Expert Cloud Consulting
Enhance Optimise & Scale
ASCP GPUonCLOUD Pvt Ltd

“Expert Cloud Consulting” -

SOP | Docker Containerization

03.January.2025

version 1.0

—

Contributed by Tejal Kale
Approved by Akshay Shinde
Expert Cloud Consulting
Office #811, Gera Imperium Rise,
Hinjewadi Phase-II Rd, Pune, India – 411057

“Expert Cloud Consulting”
Docker Containerization

1.0 Contents

Expert Cloud Consulting
Enhance Optimise & Scale
ASCP GPUonCLOUD Pvt Ltd

2.0 General Information:

2.1 Document Purpose

This document introduces the fundamentals of Docker containerization, emphasizing automation and efficient management of application environments. It includes hands-on assignments to containerize applications, set up multi-container architectures using Docker Compose, and deploy workloads to production environments. The purpose is to equip users with practical skills in Docker for building scalable, portable, and repeatable application deployments.

2.2 Document Revisions

Date	**Version**	**Contributor(s)**	**Approver(s)**	**Section(s)**	**Change(s)**
03/Jan/2025	1.0	Tejal Kale	Akshay Shinde	All Sections	New Document Created

3.0 Document Overview:

Containerization with Docker is a modern approach to building, deploying, and managing applications in lightweight, portable containers. Docker simplifies application development by allowing developers to package an application and its dependencies into a single, self-sufficient unit. This document provides a comprehensive guide to using Docker for containerizing applications, setting up multi-container environments with Docker Compose, and deploying containerized workloads to production. It also includes best practices for optimizing Docker images, managing container security, and leveraging Docker Hub for efficient application distribution.

4.0 Project Overview:

4.1 Architecture

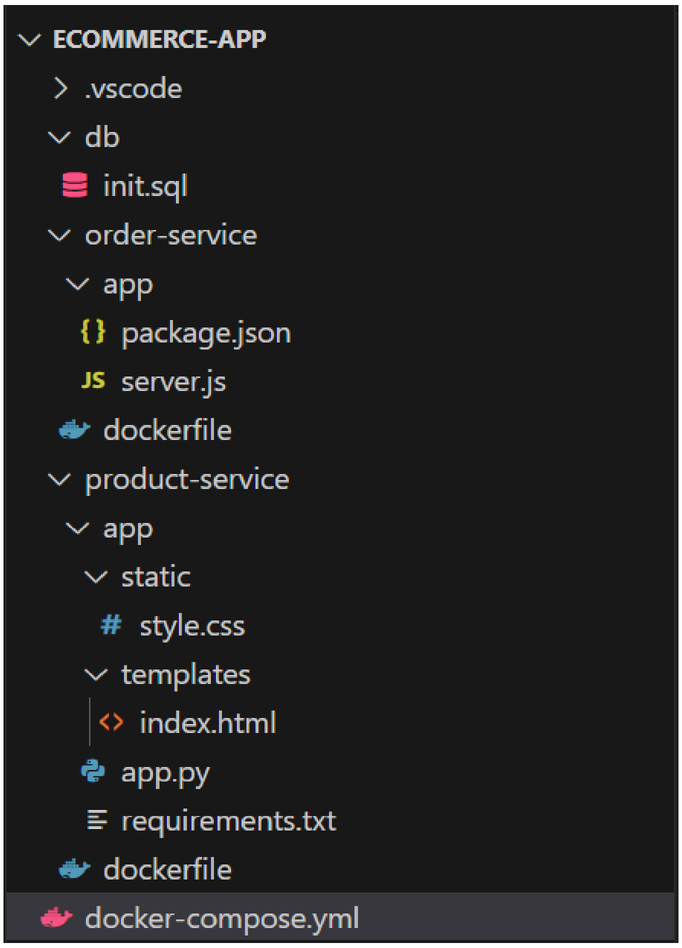
The application consists of three main components:

- Product Service (Python/Flask)
- Order Service (Node.js)
- Shared MySQL Database

4.2 Prerequisites

- Docker Desktop
- Visual Studio Code
- Python 3.9+
- Node.js 16+

4.3 Project Structure



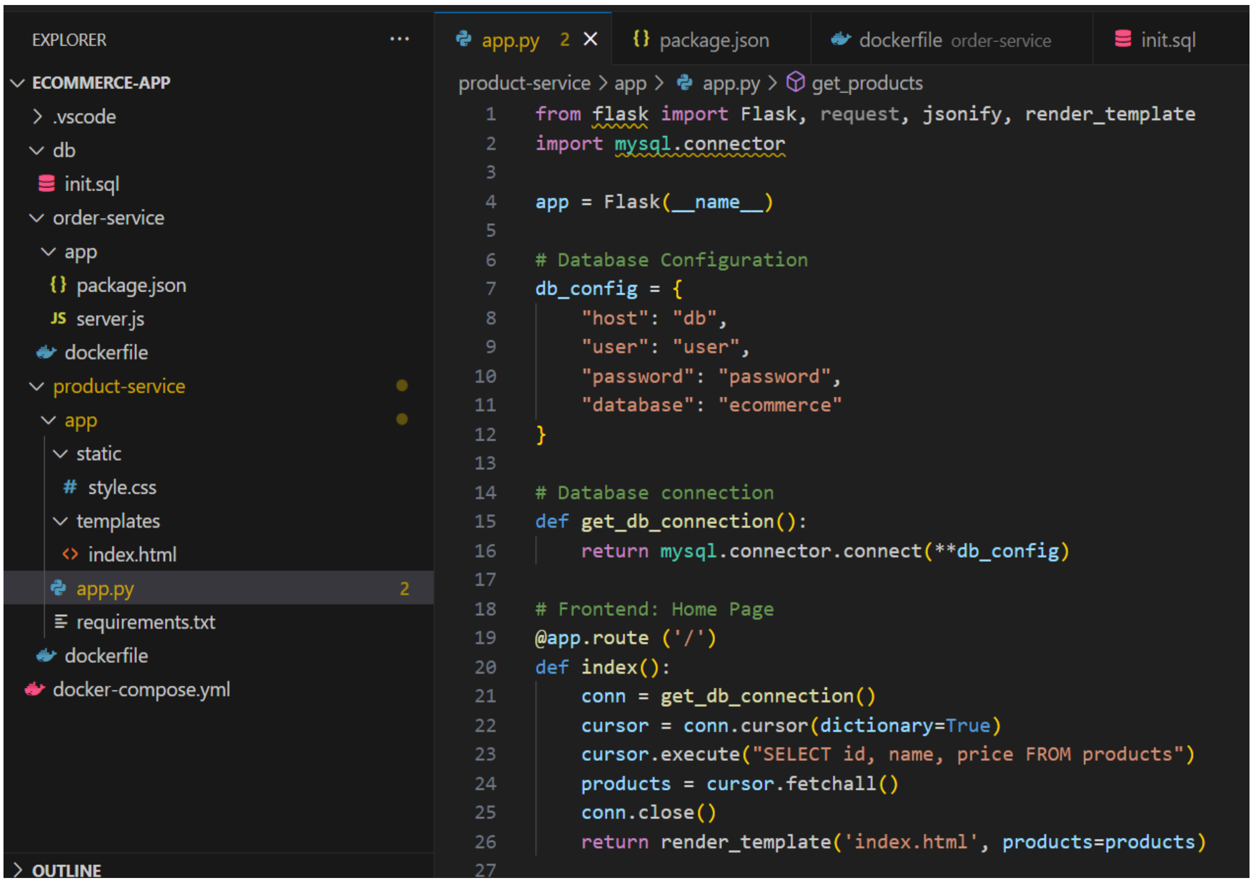
```
ECOMMERCE-APP
  > .vscode
  v db
     init.sql
  v order-service
    v app
       package.json
       server.js
     dockerfile
  v product-service
    v app
      v static
         style.css
      v templates
         index.html
       app.py
       requirements.txt
     dockerfile
   docker-compose.yml
```

5.0 Product Service (Python/Flask):

5.1 Application Setup:

Create product-service/app.py:

The product-service/app.py file is likely the entry point for your service. The file starts by importing necessary libraries and modules, such as the framework (e.g., Flask or)and any custom modules. Defines the API routes where the service handles specific HTTP requests. Specifies how the app starts running (e.g., through app.run()).

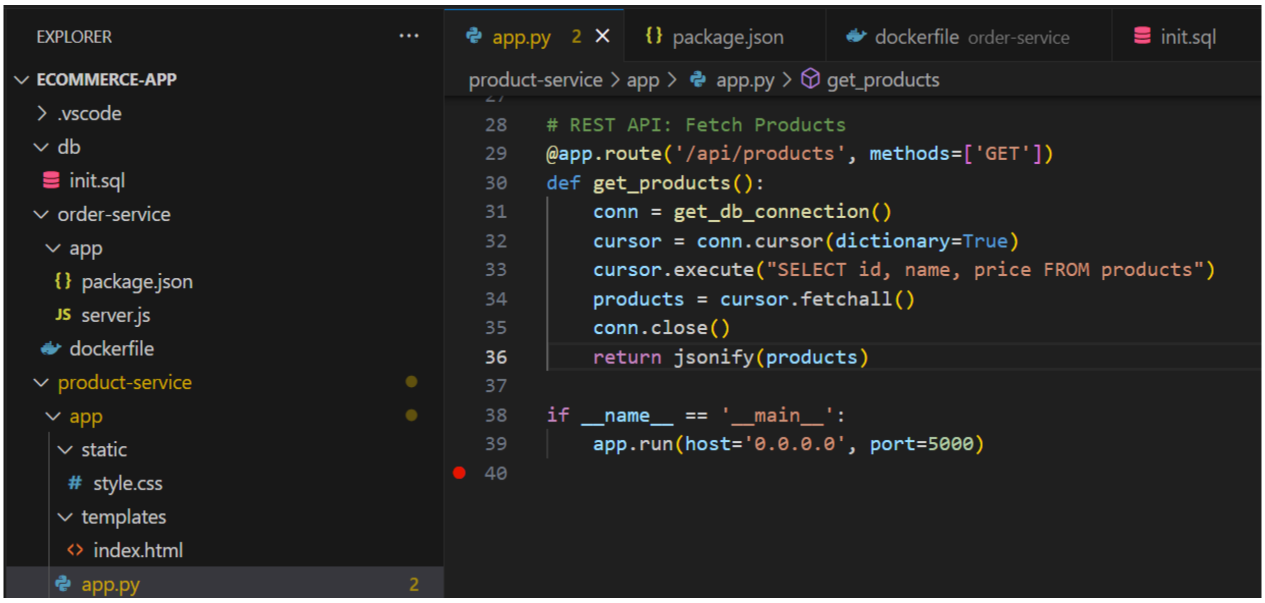


```
EXPLORER
ECOMMERCE-APP
  > .vscode
  v db
      init.sql
  v order-service
    v app
        package.json
        server.js
      dockerfile
  v product-service
    v app
      v static
          style.css
      v templates
          index.html
        app.py  2
        requirements.txt
      dockerfile
    docker-compose.yml
> OUTLINE
```

app.py 2 × | package.json | dockerfile order-service | init.sql

product-service > app > app.py > get_products

```python
from flask import Flask, request, jsonify, render_template
import mysql.connector

app = Flask(__name__)

# Database Configuration
db_config = {
    "host": "db",
    "user": "user",
    "password": "password",
    "database": "ecommerce"
}

# Database connection
def get_db_connection():
    return mysql.connector.connect(**db_config)

# Frontend: Home Page
@app.route ('/')
def index():
    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)
    cursor.execute("SELECT id, name, price FROM products")
    products = cursor.fetchall()
    conn.close()
    return render_template('index.html', products=products)
```



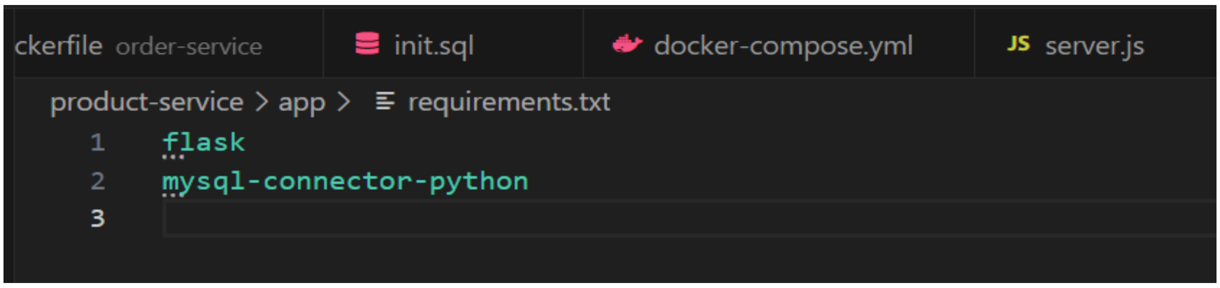
app.py 2 × | package.json | dockerfile order-service | init.sql

product-service > app > app.py > get_products

```python
# REST API: Fetch Products
@app.route('/api/products', methods=['GET'])
def get_products():
    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)
    cursor.execute("SELECT id, name, price FROM products")
    products = cursor.fetchall()
    conn.close()
    return jsonify(products)

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

5.2 product-service/requirements.txt:

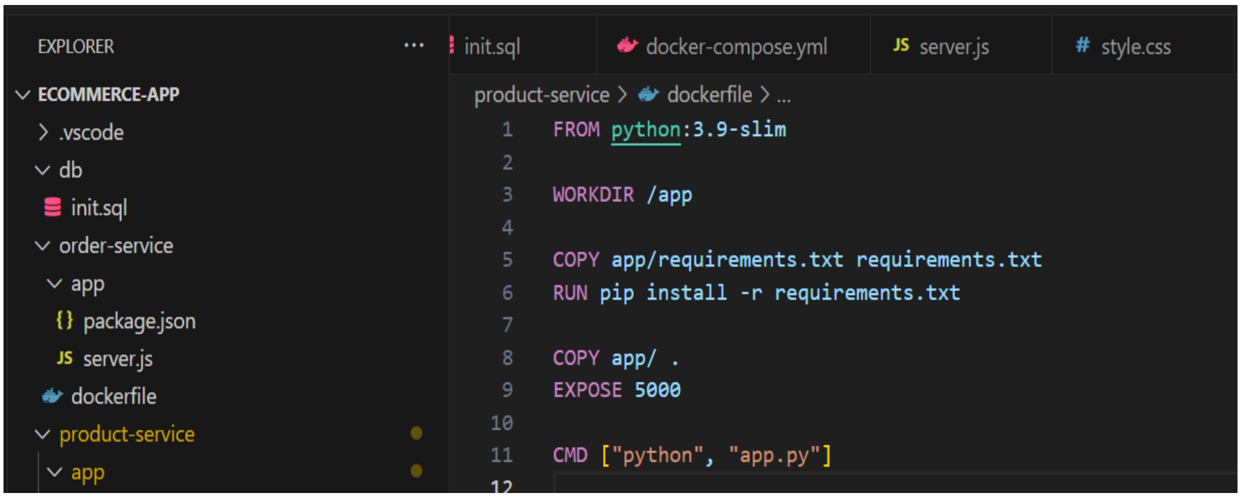
The requirements.txt file is a standard way to list the dependencies your Python application needs. It allows others (or deployment environments) to install the exact versions of the libraries used in your project by running pip install -r requirements.txt.



```
flask
mysql-connector-python
```

5.3 product-service/Dockerfile:

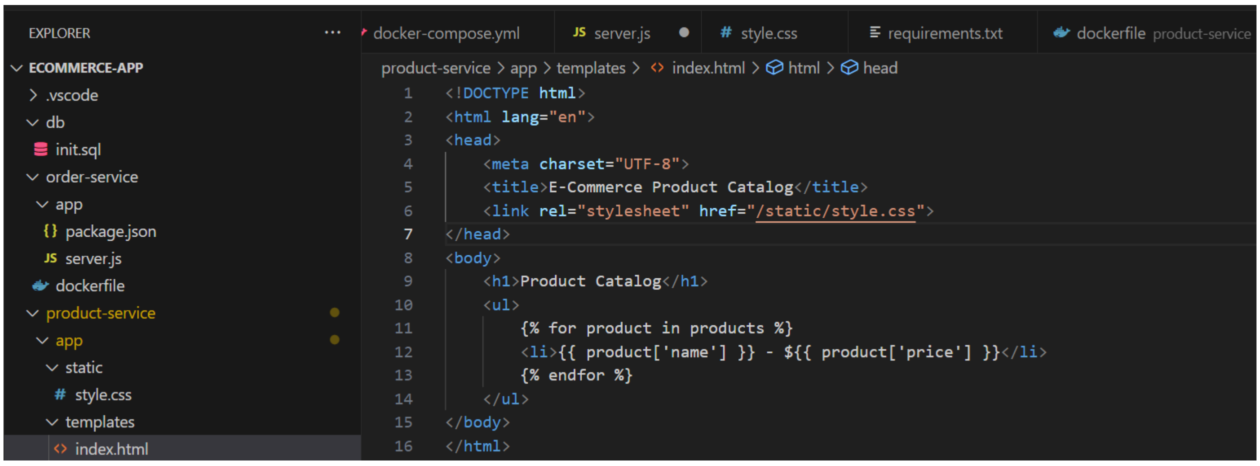
The Dockerfile is a script that contains instructions to build a Docker image for your application. For a product-service Python application, the Dockerfile would define how to package your code, dependencies, and configurations into a lightweight, portable container.



```
FROM python:3.9-slim

WORKDIR /app

COPY app/requirements.txt requirements.txt
RUN pip install -r requirements.txt

COPY app/ .
EXPOSE 5000

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

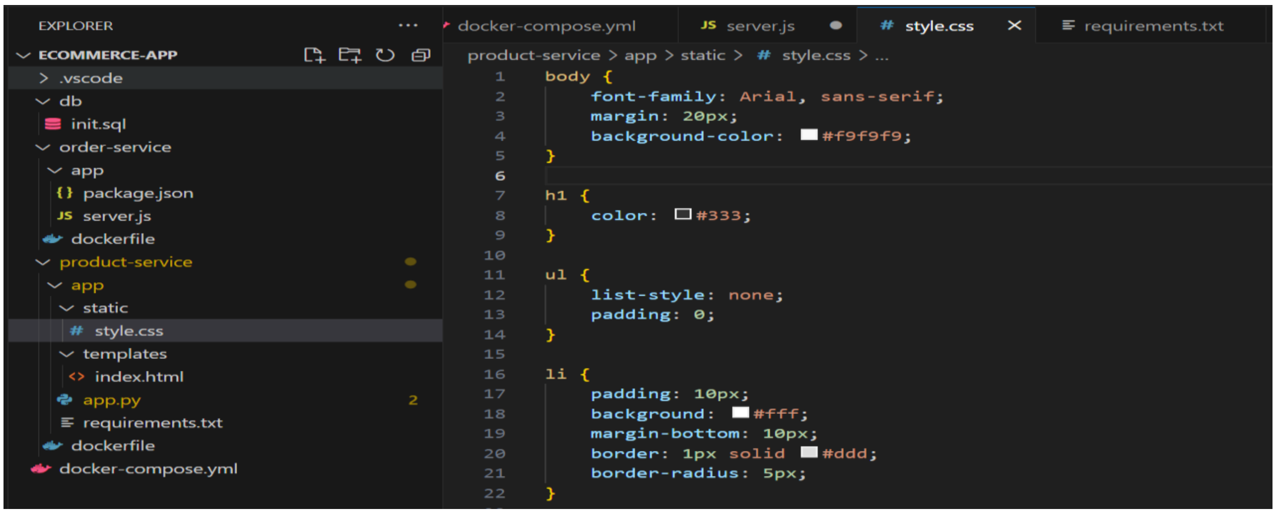
5.4 product-service/templates/index.html :

The product-service/templates/index.html file is likely an HTML template used in a web application, typically for rendering a user interface related to the product service.



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>E-Commerce Product Catalog</title>
    <link rel="stylesheet" href="/static/style.css">
</head>
<body>
    <h1>Product Catalog</h1>
    <ul>
        {% for product in products %}
        <li>{{ product['name'] }} - ${{ product['price'] }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

5.5 product-service/static/style.css

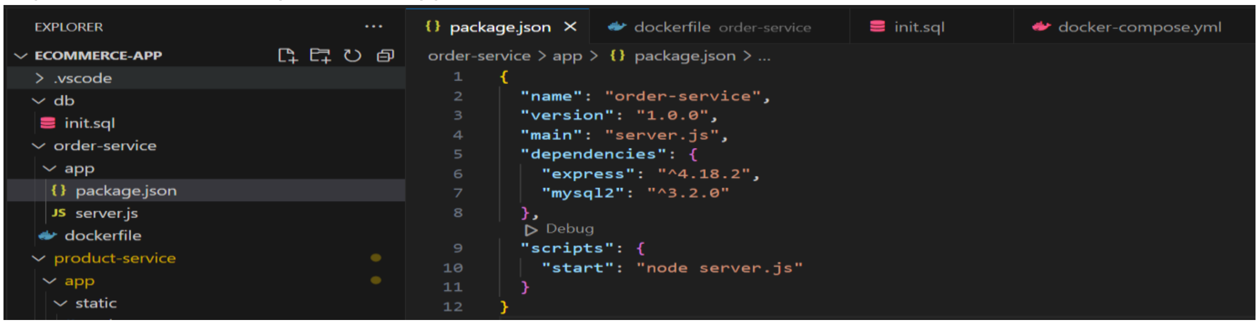


```css
body {
    font-family: Arial, sans-serif;
    margin: 20px;
    background-color: #f9f9f9;
}

h1 {
    color: #333;
}

ul {
    list-style: none;
    padding: 0;
}

li {
    padding: 10px;
    background: #fff;
    margin-bottom: 10px;
    border: 1px solid #ddd;
    border-radius: 5px;
}
```

6.0 Order Service (Node.js)

6.1 Order-service/package.json:

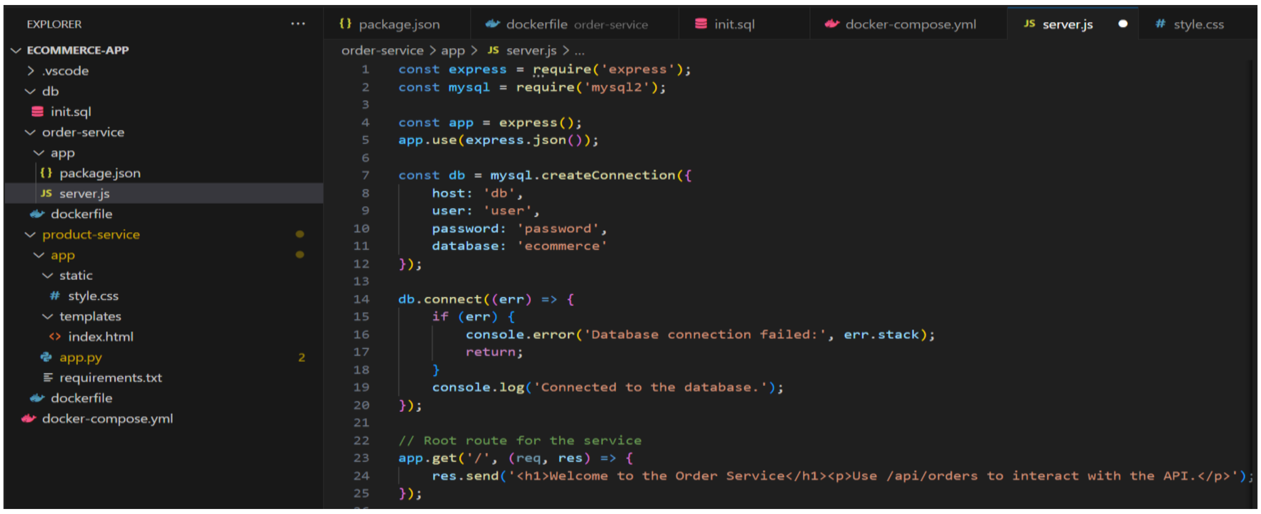
The package.json file is a configuration file used in Node.js projects to define the metadata, dependencies, and scripts for the application.



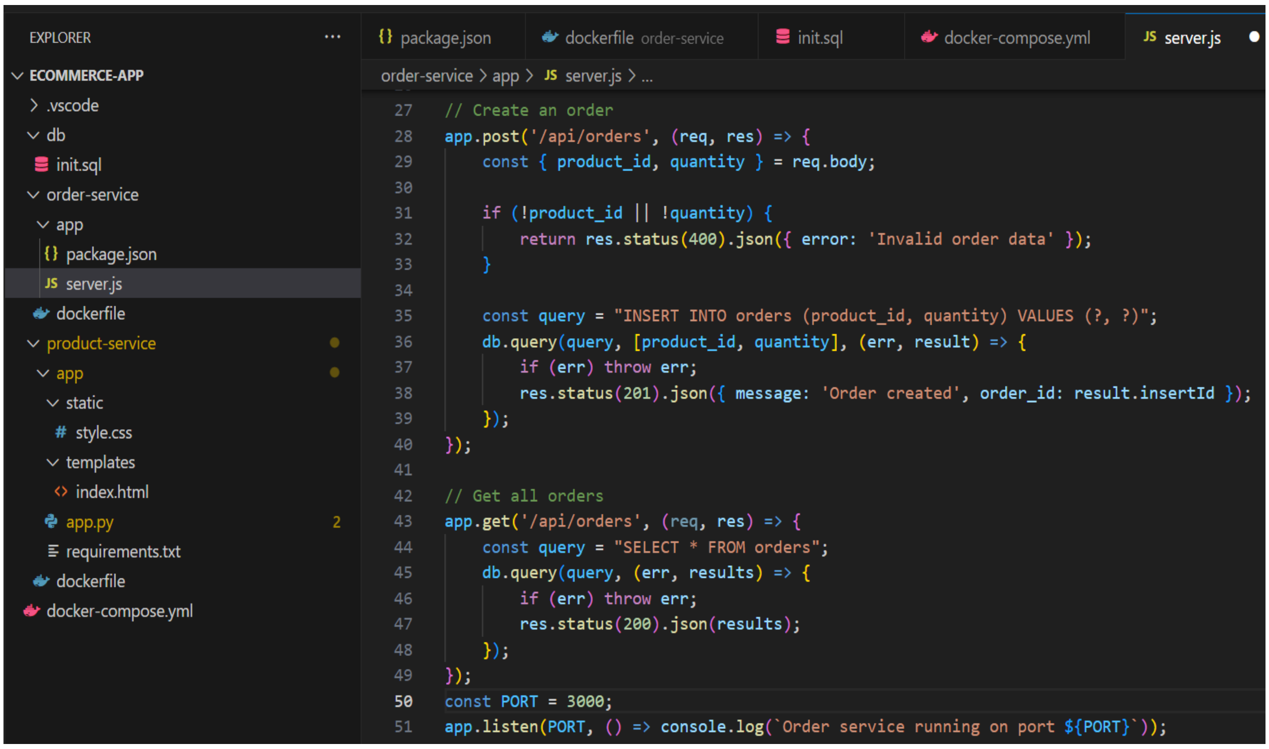
```json
{
  "name": "order-service",
  "version": "1.0.0",
  "main": "server.js",
  "dependencies": {
    "express": "^4.18.2",
    "mysql2": "^3.2.0"
  },
  "scripts": {
    "start": "node server.js"
  }
}
```

6.2 order-service/server.js

The server.js file in a Node.js application typically serves as the entry point for the application. It is responsible for setting up the server, configuring middleware, defining routes, and starting the application to listen for incoming requests.



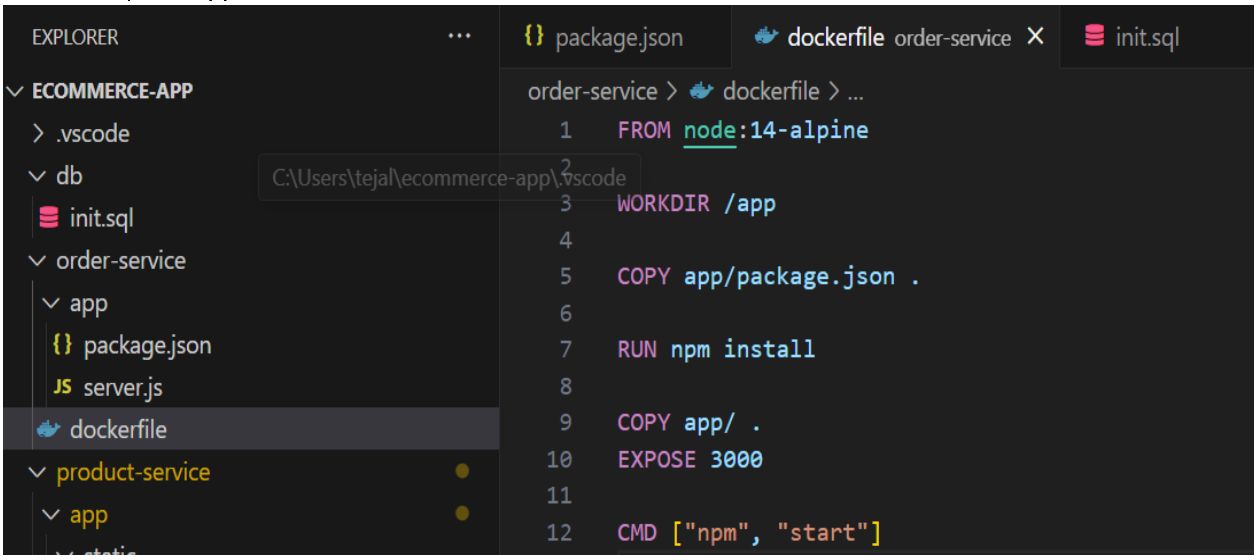
```javascript
const express = require('express');
const mysql = require('mysql2');

const app = express();
app.use(express.json());

const db = mysql.createConnection({
    host: 'db',
    user: 'user',
    password: 'password',
    database: 'ecommerce'
});

db.connect((err) => {
    if (err) {
        console.error('Database connection failed:', err.stack);
        return;
    }
    console.log('Connected to the database.');
});

// Root route for the service
app.get('/', (req, res) => {
    res.send('<h1>Welcome to the Order Service</h1><p>Use /api/orders to interact with the API.</p>');
});
```



```javascript
// Create an order
app.post('/api/orders', (req, res) => {
    const { product_id, quantity } = req.body;

    if (!product_id || !quantity) {
        return res.status(400).json({ error: 'Invalid order data' });
    }

    const query = "INSERT INTO orders (product_id, quantity) VALUES (?, ?)";
    db.query(query, [product_id, quantity], (err, result) => {
        if (err) throw err;
        res.status(201).json({ message: 'Order created', order_id: result.insertId });
    });
});

// Get all orders
app.get('/api/orders', (req, res) => {
    const query = "SELECT * FROM orders";
    db.query(query, (err, results) => {
        if (err) throw err;
        res.status(200).json(results);
    });
});
const PORT = 3000;
app.listen(PORT, () => console.log(`Order service running on port ${PORT}`));
```

6.3 order-service/dockerfile:

The Dockerfile for the order-service defines how to build a Docker image for your Node.js application. This file specifies the base image, installs dependencies, copies application files, and sets up the application to run in a container.

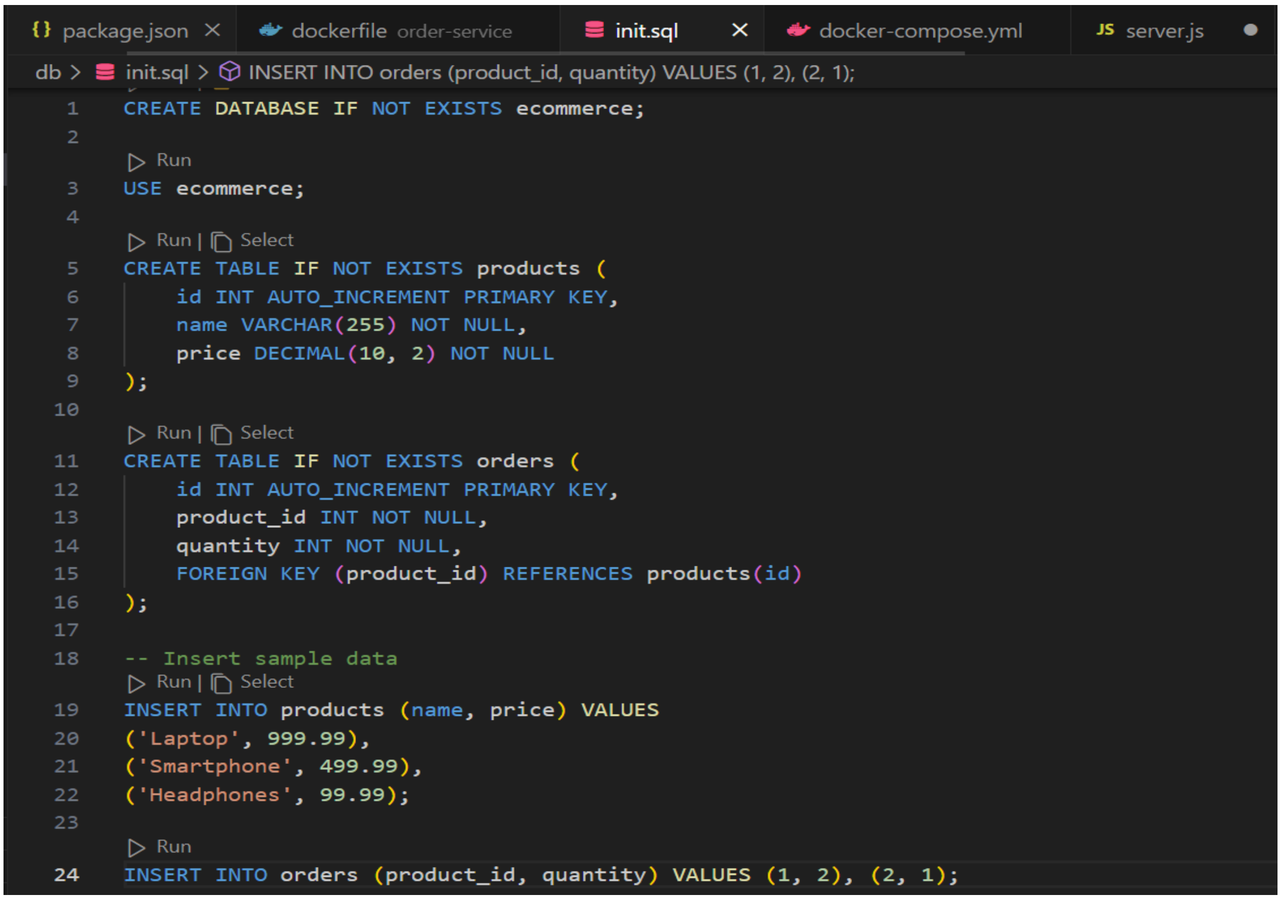


```dockerfile
FROM node:14-alpine

WORKDIR /app

COPY app/package.json .

RUN npm install

COPY app/ .
EXPOSE 3000

CMD ["npm", "start"]
```

7.0 Database Setup (MySQL)

7.1 db/init.sql:

The db/init.sql file is typically used to initialize a database by defining its structure and sometimes inserting initial data. It contains SQL statements for creating tables.

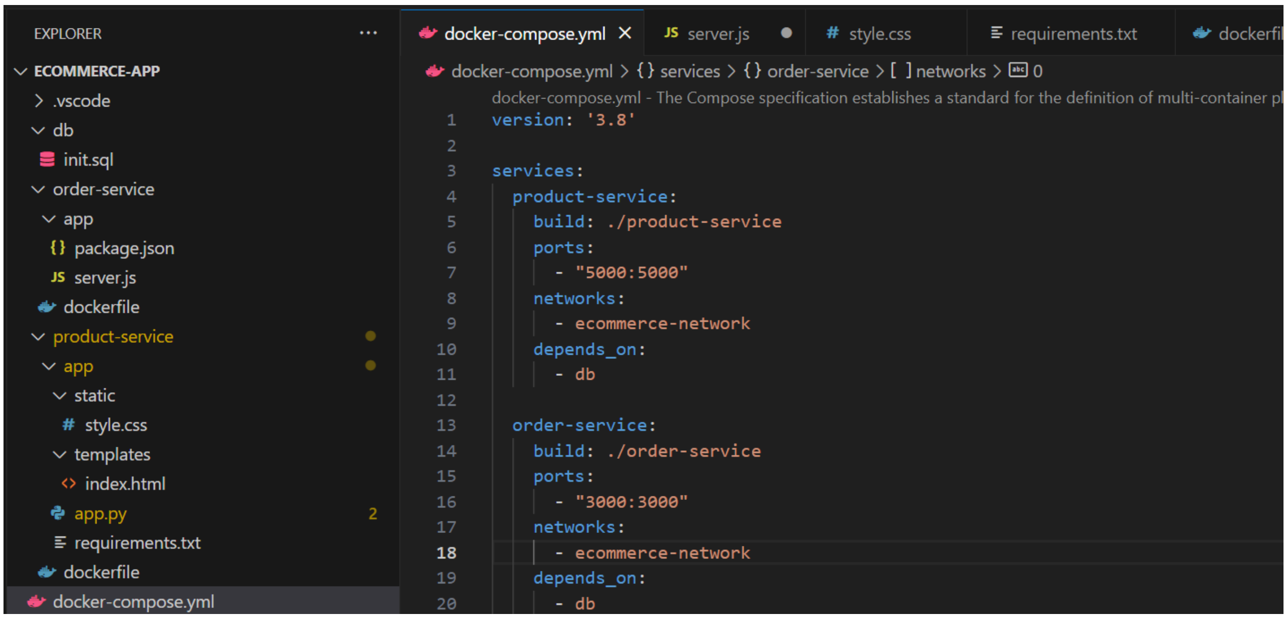


```sql
CREATE DATABASE IF NOT EXISTS ecommerce;

USE ecommerce;

CREATE TABLE IF NOT EXISTS products (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    price DECIMAL(10, 2) NOT NULL
);

CREATE TABLE IF NOT EXISTS orders (
    id INT AUTO_INCREMENT PRIMARY KEY,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    FOREIGN KEY (product_id) REFERENCES products(id)
);

-- Insert sample data
INSERT INTO products (name, price) VALUES
('Laptop', 999.99),
('Smartphone', 499.99),
('Headphones', 99.99);

INSERT INTO orders (product_id, quantity) VALUES (1, 2), (2, 1);
```

8.0 Docker Compose Configuration

8.1 docker-compose.yml

The docker-compose.yml file is a configuration file used by Docker Compose to define and manage multi-container Docker applications. It specifies how services (containers), networks, and volumes work together for your application.



```yaml
version: '3.8'

services:
  product-service:
    build: ./product-service
    ports:
      - "5000:5000"
    networks:
      - ecommerce-network
    depends_on:
      - db

  order-service:
    build: ./order-service
    ports:
      - "3000:3000"
    networks:
      - ecommerce-network
    depends_on:
      - db
```

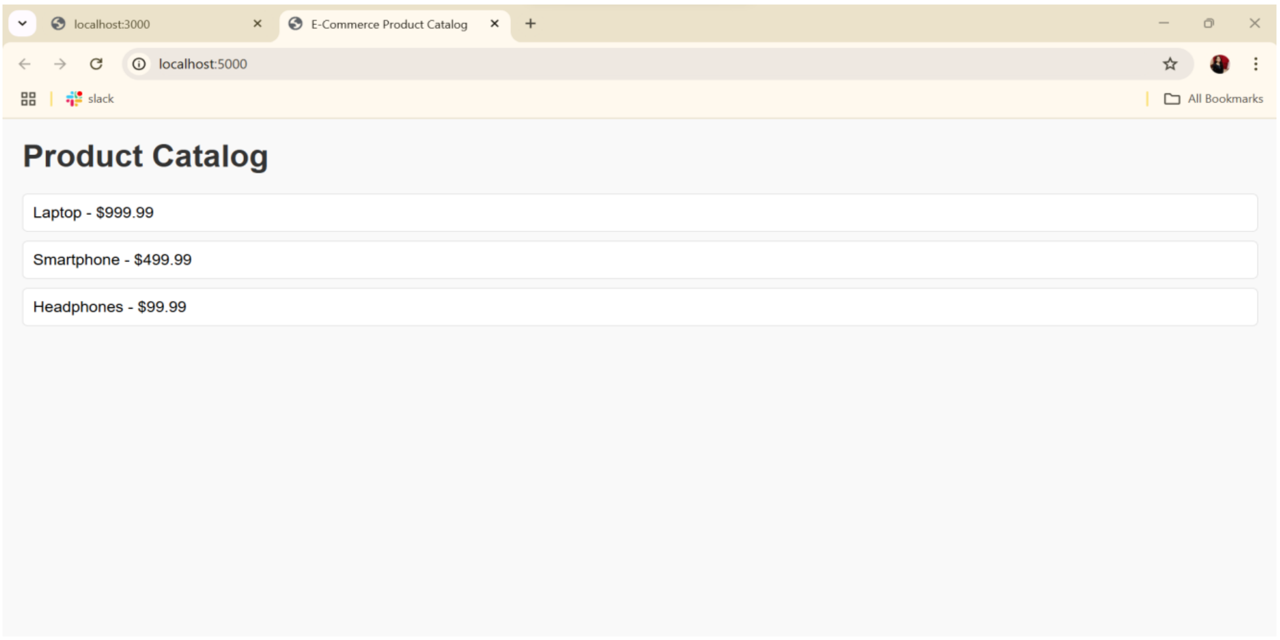
9.0 Running the Application:

9.1 Navigate to project directory in VS Code and build and start services:

- docker-compose build
- docker-compose up

9.2 Verify services:

- Product service: http://localhost:5000



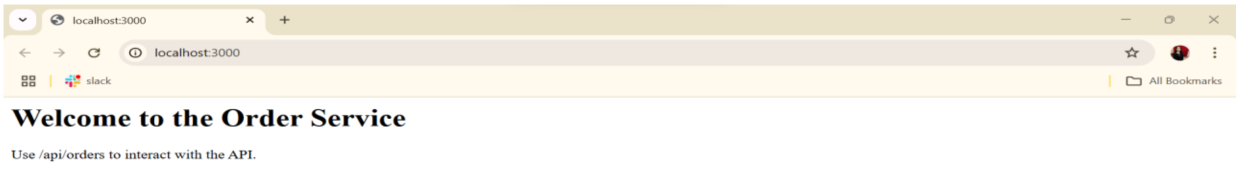
Product Catalog

Laptop - $999.99

Smartphone - $499.99

Headphones - $99.99

- Order service: http://localhost:3000

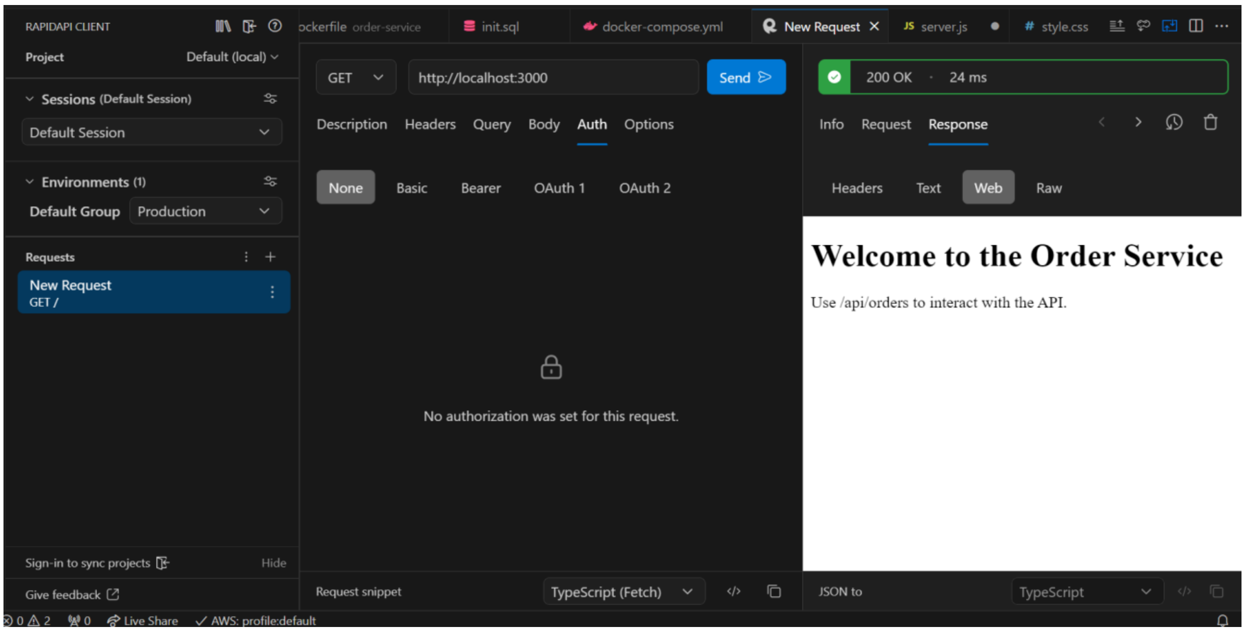


Welcome to the Order Service

Use /api/orders to interact with the API.

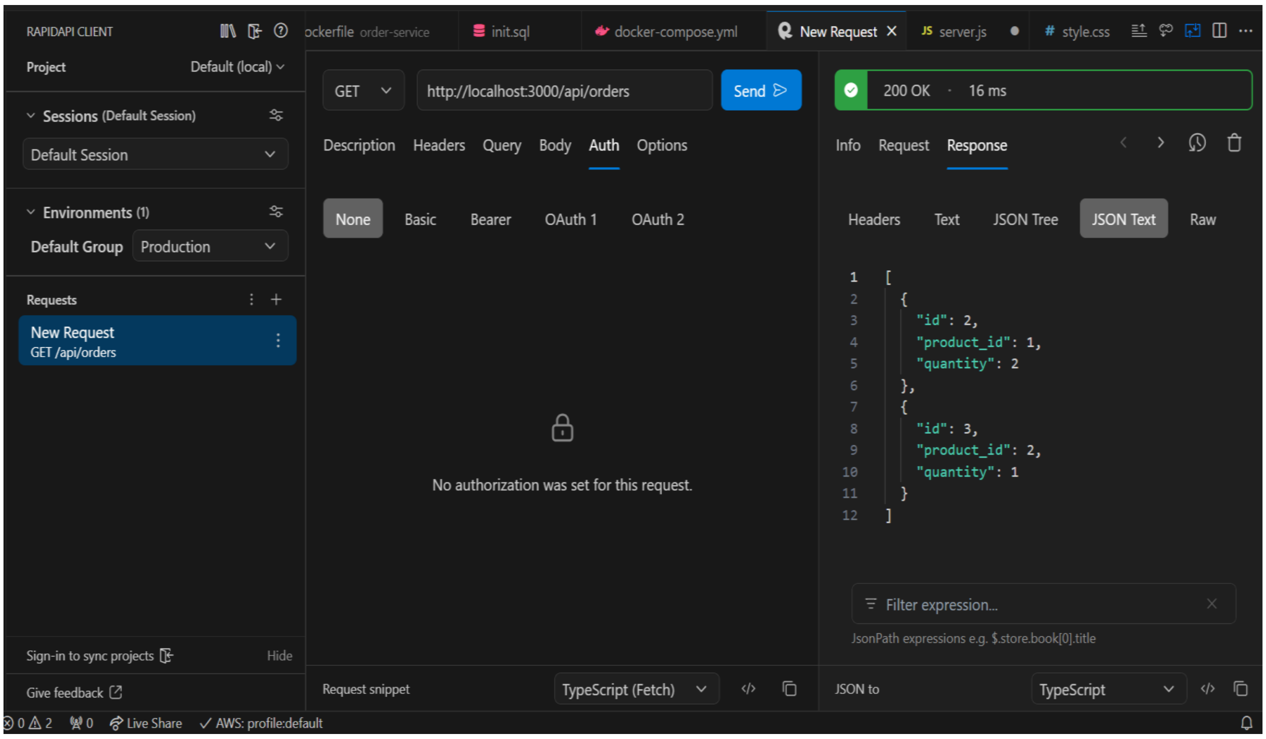
5.8 Testing:

Test the services using RapidAPI Client :



Welcome to the Order Service

Use /api/orders to interact with the API.

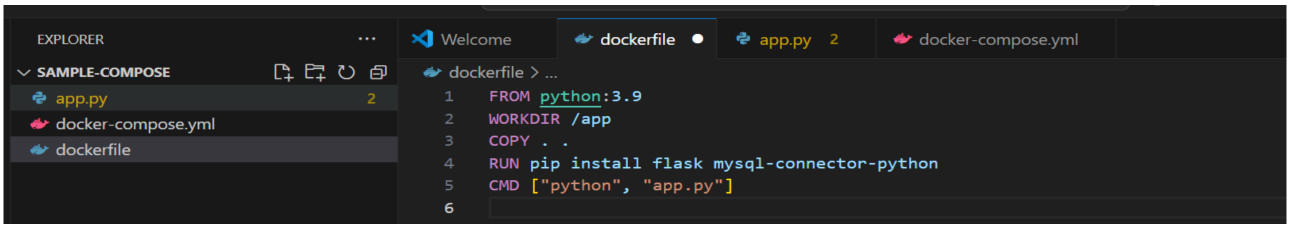


```
[
  {
    "id": 2,
    "product_id": 1,
    "quantity": 2
  },
  {
    "id": 3,
    "product_id": 2,
    "quantity": 1
  }
]
```

Setting Up Persistent Storage in Docker on Windows

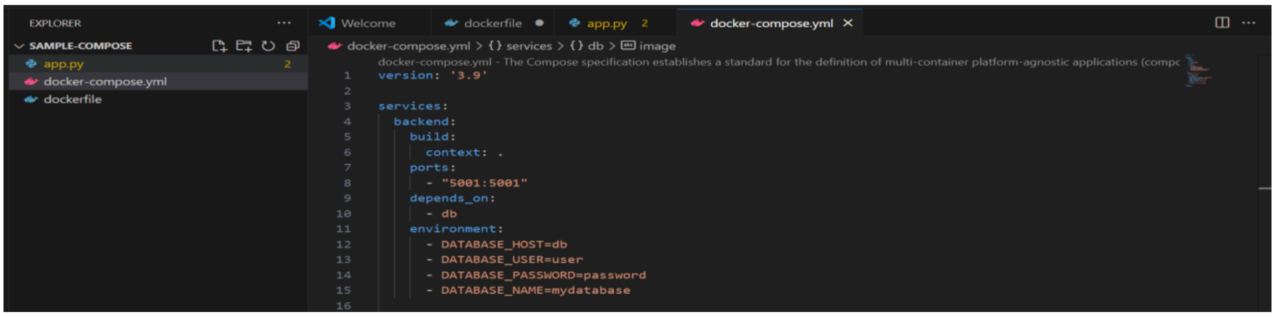
Sample-compose/dockerfile:

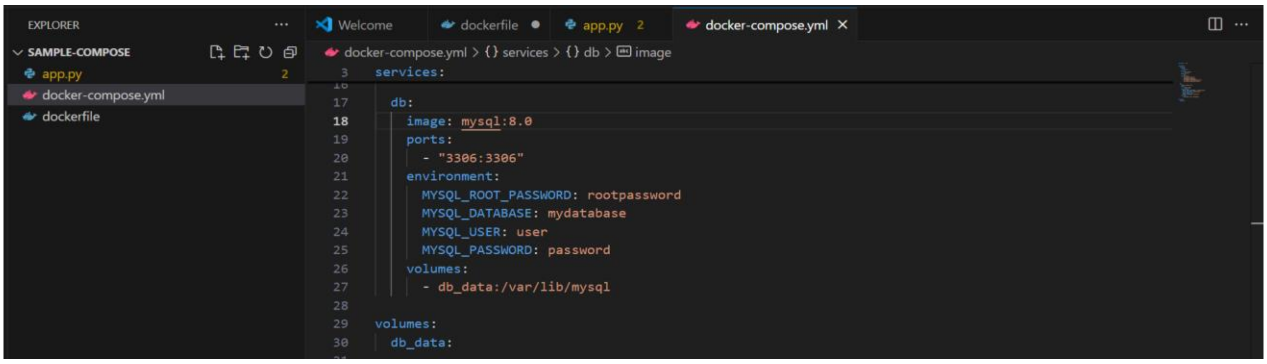
The Dockerfile defines the application image and its runtime environment


EXPLORER
SAMPLE-COMPOSE
app.py 2
docker-compose.yml
dockerfile

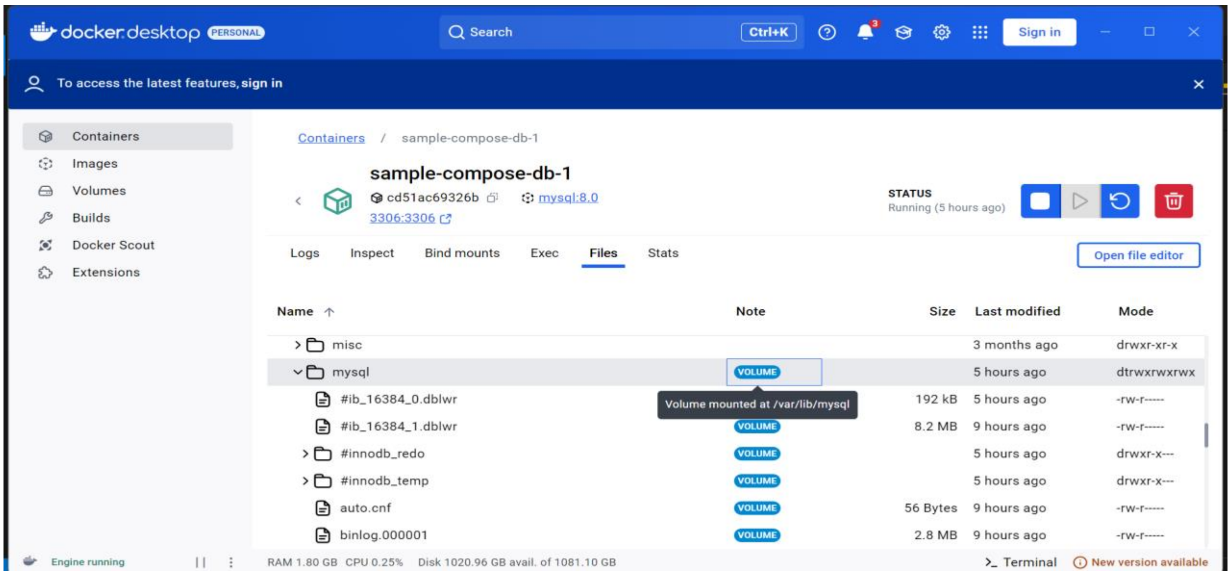
dockerfile > ...
```
FROM python:3.9
WORKDIR /app
COPY . .
RUN pip install flask mysql-connector-python
CMD ["python", "app.py"]
```

Data Persistence Strategy
docker-compose.yml
The docker-compose.yml file defines the service and volume mapping. Bind a Windows folder to the container for persistent storage. Database persistence is achieved through named volumes:


docker-compose.yml > {} services > {} db > image
docker-compose.yml - The Compose specification establishes a standard for the definition of multi-container platform-agnostic applications (compc
```
version: '3.9'

services:
  backend:
    build:
      context: .
    ports:
      - "5001:5001"
    depends_on:
      - db
    environment:
      - DATABASE_HOST=db
      - DATABASE_USER=user
      - DATABASE_PASSWORD=password
      - DATABASE_NAME=mydatabase
```


docker-compose.yml > {} services > {} db > image
```
services:
  db:
    image: mysql:8.0
    ports:
      - "3306:3306"
    environment:
      MYSQL_ROOT_PASSWORD: rootpassword
      MYSQL_DATABASE: mydatabase
      MYSQL_USER: user
      MYSQL_PASSWORD: password
    volumes:
      - db_data:/var/lib/mysql

volumes:
  db_data:
```

Verify Data Persistence:



Containers / sample-compose-db-1

sample-compose-db-1
cd51ac69326b mysql:8.0
3306:3306
STATUS
Running (5 hours ago)

Logs Inspect Bind mounts Exec Files Stats
Open file editor

Name	Note	Size	Last modified	Mode
misc			3 months ago	drwxr-xr-x
mysql	VOLUME		5 hours ago	dtrwxrwxrwx
#ib_16384_0.dblwr	Volume mounted at /var/lib/mysql	192 kB	5 hours ago	-rw-r-----
#ib_16384_1.dblwr	VOLUME	8.2 MB	9 hours ago	-rw-r-----
#innodb_redo	VOLUME		5 hours ago	drwxr-x---
#innodb_temp	VOLUME		5 hours ago	drwxr-x---
auto.cnf	VOLUME	56 Bytes	9 hours ago	-rw-r-----
binlog.000001	VOLUME	2.8 MB	9 hours ago	-rw-r-----

RAM 1.80 GB CPU 0.25% Disk 1020.96 GB avail. of 1081.10 GB
Terminal
New version available

Expert Cloud Consulting
Enhance Optimise & Scale