Work Integrated Learning Programmes

M.Tech Software Engineering

Scalable Services – SEZG583

Assignment 1

E-Commerce Microservices Application

Submitted By,

Pavithra.S

2022MT93172

2022mt93172@wilp.bits-pilani.ac.in

INDEX

Introduction

The e-commerce microservices application is developed using .NET 6 and comprises three distinct services –

- User Service,
- Product Service, and
- Purchase Service.

The implementation of microservices architecture is underscored by the individual deployment of each service within Docker containers, emphasizing a modular and scalable approach to application development.

In addition to the microservices, an API Gateway application has been integrated into the system using Ocelot. This gateway acts as a central entry point, facilitating communication between the distinct microservices. The inclusion of Ocelot streamlines the management of service interactions, enhances security, and provides a unified interface for external clients.

Furthermore, each microservice operates with its own dedicated database. This separation aligns with the microservices architectural pattern, enhancing data independence and promoting service autonomy. The specifics of this pattern fall into the category of "Database per Service," where each microservice has its own database schema tailored to its specific functionalities.

It is noteworthy that the development of this application was pursued individually, owing to challenges in group collaboration. Despite this circumstance, the ensuing application showcases a comprehensive understanding of microservices principles and their implementation.

Explanation Video

The explanation video for the application is available in the google drive link below,

Explanation Video

E-Commerce Application

The e-commerce application is designed to provide a seamless and efficient online shopping experience. The e-commerce application involves various independent services making it the most apt choice for building a microservices based application.

Let’s explore the services included in the e-commerce application and how they are integrated.

Prerequisites

Before starting the development of the e-commerce microservices application, it is required to set up the necessary tools and dependencies. The application was built on a Windows machine.

Below are the key tools and installations required for the development environment:

1. **Visual Studio Code**

   Visual Studio Code (VS Code) serves as the integrated development environment (IDE) for building and managing the application code.

   Downloaded and installed VS Code from https://code.visualstudio.com/.

2. **PostgreSQL**

   PostgreSQL is the relational database management system (RDBMS) chosen for the microservices.

   Downloaded and installed PostgreSQL from https://www.postgresql.org/download/windows/.

   Ensured that the PostgreSQL service is running after installation.

3. **Docker Desktop**

   Docker desktop provides containerization for the microservices, helping in deployment and management.

   Installed Docker Desktop from https://www.docker.com/products/docker-desktop.

   Docker will enable the encapsulation of each microservice in a container for consistent and reproducible deployment.

4. **.NET 6 SDK**

   The .NET 6 Software Development Kit (SDK) is crucial for building and running our .NET-based microservices.

   Downloaded and installed .NET 6 SDK from https://dotnet.microsoft.com/download/dotnet/6.0.

5. **Postman Desktop**

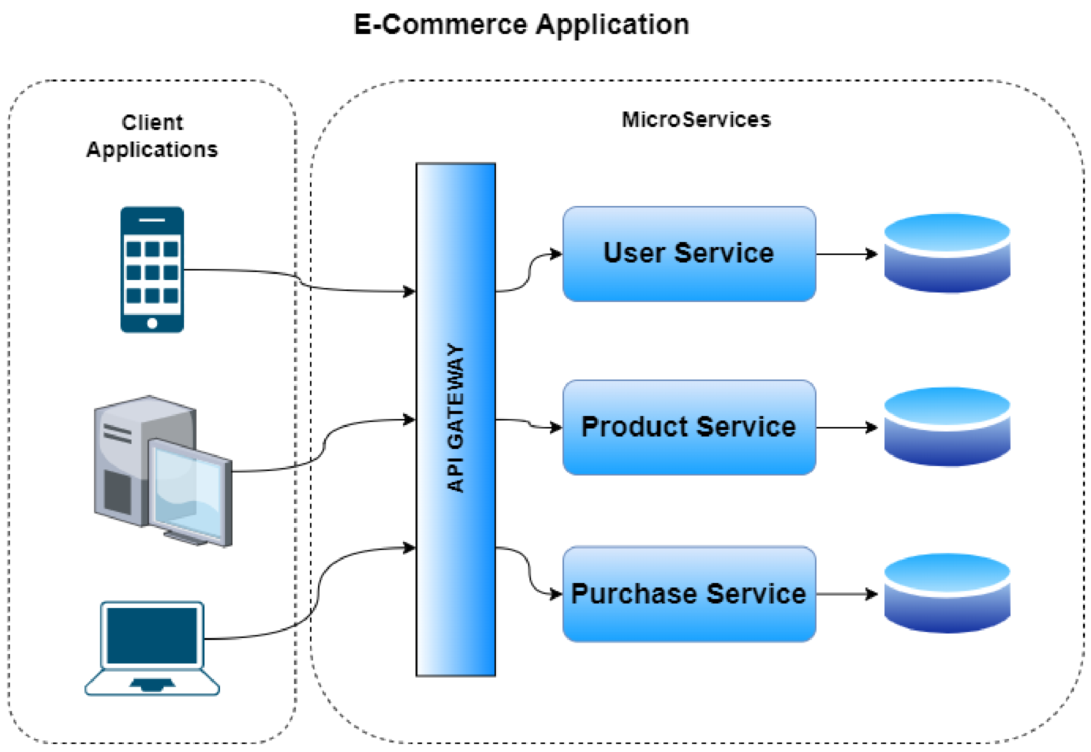
   Postman is a powerful tool for testing APIs and streamlining the development workflow.

   Downloaded and installed Postman Desktop from https://www.postman.com/downloads/.

Postman is very helpful for testing the functionalities of our microservices and ensuring smooth API interactions.

Once these tools are successfully installed, all the setup required for the development of the e-commerce microservices application is done.

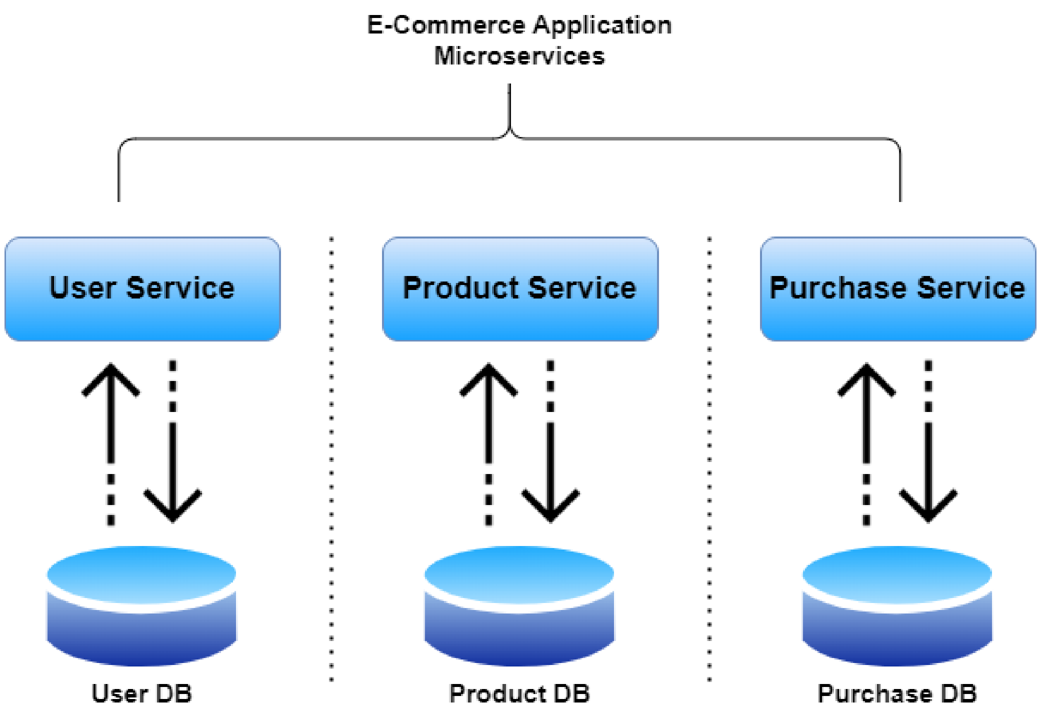
System Architecture



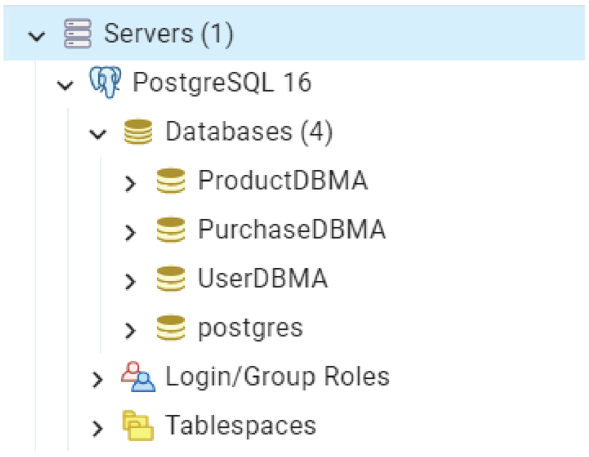
Database Scheme

The database schema for our e-commerce microservices application is designed to align with the microservices architecture, specifically following the **"Database per Service"** pattern.

The **"Database per Service"** model is an approach in microservices architecture where each microservice has its own dedicated database. Instead of sharing a common database across multiple services, each microservice manages its own data storage independently.



The PostgreSQL database is used for the application with three separate databases for each of the service.



The advantages for using this pattern are,

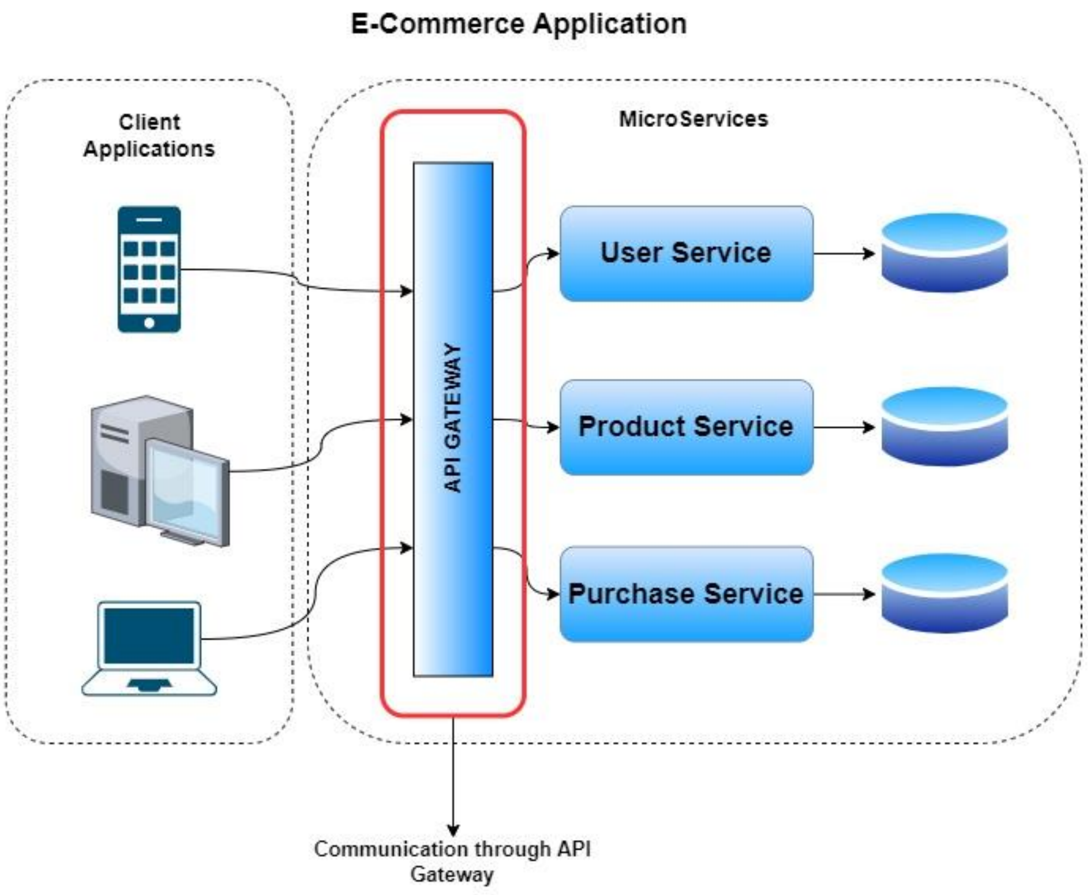
1. **Autonomy:** Each microservice operates independently with its own database, allowing for autonomous development, deployment, and scaling.
2. **Decoupling:** Reduces dependencies between microservices, enhancing agility and making it easier to modify or replace a microservice without affecting others.
3. **Data Consistency:** Transactions within a microservice are contained within its database, ensuring data consistency and integrity within that service.
4. **Granular Scalability:** Enables granular scalability, as each microservice can be scaled independently based on its specific resource requirements.

5. **Technology Flexibility:** Allows for the use of different database technologies that best suit the specific needs of each microservice.
6. **Isolation of Failure:** Failures or issues in one microservice or its database do not impact other microservices, improving overall system resilience.
7. **Security and Compliance:** Enables tailored security measures at the database level for microservices dealing with sensitive data or specific compliance requirements.
8. **Maintenance and Evolvability:** Facilitates independent maintenance tasks, such as schema changes or updates, for each microservice's database, making it easier to evolve and update individual services.

Micrservices Communication

In our e-commerce microservices architecture, the communication between the individual services is done through APIs, adhering to the **API Gateway Pattern**.

This pattern includes a centralized API Gateway application to manage and streamline communication between client applications and the diverse microservices within the system.



The API Gateway application acts as the main entry point for external clients, providing a unified interface to interact with the microservices ecosystem. It routes incoming requests to the appropriate microservice based on the requested functionality.

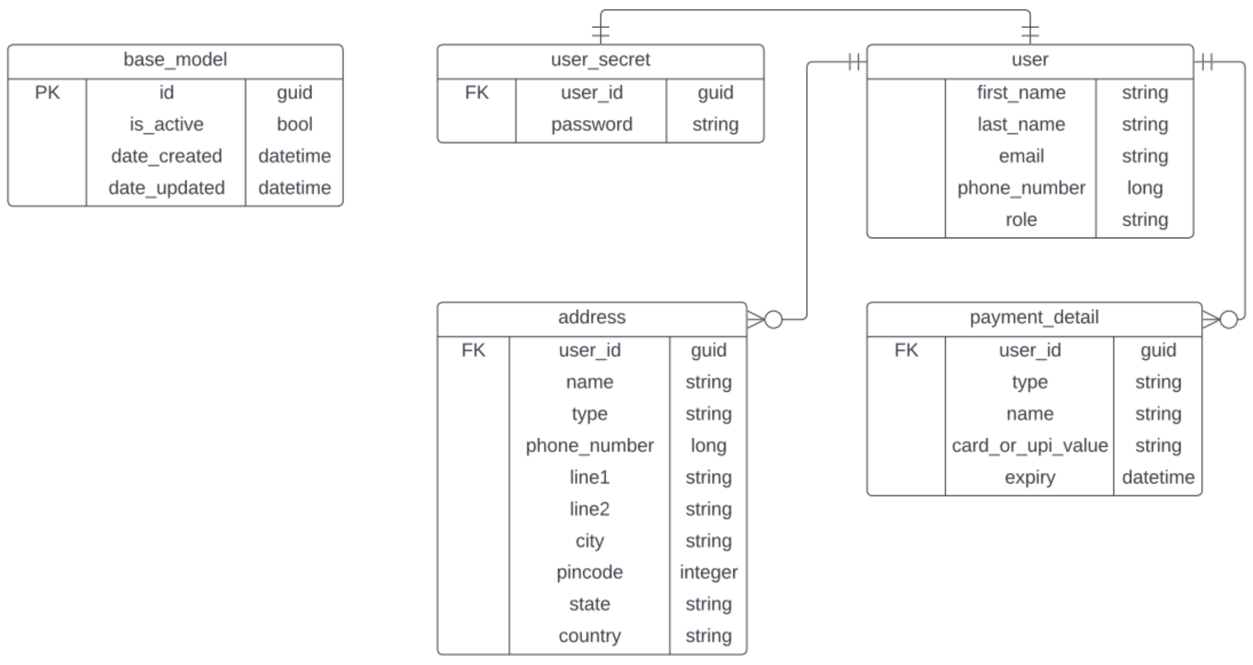
User Service

Overview

The User Microservice offers a range of endpoints to facilitate user account management, including user creation, authentication, and the management of user addresses and payments.

Database Design

The user database includes the below tables and relationships.



API Endpoints

User Account Endpoints

This group of endpoints provides functionalities to manage user accounts, including creation, authentication, retrieval, updating, and deletion. The "Verify User" endpoint allows the verification of a user's account.

#	Endpoint	Method	Description
1	**/api/user**	POST	Creates a new user account.
2	**/api/user**	GET	Retrieves a list of all user accounts.
3	**/api/user/login**	POST	Authenticates a user based on provided credentials.
4	**/api/user/{id}**	GET	Retrieves detailed information about a user by ID.
5	**/api/user/{id}**	PUT	Updates user details by ID.

6	**/api/user/{id}**	DELETE	Deletes a user account by ID.
7	**/api/user/{userId}/verify**	GET	Verifies the user's account.

User Address Endpoints

This group of endpoints manages user addresses, allowing the addition, retrieval, updating, and deletion of addresses. The "Verify User Address" endpoint verifies a specific user address.

#	Endpoint	Method	Description
1	**/api/user/{userId}/address**	POST	Adds a new address for the specified user.
2	**/api/user/{userId}/address**	GET	Retrieves all addresses associated with a user.
3	**/api/user/{userId}/address/{addressId}**	GET	Retrieves detailed information about a specific user address by ID.
4	**/api/user/{userId}/address/{addressId}**	PUT	Updates a user address by ID.
5	**/api/user/{userId}/address/{addressId}**	DELETE	Deletes a user address by ID.
6	**/api/user/{userId}/address/{addressId}/verify**	GET	Verifies the specified user address.

User Payment Endpoints

This group of endpoints manages user payments, allowing the addition, retrieval, updating, and deletion of payment methods. The "Verify User Payment" endpoint verifies a specific user payment method.

#	Endpoint	Method	Description
1	**/api/user/{userId}/payment**	POST	Adds a new payment method for the specified user.
2	**/api/user/{userId}/payment**	GET	Retrieves all payment methods associated with a user.
3	**/api/user/{userId}/payment/{paymentId}**	GET	Retrieves detailed information about a specific user payment method by ID.
4	**/api/user/{userId}/payment/{paymentId}**	PUT	Updates a user payment method by ID.

5	**/api/user/{userId}/payment/{paymentId}**	DELETE	Deletes a user payment method by ID.
6	**/api/user/{userId}/payment/{paymentId}/verify**	GET	Verifies the specified user payment method.

Code Repository

The complete source code for the User Microservice is available on GitHub. You can access and explore the code repository at the following link:

[GitHub Repository: User Microservice]

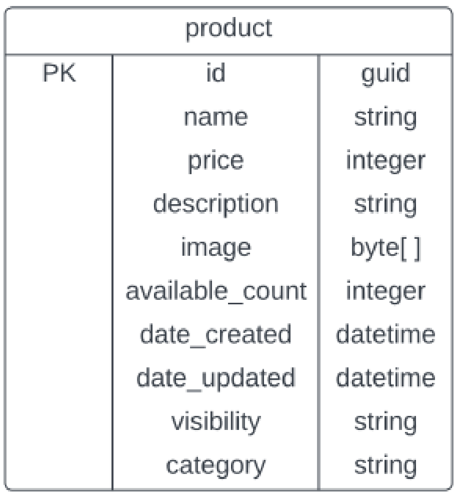
Product Service

Overview

The Product Microservice API offers a set of endpoints for managing product-related functionalities, including the creation, retrieval, updating, and deletion of products. Additionally, it provides an endpoint for verifying product information.

Database Design

The product database includes the below product table.



product		
PK	id	guid
	name	string
	price	integer
	description	string
	image	byte[ ]
	available_count	integer
	date_created	datetime
	date_updated	datetime
	visibility	string
	category	string

API Endpoints

Product Endpoints

This group of endpoints provides functionalities to manage products, including creation, retrieval, updating, and deletion. The "Verify Product" endpoint allows the verification of product information.

#	Endpoint	Method	Description
1	**/api/product**	POST	Creates a new product.
2	**/api/product**	GET	Retrieves a list of all products.
3	**/api/product/{productId}**	GET	Retrieves detailed information about a product by ID.
4	**/api/product/{productId}**	PUT	Updates product details by ID.
5	**/api/product/{productId}**	DELETE	Deletes a product by ID.
6	**/api/product/verify**	POST	Verifies product information.

Code Repository

The complete source code for the Product Microservice is available on GitHub. You can access and explore the code repository at the following link:

GitHub Repository: Product Microservice

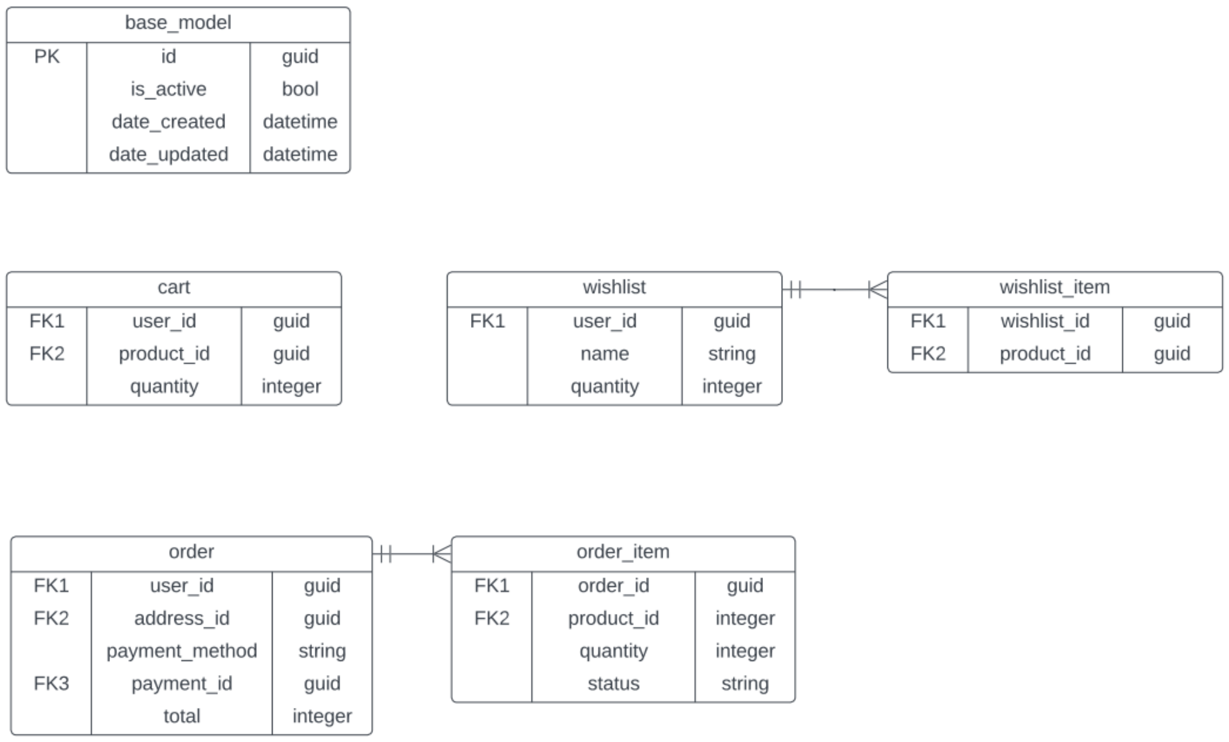
Purchase Service

Overview

The Purchase Microservice API offers a set of endpoints for managing purchase-related functionalities, including cart management, order processing, and wish list management.

Database Design

The purchase database includes the below tables and relationships.



API Endpoints

Cart Endpoints

This group of endpoints provides functionalities to manage the shopping cart, including creation, retrieval, and updating.

#	Endpoint	Method	Description
1	**/api/cart**	POST	Creates a new shopping cart.
2	**/api/cart**	GET	Retrieves a list of all shopping carts.
3	**/api/cart**	PUT	Updates the contents of the shopping cart.

Order Endpoints

This group of endpoints manages order processing, including order creation and retrieval.

#	Endpoint	Method	Description
1	**/api/order**	POST	Creates a new order.
2	**/api/order**	GET	Retrieves a list of all orders.
3	**/api/order/{orderId}**	GET	Retrieves detailed information about an order by ID.

Wish List Endpoints

This group of endpoints manages wish lists, allowing the creation, retrieval, updating, and deletion of wish list items.

#	Endpoint	Method	Description
1	**/api/wish-list**	POST	Creates a new wish list.
2	**/api/wish-list**	GET	Retrieves a list of all wish lists.
3	**/api/wish-list/{wishListId}**	GET	Retrieves detailed information about a wish list by ID.
4	**/api/wish-list/{wishListId}**	PUT	Updates a wish list by ID.
5	**/api/wish-list/{wishListId}**	DELETE	Deletes a wish list by ID.

Code Repository

The complete source code for the Purchase Microservice is available on GitHub. You can access and explore the code repository at the following link:

GitHub Repository: Purchase Microservice

API Gateway

Overview

The API Gateway built with Ocelot serves as a centralized entry point for accessing and managing the User, Product, and Purchase microservices. It handles routing and aggregation of microservices' functionalities.

Ocelot

Ocelot is an open-source API Gateway for .NET. It is designed to work with the .NET Core platform and provides a simple way to manage, aggregate, and secure APIs. Ocelot acts as a central entry point for incoming HTTP requests and facilitates communication between microservices in a distributed architecture.

Some key advantages of using Ocelot are,

1. **Simple Gateway:** Ocelot provides a lightweight and straightforward API Gateway solution.
2. **Flexible Routing:** Easily configure rules for directing requests to different microservices.

3. **Centralized Security:** Manage authentication and authorization centrally for consistent security policies.
4. **Load Balancing:** Support for load balancing enhances system reliability and resource utilization.
5. **Response Aggregation:** Aggregate multiple microservice responses into a single response.
6. **Service Discovery:** Seamlessly integrate with service discovery mechanisms for dynamic routing.
7. **Middleware Integration:** Integrates smoothly into the ASP.NET Core pipeline as middleware.
8. **Dynamic Reconfiguration:** Modify configurations dynamically without restarting the service.
9. **Logging and Monitoring:** Built-in features for tracing and debugging requests.
10. **Transformation Support:** Easily transform both incoming requests and outgoing responses.
11. **Extensibility:** Add custom middleware to extend functionality as needed.
12. **Rate Limiting:** Built-in support for controlling the rate of requests to microservices.
13. **Open-Source Community:** Ocelot benefits from an active open-source community for ongoing support and development.

Code Repository

The complete source code for the Product Microservice is available on GitHub. You can access and explore the code repository at the following link:

GitHub Repository: API Gateway

Docker Deployment

Docker provides a standardized and lightweight approach to packaging applications and their dependencies into containers. These containers encapsulate the runtime environment, ensuring consistency from development through testing to production.

In our e-commerce microservices application built using .NET 6, Docker plays a pivotal role in facilitating a smooth and efficient deployment workflow.

Some key advantages of using Docker are,

1. **Consistency:** Ensure consistent environments across development, testing, and production.
2. **Isolation:** Run microservices in isolated containers, preventing dependency conflicts.
3. **Efficient Scaling**: Seamlessly scale individual services based on demand.
4. **Streamlined Integration:** Simplify the integration of disparate microservices using Docker Compose or Kubernetes.
5. **Portability:** Containers are portable, enabling smooth transitions between environments.

Step-by-Step Deployment to Docker

The below steps were followed to perform the docker deployment.

1. Create Dockerfiles for Microservices

For each microservice, User, Product, and Purchase, create a Dockerfile specifying the necessary dependencies and configurations.

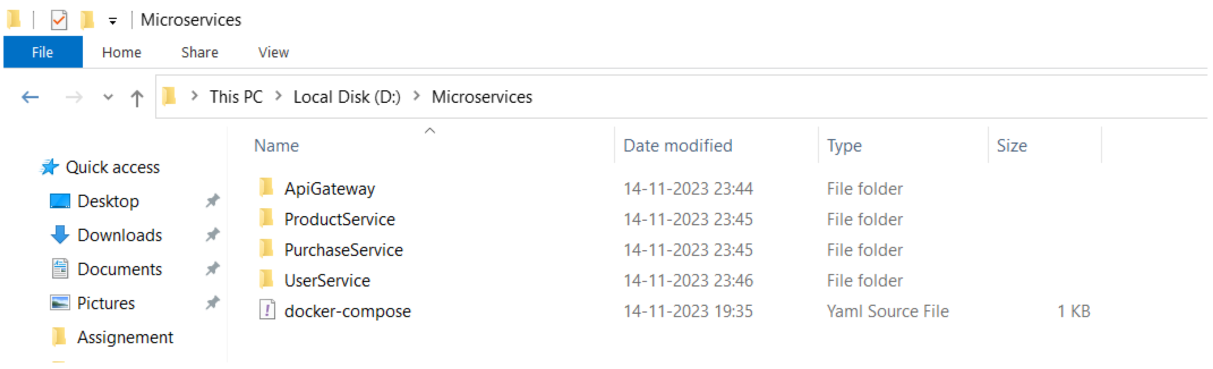
Example Dockerfile (User Microservice):

```
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /app

EXPOSE 443
EXPOSE 80

# copy everything into the docker directory and restore it in docker directory
COPY . .
RUN dotnet restore

# Build the application
RUN dotnet publish -c Release -o out

# Build runtime final image
FROM mcr.microsoft.com/dotnet/aspnet:6.0
WORKDIR /app
COPY --from=build /app/out .

ENTRYPOINT ["dotnet", "UserService.dll"]
```

The same structure is followed for the product and purchase services too.

2. Create Docker Compose File

Create a **docker-compose.yml** file outside all the microservice folder, to define and configure each microservice.



Name	Date modified	Type	Size
ApiGateway	14-11-2023 23:44	File folder	
ProductService	14-11-2023 23:45	File folder	
PurchaseService	14-11-2023 23:45	File folder	
UserService	14-11-2023 23:46	File folder	
docker-compose	14-11-2023 19:35	Yaml Source File	1 KB

The docker-compose.yml file used for the microservices are,

```
version: '3.5'
services:
  UserService:
    image: ${DOCKER_REGISTRY-}user-microservice:v1
    build:
     context: ./UserService
     dockerfile: Dockerfile
    environment:
     -
CONNECTIONSTRINGS__DEFAULTCONNECTION=Server=host.docker.internal;Port=5432;Dat
abase=UserDBMA;Username=postgres;Password=pavithra
    ports:
     - "5004:80"
  ProductService:
    image: ${DOCKER_REGISTRY-}product-microservice:v1
    build:
     context: ./ProductService
     dockerfile: Dockerfile
    environment:
     -
CONNECTIONSTRINGS__DEFAULTCONNECTION=Server=host.docker.internal;Port=5432;Dat
abase=ProductDBMA;Username=postgres;Password=pavithra
    ports:
     - "5054:80"
  PurchaseService:
    image: ${DOCKER_REGISTRY-}purchase-microservice:v1
    build:
     context: ./PurchaseService
     dockerfile: Dockerfile
    environment:
     -
CONNECTIONSTRINGS__DEFAULTCONNECTION=Server=host.docker.internal;Port=5432;Dat
abase=PurchaseDBMA;Username=postgres;Password=pavithra
    ports:
     - "5113:80"
```

UserService

- **Image:** Docker image named **user-microservice:v1**.
- **Build Context:** The build context is set to the **./UserService** directory.
- **Dockerfile:** The Dockerfile named **Dockerfile** is used for building the image.
- **Environment:**
    - **CONNECTIONSTRINGS__DEFAULTCONNECTION**: Specifies the PostgreSQL connection string for the UserDBMA database.
- **Ports:** Maps port 80 from the container to port 5004 on the host machine.

ProductService

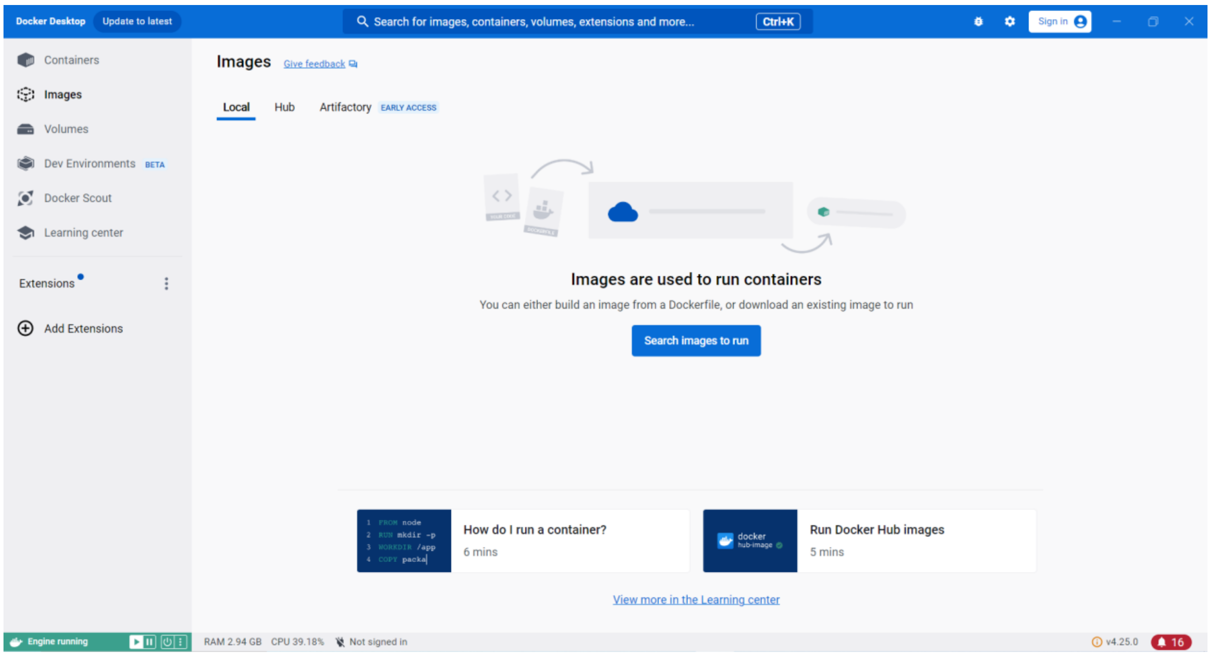
- **Image:** Docker image named **product-microservice:v1**.
- **Build Context:** The build context is set to the **./ProductService** directory.
- **Dockerfile:** The Dockerfile named **Dockerfile** is used for building the image.
- **Environment:**
    - **CONNECTIONSTRINGS__DEFAULTCONNECTION**: Specifies the PostgreSQL connection string for the ProductDBMA database.
- **Ports:** Maps port 80 from the container to port 5054 on the host machine.

PurchaseService

- **Image:** Docker image named **purchase-microservice:v1**.
- **Build Context:** The build context is set to the **./PurchaseService** directory.
- **Dockerfile:** The Dockerfile named **Dockerfile** is used for building the image.
- **Environment:**
    - **CONNECTIONSTRINGS__DEFAULTCONNECTION**: Specifies the PostgreSQL connection string for the PurchaseDBMA database.
- **Ports:** Maps port 80 from the container to port 5113 on the host machine.

3. Build Docker Images

Make sure to keep the docker desktop application running before executing this command.



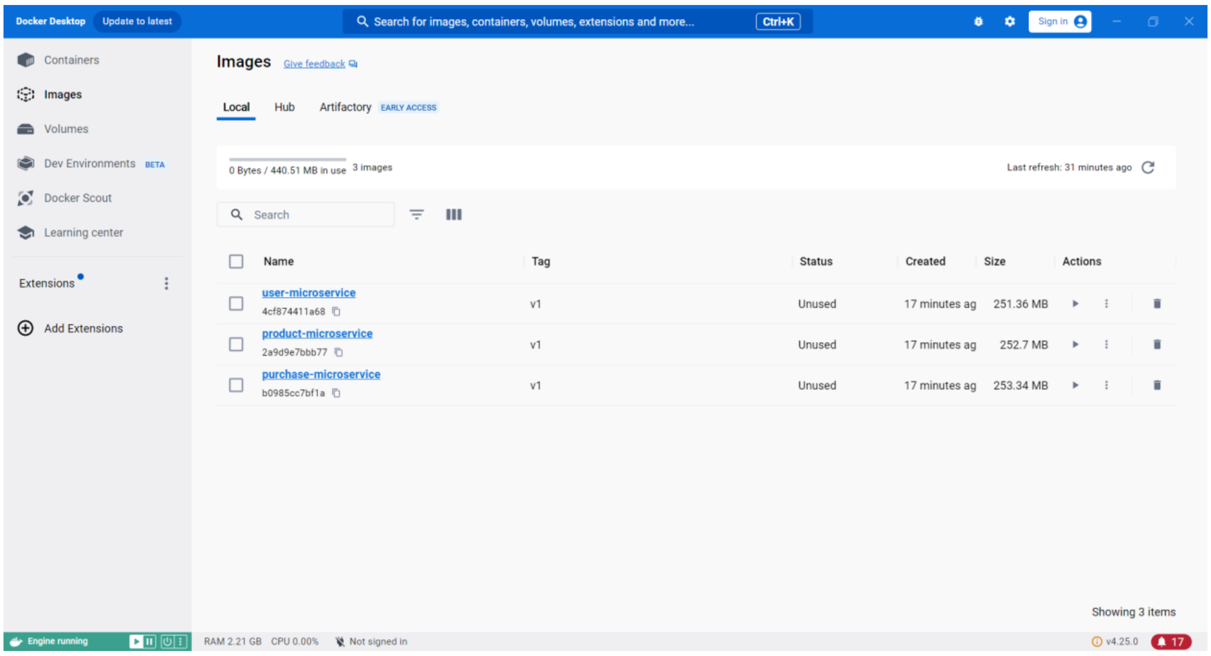
Images are used to run containers

You can either build an image from a Dockerfile, or download an existing image to run

Use the below command to build the docker images for the microservices.

```
docker-compose build
```

Once the execution is completed, the docker images will be created.



0 Bytes / 440.51 MB in use 3 images

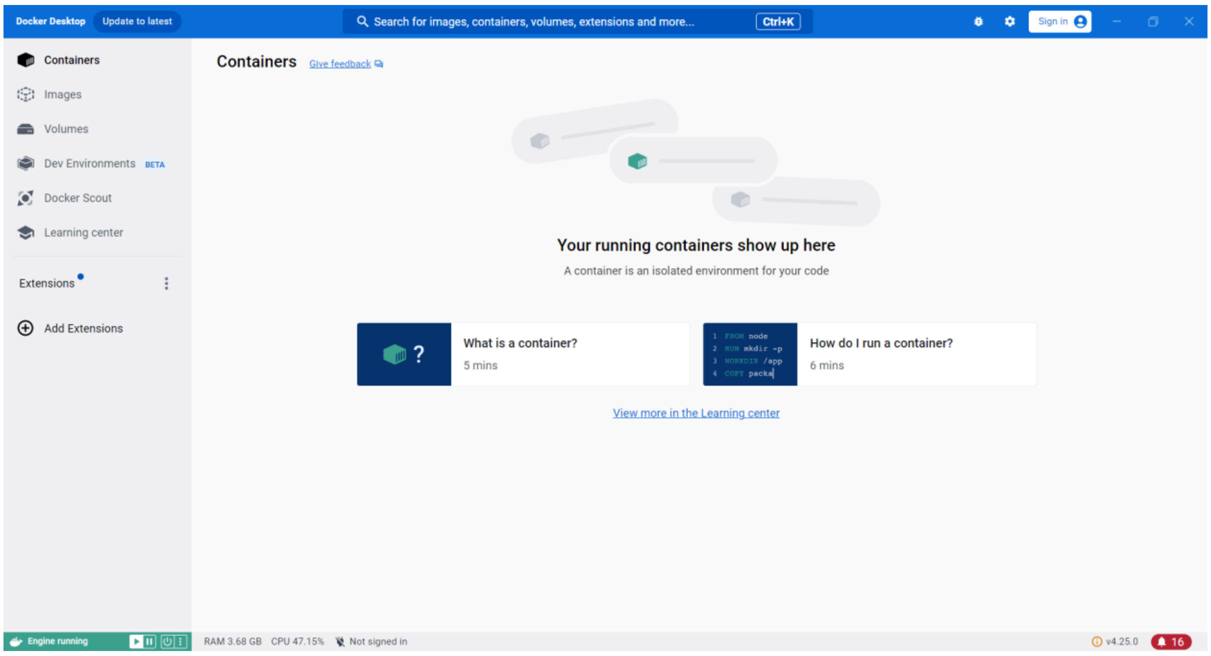
Last refresh: 31 minutes ago

Name	Tag	Status	Created	Size	Actions
user-microservice 4cf874411a68	v1	Unused	17 minutes ag	251.36 MB	
product-microservice 2a9d9e7bbb77	v1	Unused	17 minutes ag	252.7 MB	
purchase-microservice b0985cc7bf1a	v1	Unused	17 minutes ag	253.34 MB	

Showing 3 items

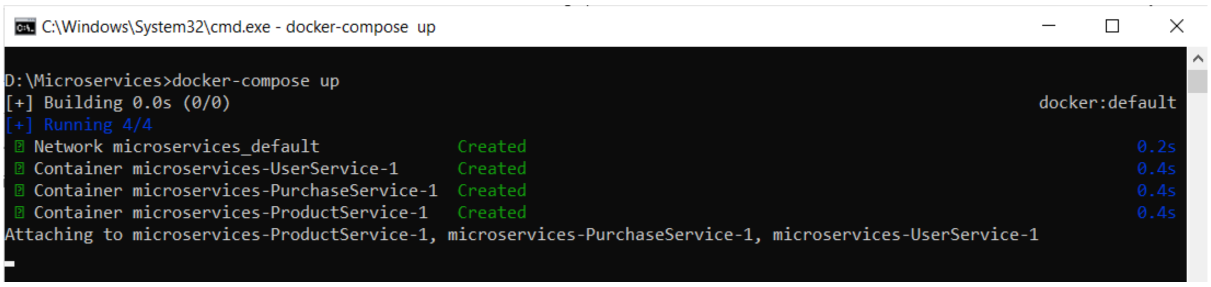
4. Run Docker Compose

The containers can be viewed under the container tab inside docker desktop.


Docker Desktop
Update to latest
Search for images, containers, volumes, extensions and more...
Ctrl+K
Sign in
Containers
Images
Volumes
Dev Environments BETA
Docker Scout
Learning center
Extensions
Add Extensions
Containers
Give feedback
Your running containers show up here
A container is an isolated environment for your code
What is a container?
5 mins
How do I run a container?
6 mins
View more in the Learning center
Engine running
RAM 3.68 GB CPU 47.15% Not signed in
v4.25.0
16

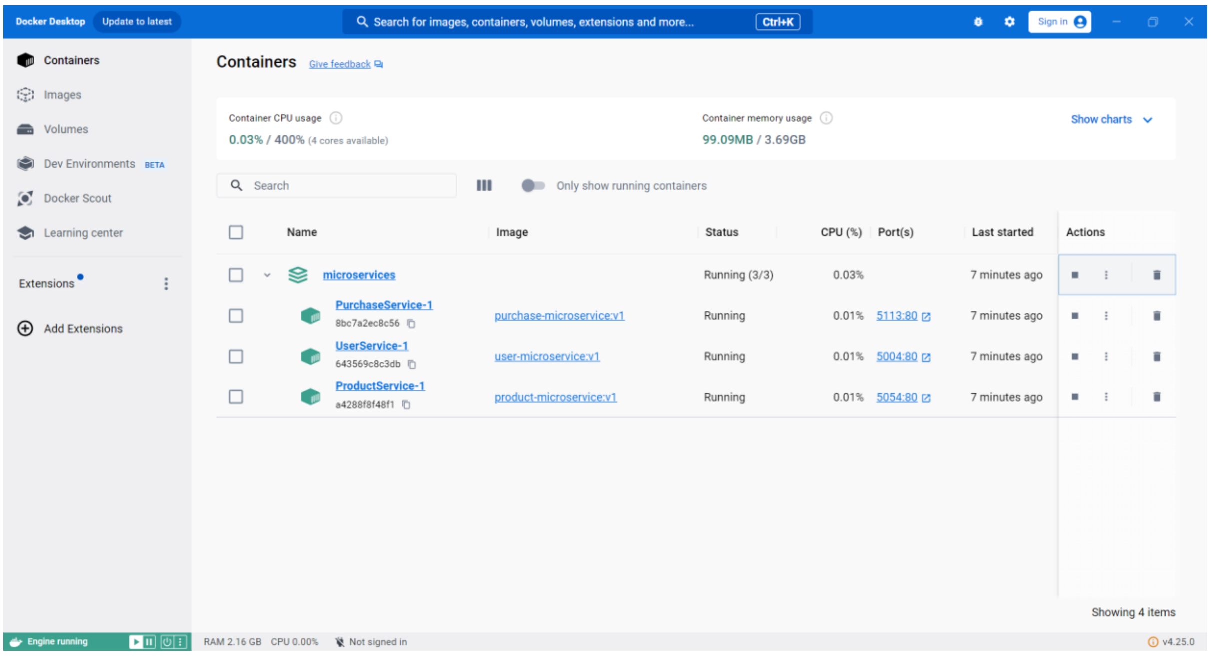
Use the "**docker-compose up**" command to start the services defined in the Docker Compose file.

```
docker-compose up
```



```
C:\Windows\System32\cmd.exe - docker-compose up

D:\Microservices>docker-compose up
[+] Building 0.0s (0/0)                                                          docker:default
[+] Running 4/4
 ✔ Network microservices_default            Created                                      0.2s
 ✔ Container microservices-UserService-1     Created                                      0.4s
 ✔ Container microservices-PurchaseService-1  Created                                    0.4s
 ✔ Container microservices-ProductService-1   Created                                    0.4s
Attaching to microservices-ProductService-1, microservices-PurchaseService-1, microservices-UserService-1
```

Once the execution is completed, we can see the docker containers running in the docker desktop application.



Containers

Container CPU usage 0.03% / 400% (4 cores available)

Container memory usage 99.09MB / 3.69GB

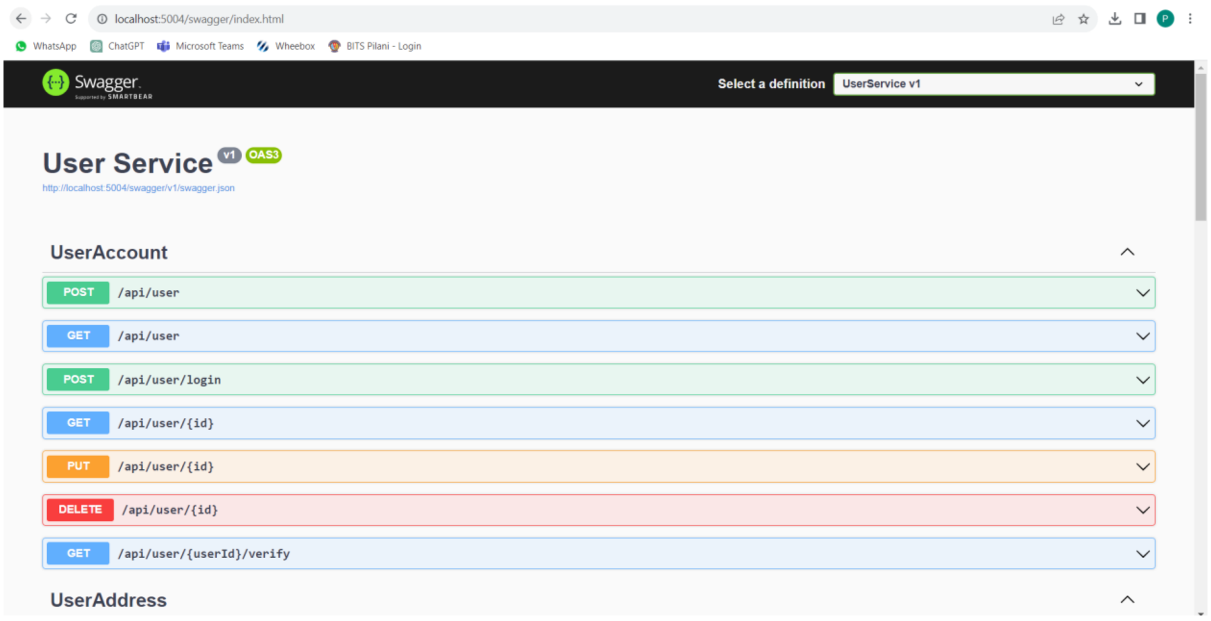
Name	Image	Status	CPU (%)	Port(s)	Last started
microservices		Running (3/3)	0.03%		7 minutes ago
PurchaseService-1 8bc7a2ec8c56	purchase-microservice:v1	Running	0.01%	5113:80	7 minutes ago
UserService-1 643569c8c3db	user-microservice:v1	Running	0.01%	5004:80	7 minutes ago
ProductService-1 a4288f8f48f1	product-microservice:v1	Running	0.01%	5054:80	7 minutes ago

Showing 4 items

5. Access Microservices

Access the microservices using the specified ports.
For example:

- User Service: http://localhost:5004
- Product Service: http://localhost:5054
- Purchase Service: http://localhost:5113



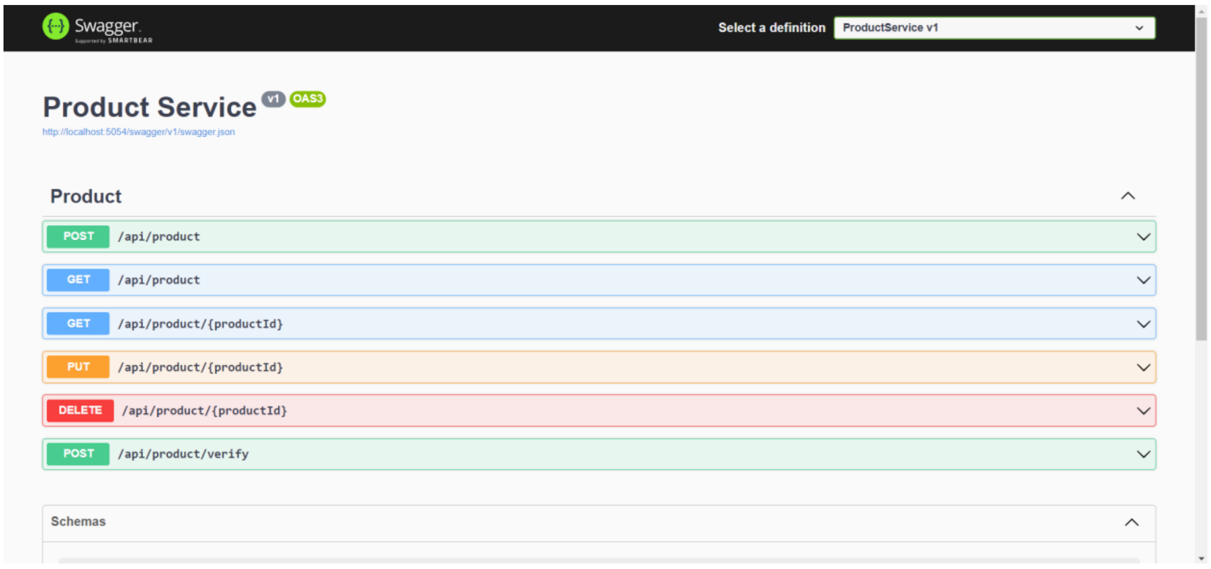
User Service v1 OAS3

http://localhost:5004/swagger/v1/swagger.json

UserAccount

- POST /api/user
- GET /api/user
- POST /api/user/login
- GET /api/user/{id}
- PUT /api/user/{id}
- DELETE /api/user/{id}
- GET /api/user/{userId}/verify

UserAddress


Swagger

Select a definition ProductService v1

Product Service v1 OAS3

http://localhost:5054/swagger/v1/swagger.json

Product

POST /api/product

GET /api/product

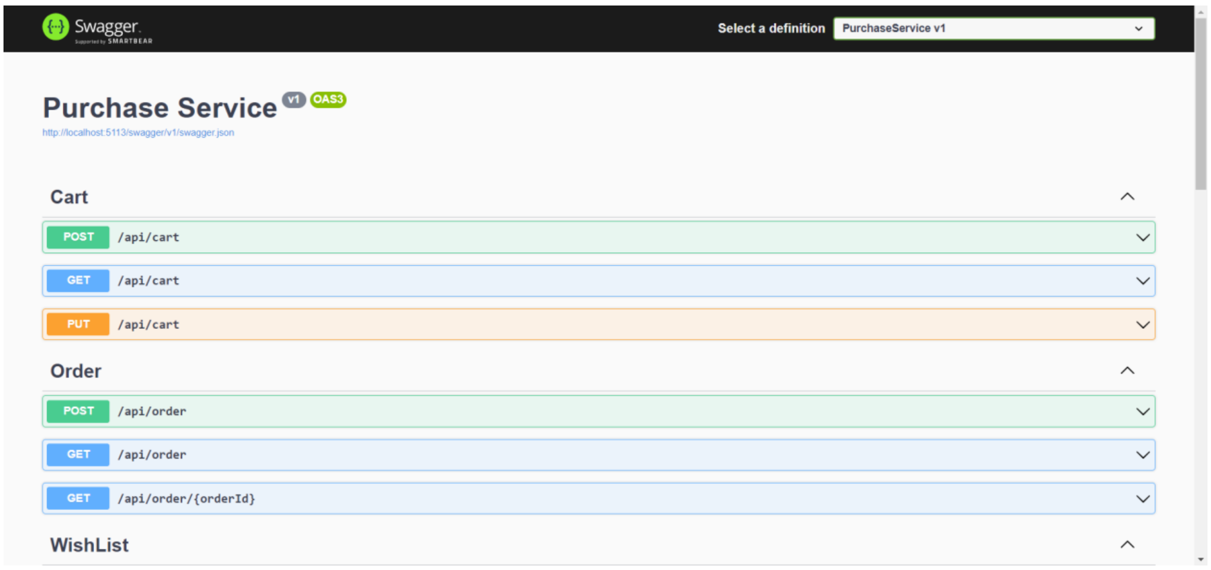
GET /api/product/{productId}

PUT /api/product/{productId}

DELETE /api/product/{productId}

POST /api/product/verify

Schemas


Swagger

Select a definition PurchaseService v1

Purchase Service v1 OAS3

http://localhost:5113/swagger/v1/swagger.json

Cart

POST /api/cart

GET /api/cart

PUT /api/cart

Order

POST /api/order

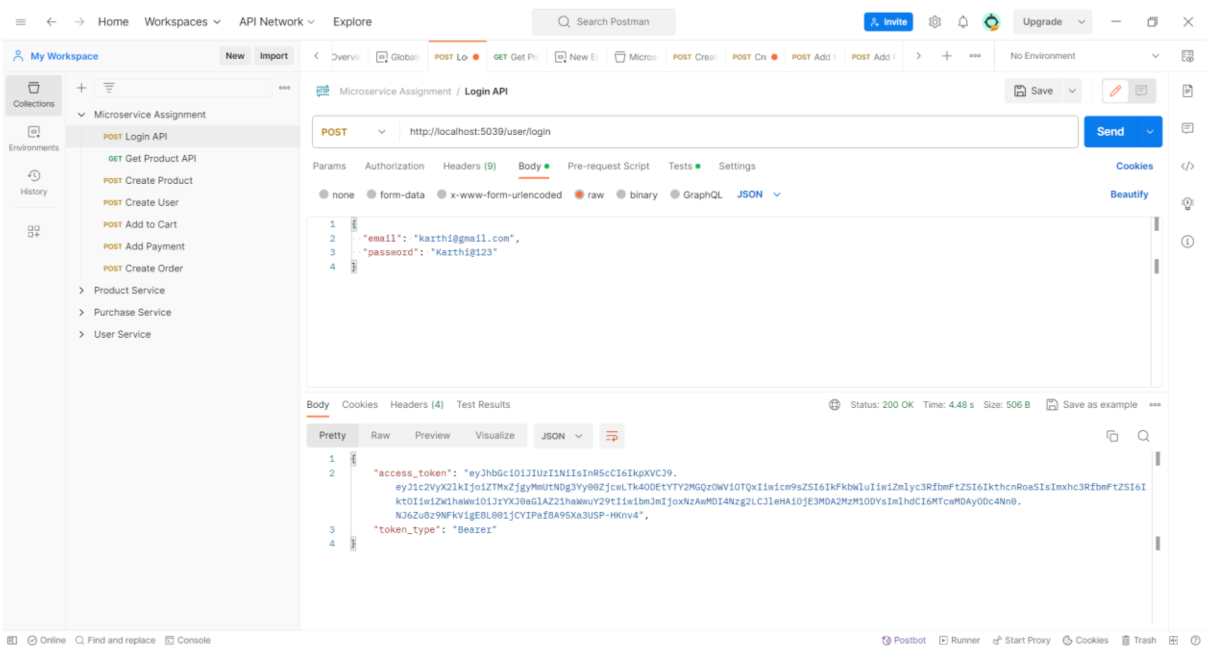
GET /api/order

GET /api/order/{orderId}

WishList

Testing using Postman

Postman provides a powerful and user-friendly platform for testing microservices. It helps in identifying potential issues early in the development lifecycle.



POST http://localhost:5039/user/login

```
{
  "email": "karthi@gmail.com",
  "password": "Karthi@123"
}
```

Status: 200 OK   Time: 4.48 s   Size: 506 B

```
{
    "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJ1c2VyX2lkIjoiZTMxZjgyMmUtNDg3Yy00ZjcwLTk4ODEtYTY2MGQzOWViOTQxIiwicm9sZSI6IkFkbWluIiwiZmlyc3RfbmFtZSI6IkthcnRoaSIsImxhc3RfbmFtZSI6I
        ktOIiwiZW1haWwiOiJrYXJ0aGlAZ21haWwuY29tIiwibmJmIjoxNzAwMDI4Nzg2LCJleHAiOjE3MDA2MzM1ODYsImlhdCI6MTcwMDAyODc4Nn0.
        NJ6Zu8z9NFkVigE8L001jCYIPaf8A95Xa3USP-HKnv4",
    "token_type": "Bearer"
}
```

References

https://www.c-sharpcorner.com/article/implementation-and-containerization-of-microservices-using-net-core-6-and-docke/

https://minikube.sigs.k8s.io/docs/start/