**Microservices Architecture Deployment Report**

**Name: Mbiydzenyuy Stephen Junior**

**Introduction**

This report captures my experience and methodology in completing the Microservices Architecture Deployment Lab. The lab focused on developing and deploying a simplified microservices framework utilizing **FastAPI**, **Docker**, and **Kubernetes**. The primary objective was to construct three distinct microservices—**Product Service**, **Order Service**, and **User Service**—containerize them with Docker, and orchestrate their deployment via Kubernetes.

**Overview of Task**

For the lab, I utilized the code provided to develop a basic e-commerce application comprising three microservices:

1. **Product Service**: Responsible for managing the product catalog.
2. **Order Service**: Facilitates the creation and management of orders.
3. **User Service**: Handles user-related information.

These services were containerized with Docker and deployed on Kubernetes. The primary focus was on gaining practical insights into microservices architecture, containerization, and orchestration using Kubernetes.

**Steps Taken**

**1️⃣ Setting Up Microservices**

Using the provided code, I constructed a simple microservices framework using **FastAPI**, **Requests**, and **Uvicorn**:

* **FastAPI**: A Python framework optimized for high-performance API development.
* **Requests**: A Python library for executing HTTP requests, enabling interaction with APIs or web services.
* **Uvicorn**: An ASGI (Asynchronous Server Gateway Interface) server designed to host web applications, particularly those built with frameworks like FastAPI.

**Process for Verification**

Before advancing, I tested the microservice architecture to ensure its proper functionality:

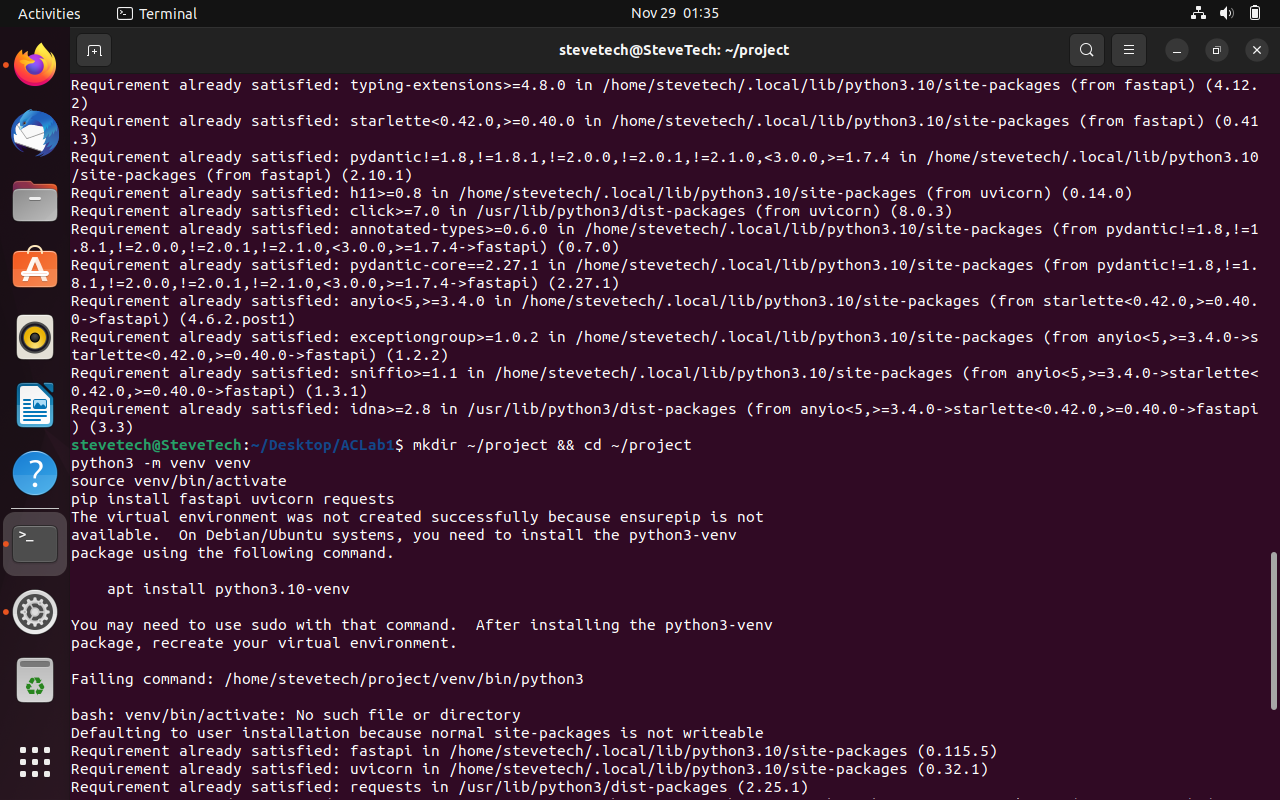
* **Installing Dependencies**: Installed all required libraries and tools.

Figure 1: Installing Uvicorn

* **Running Services with Uvicorn**: Executed the services using Uvicorn to ensure local functionality.

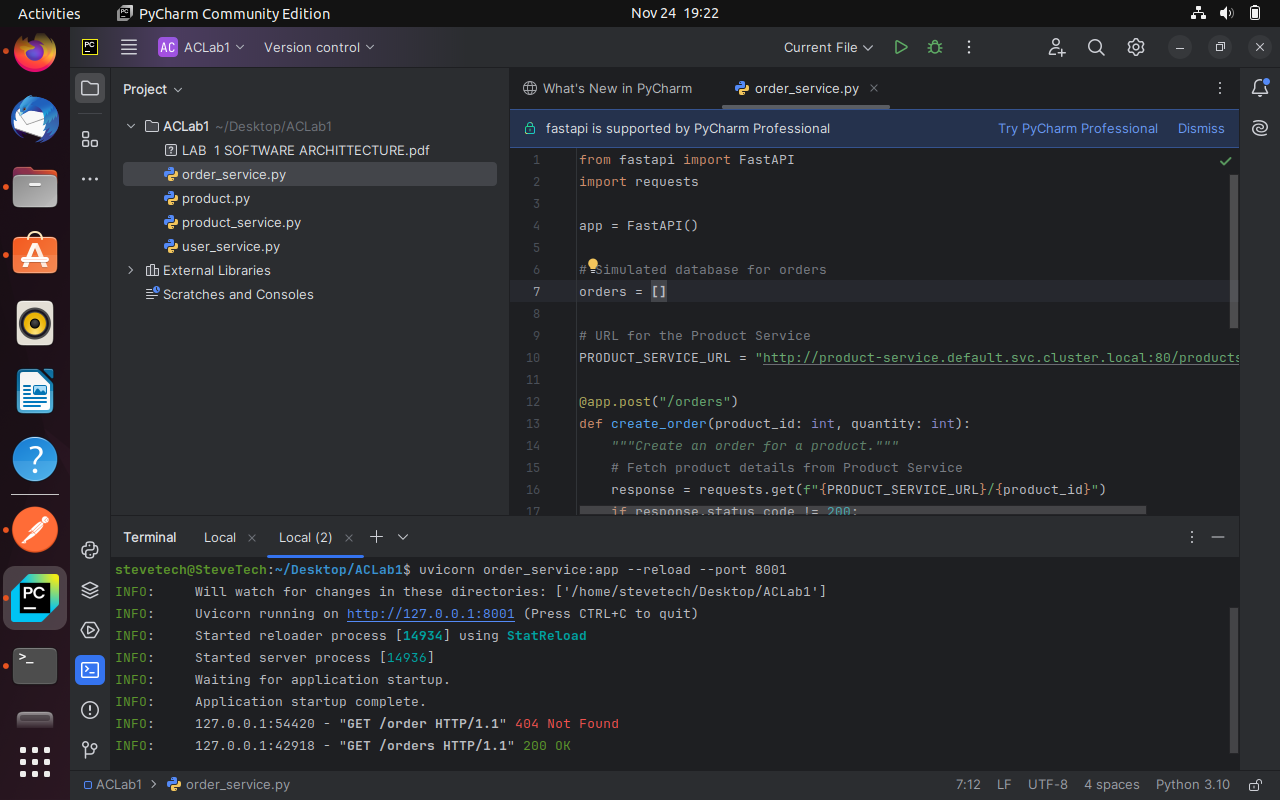


Figure 2: Running the Order Service

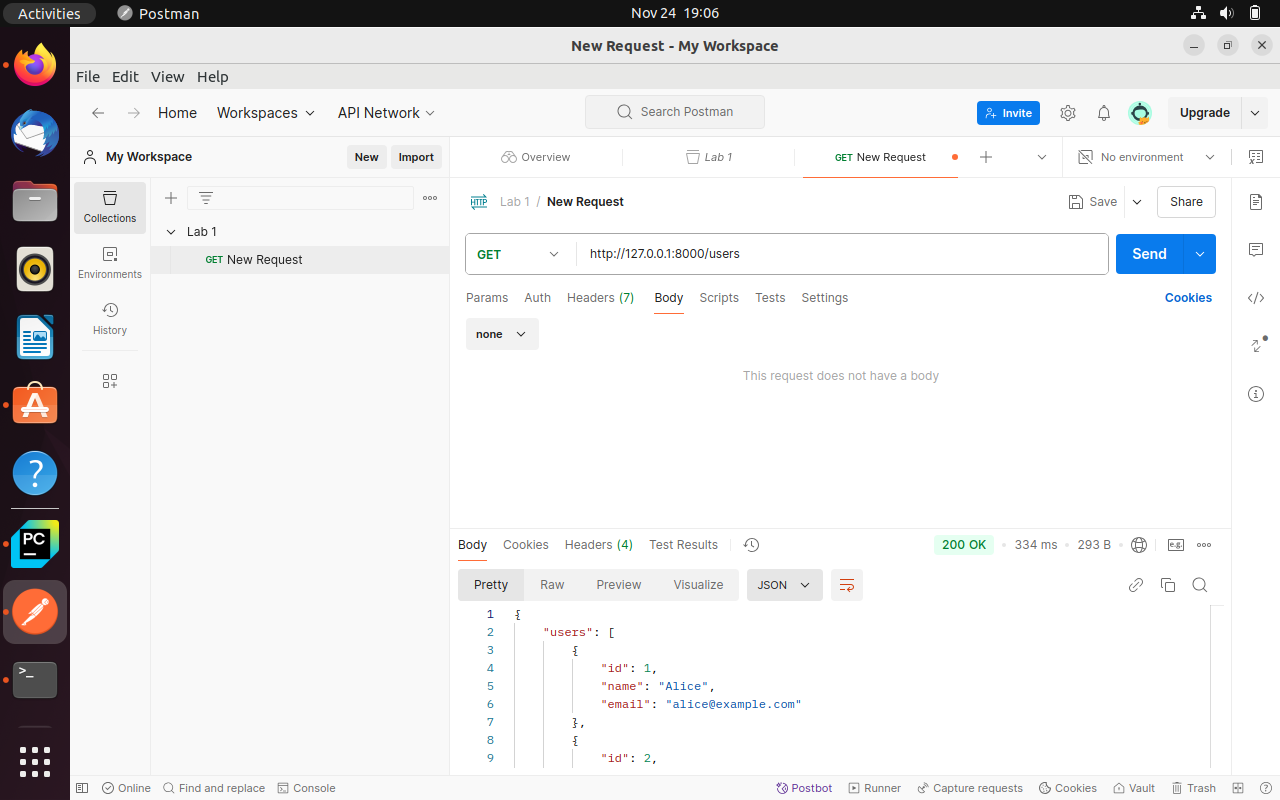
* **Testing Services with Postman**: Verified API endpoints using Postman to ensure proper service responses.  
  

Figure 3 Testing the Order Service

**2️⃣ Dockerization**

The next step involved containerizing the microservices:

* **Creating Dockerfiles**: Developed Dockerfiles to define the container environment and dependencies for each service.

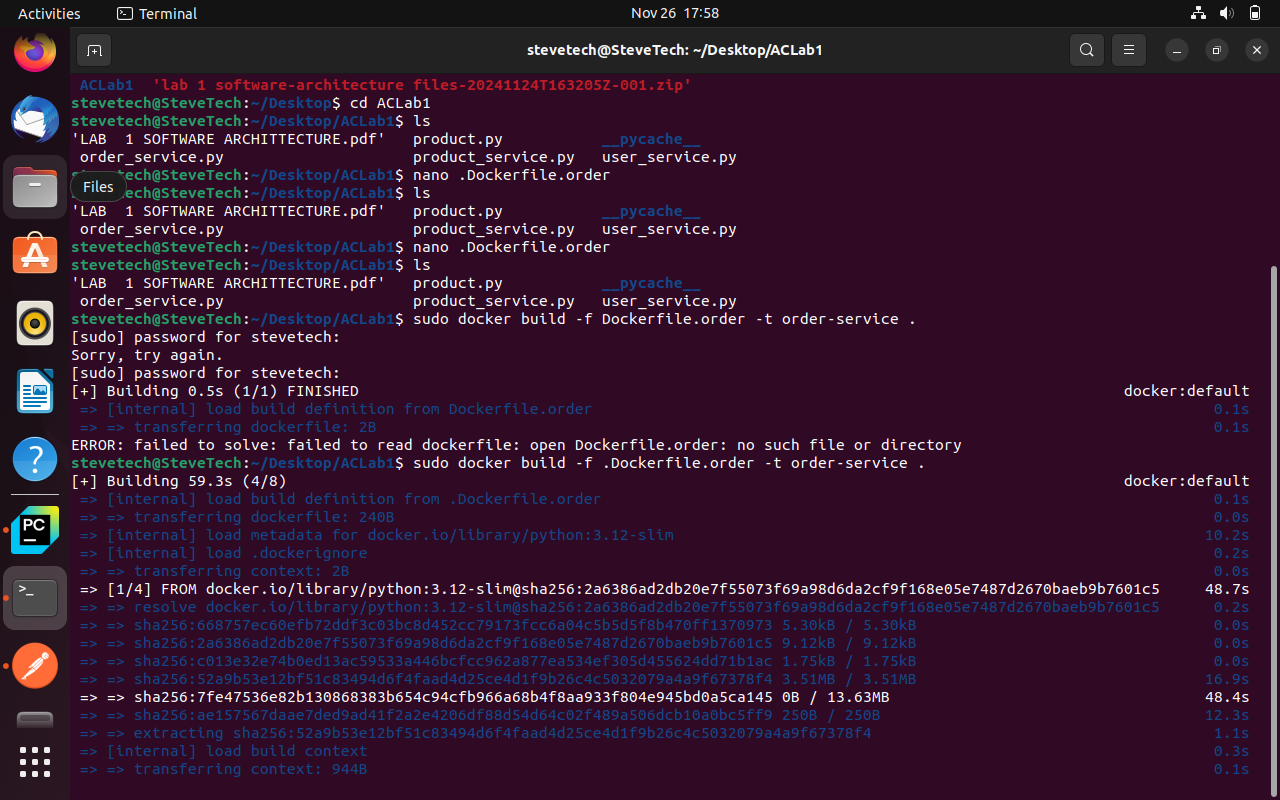
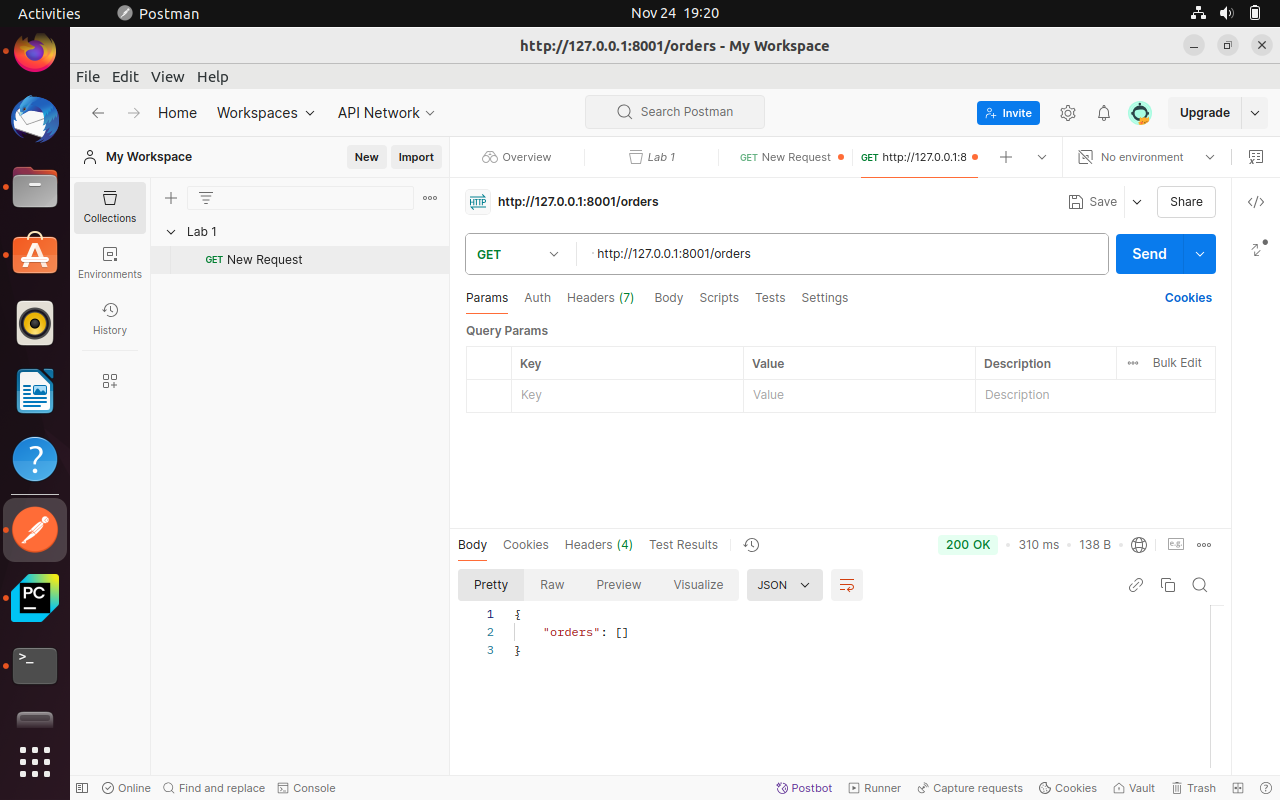


Figure 4: Creating Dockerfile for the User Service

* **Building Docker Images**: Used the Docker CLI to build images from the Dockerfiles.



* **Running Docker Containers**: Deployed the Docker containers to confirm they were working as expected.

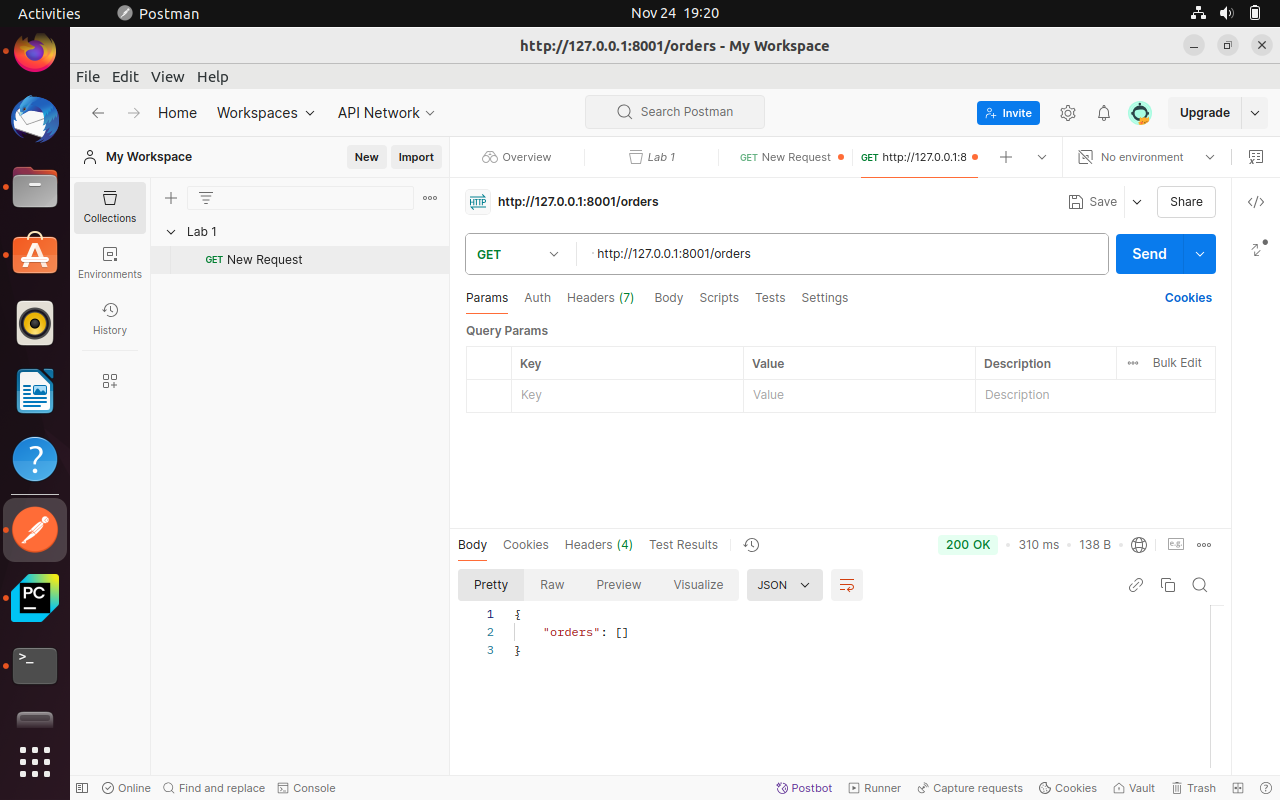


Figure 5Figure 6: Running Docker images and executing docker run for the User Service

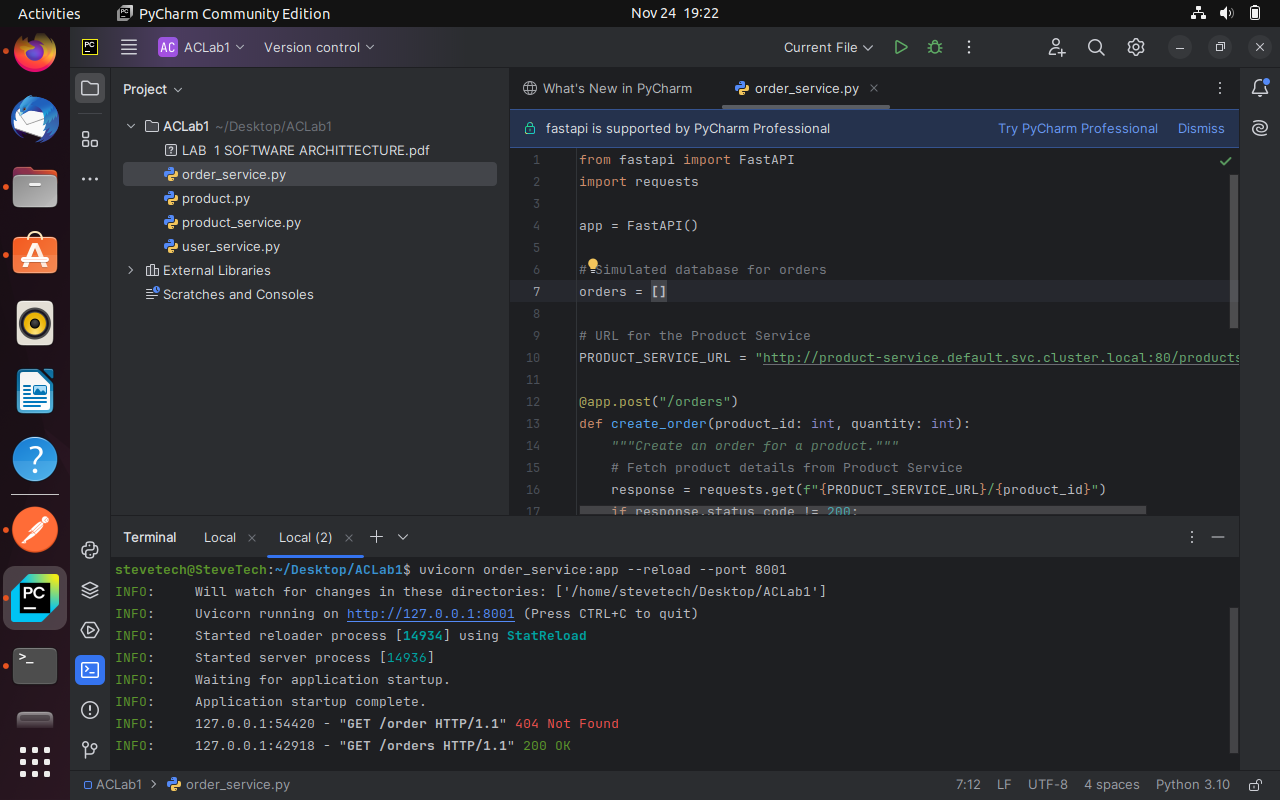
* **Testing Containers**: Used Postman to ensure that the containerized services responded correctly to API requests.
* 

Figure 6 Testing the User Service Container with Postman

**3️⃣ Kubernetes Deployment**

I created Kubernetes manifests to define the deployment configurations for each microservice. These YAML files specified:

* Number of replicas.
* Container images to deploy.
* Service endpoints for intercommunication.

*Note*: The **ClusterIP: None** setting facilitated internal communication between services within the Kubernetes cluster.

**4️⃣ Challenges Encountered**

During the deployment of the **Order Service**, I encountered a configuration error in the YAML file. The error was due to:

* Incorrectly specified port number.
* Mismatched image tag.

**Resolution**

By meticulously reviewing the YAML file, I updated the image tag and corrected the port to align with the Docker image and service configuration.

**5️⃣ Inter-Service Communication**

To ensure proper interaction between the services, I configured the **Order Service** to communicate with the **Product Service** using Kubernetes' internal DNS system. This allowed the Order Service to fetch product details seamlessly.

**6️⃣ Deployment and Testing**

After rectifying configurations, I deployed the services using Kubernetes commands:

bash

Copy code

kubectl apply -f <deployment-file>.yaml

Subsequently, I tested the services by forwarding ports to my local machine and making API calls via Postman.

**Postman Requests**

* **POST /orders**: Created new orders for products.
* **GET /orders**: Retrieved all existing orders.

**7️⃣ Final Enhancements**

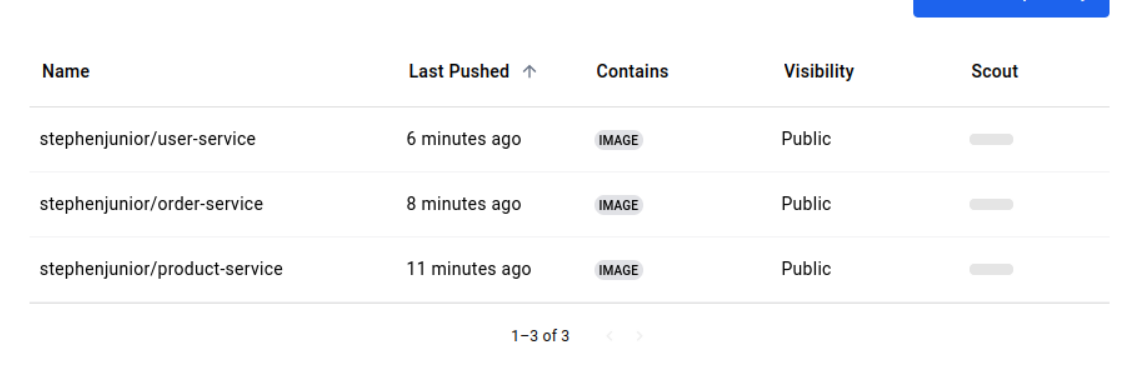
To optimize and enhance the deployment, the following additional steps were implemented:

* **API Gateway**: Set up an API Gateway to streamline routing requests to appropriate services.
* **Autoscaling**: Configured Kubernetes autoscaling to manage spikes in traffic.
* **Monitoring**: Deployed Prometheus and Grafana to monitor service performance and health.

**Screenshots**

The following visuals demonstrate various stages of the deployment:

1. Containers running successfully on the local machine after Docker image creation.



1. Kubernetes pods and services actively running in the cluster.
2. API responses from Postman verifying proper service functionality.

**Conclusion**

This lab provided practical exposure to designing and deploying a microservices architecture. The process involved:

* Building and containerizing services with Docker.
* Deploying and orchestrating them using Kubernetes.
* Establishing inter-service communication.

Despite initial challenges with YAML configuration, troubleshooting enabled the successful deployment of all services. This lab has been instrumental in strengthening my understanding of microservices architecture, containerization, and orchestration.