**FastAPI-based microservices**

**Overview**

This report describes a FastAPI application that serves as a basic microservices architecture for an e-commerce platform. The application consists of three services: Product Service, Order Service, and User Service. Each service manages different aspects of the e-commerce system, allowing for modular development and deployment.

**Components**

**1. Framework**

* **FastAPI**: A modern, fast (high-performance) web framework for building APIs with Python 3.6+ based on standard Python type hints. It is designed to be easy to use and to provide automatic interactive API documentation.

**2. Services**

The application includes three primary services:

**a. Product Service**

* **Functionality**: Manages a catalog of products.
* **Endpoints**:
  + GET /products: Returns a list of products.
  + GET /products/{product\_id}: Returns details of a specific product based on its ID.

**b. Order Service**

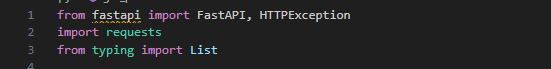
* **Functionality**: Handles order creation and management.
* **Endpoints**:
  + POST /orders: Creates an order for a product specified by product ID and quantity. It fetches product details from the Product Service.
  + GET /orders: Retrieves all orders that have been created.

**c. User Service**

* **Functionality**: Manages user information.
* **Endpoints**:
  + GET /users: Returns a list of users.
  + GET /users/{user\_id}: Returns details of a specific user based on their ID.

**Code Breakdown**

**Imports**

* **FastAPI**: To create the API endpoints and handle requests.
* **HTTPException**: To manage error responses.
* **requests**: To make HTTP requests to the Product Service.
* **List**: For type hinting to define lists in response models.

**Application Initialization**



* Initializes the FastAPI application.

**Sample Data**

python

RunCopy

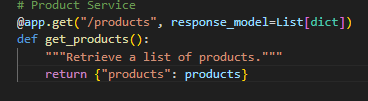
products = [...] # List of product dictionaries

users = [...] # List of user dictionaries

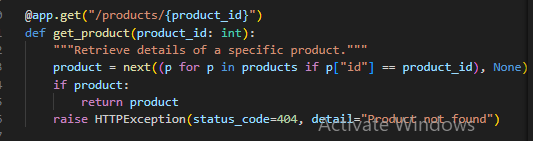
orders = [] # List to store orders

* Sample product and user data are defined as lists of dictionaries, serving as in-memory storage.

**Product Service Endpoints**

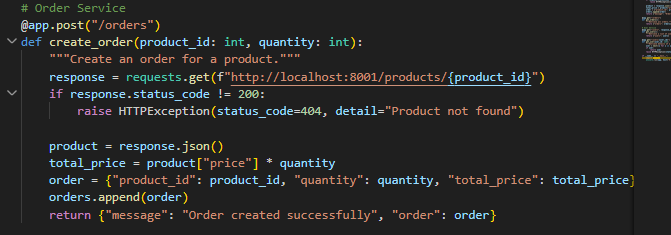
**G**

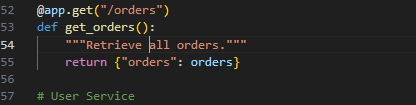
* **ET /products**: Returns the list of products.



**GET /products/{product\_id}**: Retrieves a specific product by ID, raising a 404 error if not found.

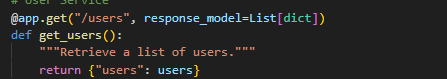
**Order Service Endpoints**

* **POST /orders**: Creates an order by fetching product details based on the product ID. It raises a 404 error if the product is not found.

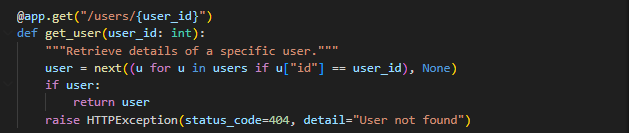


* **GET /orders**: Returns the list of all orders created.

**User Service Endpoints**



**GET /users**: Returns the list of users.

* **GET /users/{user\_id}**: Retrieves a specific user by ID, raising a 404 error if not found.

**Running the Application**



The application runs using Uvicorn, a lightning-fast ASGI server for asynchronous web applications. It listens on port 8000.

**Features**

1. **Modularity**: Each service has its own responsibility, making it easy to maintain and extend.
2. **RESTful API Design**: Services follow REST principles, using standard HTTP methods and status codes.
3. **Error Handling**: Uses HTTP exceptions to manage errors gracefully, providing clear feedback to clients.
4. **In-Memory Data Storage**: Sample data is stored in memory for demonstration; in a production setting, a database would be used.

**Limitations**

1. **In-Memory Data**: The current implementation uses in-memory data, which means data will be lost when the server restarts. A persistent database should be integrated for production use.
2. **No Authentication**: The application lacks authentication and authorization mechanisms, which are essential for a real-world application.
3. **Basic Inter-Service Communication**: The Order Service queries the Product Service using a hardcoded URL, which would need to be updated for a microservices architecture using Kubernetes.