python application code (save it as app.py):

```
from flask import Flask
import os

app = Flask(__name__)

@app.route('/')
def hello():
    name = os.getenv("NAME", "Docker")
    return f'Hello {name}!'

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

Your task is to:

Write a Dockerfile to containerize this Python application.
Write a docker-compose.yml file that sets the NAME environment variable to a value of your choice.
Test the application to ensure that the Python Flask app displays the message with the name you set.

Exercise 11: Docker Compose with Volumes

We'll create a Node.js Express app that writes data to a file and a volume to persist this data.

Here is the Node.js app code (save it as app.js):

```
const express = require('express');
const fs = require('fs');
const app = express();
const dataFile = '/data/data.txt';

app.get('/', (req, res) => {
  let data = 'Hello Docker!';
  fs.writeFileSync(dataFile, data);
  res.send(data);
});

app.get('/data', (req, res) => {
  let data = fs.readFileSync(dataFile, 'utf8');
  res.send(data);
});

app.listen(3000, () => console.log('App is listening on port 3000'));
```

Your task is to:

Write a Dockerfile to containerize the Node.js application.
Write a docker-compose.yml file to define a volume that is mounted to the "/data" directory in the container.
Test the application by writing data and then reading it back from the volume.

Real Time Group

RT Embedded Linux Solutions