Building and Deploying a Python Microservice with Docker

Objective:

Learn to develop a simple Python application, transform it into a microservice, and deploy it using Docker and Docker Compose with an additional service like a database or Redis.

This lab guides you through creating a Python Flask application with CRUD operations, refactoring it into microservices, and deploying these services using Docker and Docker Compose. You'll start with a monolithic app, split it into Read and Write services, and later add Notification and Logging services for inter-service communication.

1. Create a simple python flask application

You will develop a basic Python Flask application with CRUD functionalities using an SQLite database.

"SQLite is chosen for simplicity and ease of setup, as SQLite doesn't require a separate database server and is straightforward to use for small-scale applications or development purposes. Later, the configuration will be switched to MySQL when you are implementing Microservice."

Requirements: Python, Flask, SQLAlchemy, Docker

Later stage: Docker Compose, MySQL, pymysql, Postman (API Testing)

Some commands to use:

- pip install Flask
- pip install SQLAlchemy
- pip install pymysql

sample code (You may need to modify based on your environment)

file: app.py

```
from flask import Flask, request, jsonify

from flask_sqlalchemy import SQLAlchemy


app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] =
'sqlite:///:memory:'

db = SQLAlchemy(app)


class Item(db.Model):

    id = db.Column(db.Integer, primary_key=True)

    name = db.Column(db.String(80), nullable=False)


@app.before_first_request

def create_tables():

    db.create_all()


@app.route('/items', methods=['POST', 'GET'])
```

```
def handle_items():
    if request.method == 'POST':
        data = request.json
        new_item = Item(name=data['name'])
        db.session.add(new_item)
        db.session.commit()
        return jsonify({'id': new_item.id, 'name':
new_item.name}), 201

    items = Item.query.all()
    return jsonify([{'id': item.id, 'name': item.name} for
item in items])

@app.route('/items/<int:item_id>', methods=['PUT',
'DELETE'])
def handle_item(item_id):
    item = Item.query.get_or_404(item_id)

    if request.method == 'PUT':
        data = request.json
        item.name = data['name']
        db.session.commit()
        return jsonify({'id': item.id, 'name': item.name})

    db.session.delete(item)
    db.session.commit()
    return jsonify({}), 204

if __name__ == '__main__':
    app.run(debug=True)
```

- Run the app: python app.py.
- Test crud operation using Postman.

Current file structure:

```
/my_flask_app
│
└── app.py
```

This code is a simple yet complete example of a web application built using Flask, a lightweight web framework in Python, and SQLAlchemy, an Object-Relational Mapping (ORM) tool. The application provides a basic CRUD (Create, Read, Update, Delete) API for managing Item objects.

API Endpoints:

1. Handle Items (/items):
   - The route /items accepts both GET and POST requests.
   - POST Request: Used to create a new item. The item's name is extracted from the request JSON, a new Item object is created, added to the database, and committed. It returns the new item's ID and name in JSON format.
   - GET Request: Retrieves all items from the database and returns them in JSON format.

2. Handle a Single Item (/items/<int:item_id>):

   - The route /items/<int:item_id> accepts PUT and DELETE requests for a specific item identified by its item_id.

- PUT Request: Used to update the name of an existing item. The function finds the item by ID, updates its name, and saves the changes to the database.
- DELETE Request: Used to delete an item. The function finds the item by ID and removes it from the database.

3. Running the App:

if __name__ == '__main__': app.run(debug=True): This block runs the Flask application in debug mode when the script is executed directly.

2. Split the Application into Microservices

Objective:

Refactor the monolithic application into two separate microservices: Read Service and Write Service.

Instructions:

- Create Read Service (read_service.py): Extract the read (GET) functionality into a separate Flask application.
- Create Write Service (write_service.py): Extract the create (POST), update (PUT), and delete (DELETE) functionalities into another Flask application.
- Run and Test Both Services: Ensure both services are functioning correctly on their own.

```
/my_microservices_app
│
├── read_service/
│   └── read_service.py
│
└── write_service/
    └── write_service.py
```

read_service.py

```python
from flask import Flask, jsonify
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql+pymysql://user:password@127.0.0.1:3306/mydatabase'
db = SQLAlchemy(app)

class Item(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable=False)

@app.route('/items', methods=['GET'])
def get_items():
    items = Item.query.all()
    return jsonify([{'id': item.id, 'name': item.name} for item in items])

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

Dockerfile:

```dockerfile
FROM python:3.12
WORKDIR /app
COPY ./read_service /app
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 5000
CMD ["python", "read_service.py"]
```