

Lab 9

Distributed Software Systems
UNIBO - 2024/25

Anouk Wuyts - 1900124167

Davide De Rosa - 1186536

Microservices

A **microservice** is a small, independently deployable unit of software that performs a specific function within a larger distributed system.

It is a component of the **microservices architecture**, where an application is divided into multiple loosely coupled services.

Each *microservice* focuses on a single business capability and interacts with others through well-defined APIs, often using lightweight communication protocols like **HTTP**, **gRPC**, or **messaging systems**.

Microservices Key Features

Modularity: each service handles a specific function, such as user authentication, payment processing, or inventory management.

Independence: microservices can be developed, deployed, and scaled independently of one another.

Decentralized Data Management: each service can have its own database, best suited for its specific needs.

Technology Heterogeneity: different microservices can use different programming languages, frameworks, or tools, depending on the requirements.

Fault Isolation: a failure in one microservice typically does not cause the entire system to fail.

REST API

A **REST API** (Representational State Transfer Application Programming Interface) is an architectural style for designing networked applications.

It allows communication between clients and servers using stateless operations over the **HTTP/HTTPS** protocol.

REST APIs are widely used for building distributed systems, especially web services and microservices.

We implemented REST API's in our solution using **Python** and **Flask**.

REST API Key Features

Stateless: each request from a client to a server contains all the information needed to understand and process it. The server does not retain client state between requests.

Resource-Based: resources (data or functionality) are represented by URLs (Uniform Resource Locators).

HTTP Methods: uses standard HTTP methods to perform operations on resources.

Uniform Interface: REST defines a consistent and standardized way to interact with resources.

Representation of Resources: resources can be represented in various formats such as JSON (most common), XML, or HTML.

Client-Server Model: separation of concerns allows clients and servers to evolve independently.

Cacheable: responses can be marked as cacheable to improve performance and scalability.

Docker

Docker is an open-source platform designed to automate the deployment, scaling, and management of applications using **containers**.

Containers are lightweight, portable units of software that bundle an application and its dependencies into a single package, ensuring consistency across development, testing, and production environments.

Some Docker use cases are **deploying microservices, CI/CD pipelines, DevOps** and **Cloud Native Applications**.

Docker has become a cornerstone of modern software development and DevOps practices due to its efficiency, consistency, and scalability.

Docker compose

Docker Compose is a tool that simplifies the management of multi-container Docker applications.

It allows you to define and orchestrate the deployment of multiple containers using a single YAML file, making it easy to set up complex applications with interdependent services.

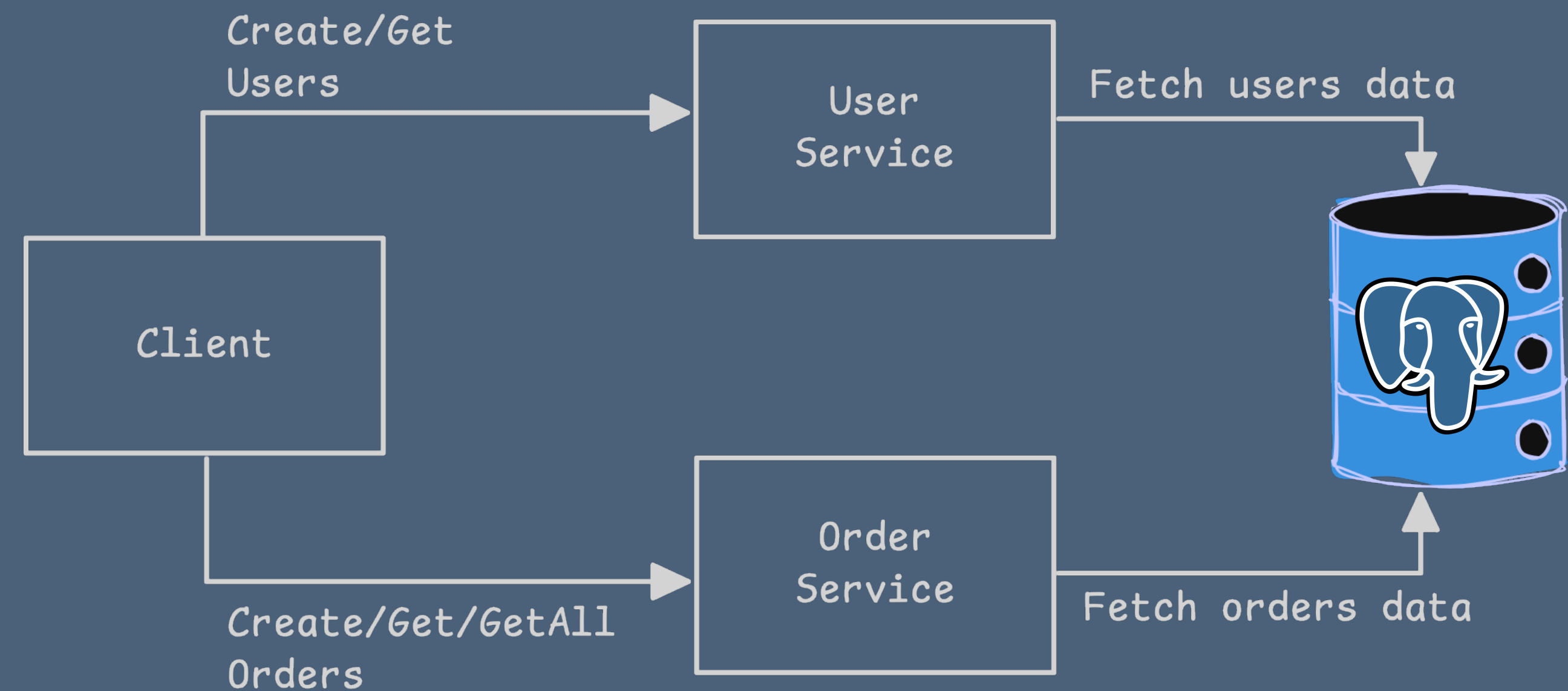
Docker Compose is a critical tool for managing containerized applications efficiently, enabling rapid development and deployment of distributed systems.

Implementation Architecture

We implemented an **User Service** where Clients can create new Users and get an user using the ID.

We also implemented an **Order Service** where Clients can create a new Order, get an order using the ID and get the list of all the Orders.

Both use the same PostgreSQL database for data persistency.



API's exposed by both services

The **User Service** exposes **two HTTP endpoints**:

- **POST users**: allows to create a new User
- **GET users/{id}**: allows to get the data of the user with that ID

The **Order Service** exposes **three HTTP endpoints**:

- **POST orders**: allows to create a new Order, connected to the User
- **GET orders**: allows to get the data of all the orders inside the database
- **GET orders/{id}**: allows to get the data of the order with that ID

Dockerfile

A **Dockerfile** is a text file that contains a set of instructions to automate the creation of a **Docker image**.

It serves as a blueprint that defines the environment, dependencies, and commands needed for the application to run within a container.

A well-written Dockerfile ensures efficient, secure, and consistent containerized applications.

Both services use the same Dockerfile to deploy the *Python* script.

Dockerfile for our services

FROM: used to define the image of the container.
We choose a slim image with only Python installed

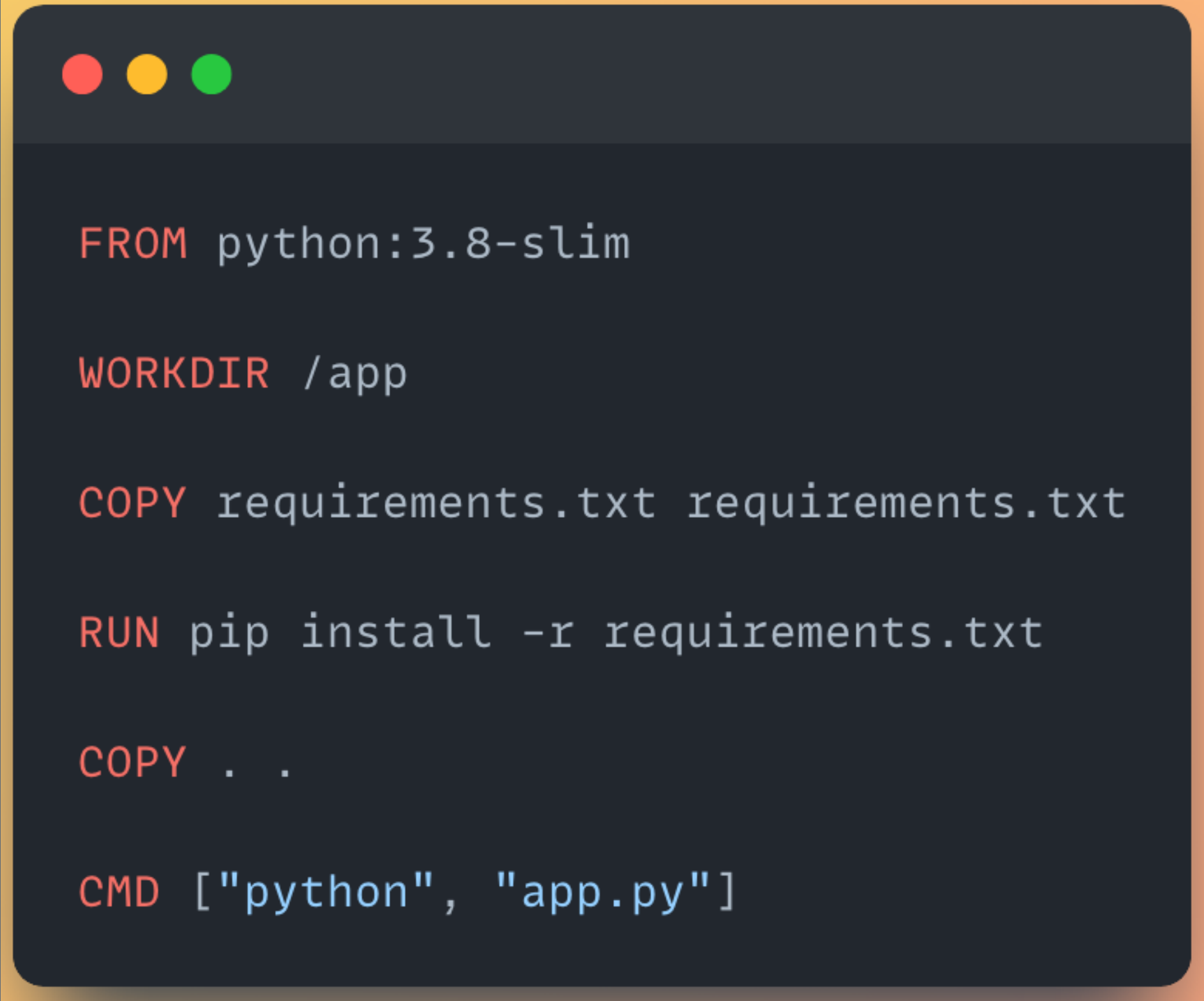
WORKDIR: changing the work directory to */app*

COPY: copying the requirements.txt file to the container

RUN: using the *pip* command to install the Python requirements, using the file that we just copied

COPY: copying the Python script inside the container

CMD: running the Python script



```
FROM python:3.8-slim

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

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

Docker compose for our services

Inside our docker-compose.yml file, we defined three different services.

Both the user and order services are defined in the same way. We declare the **Dockerfile path**, the **port mapping**, the **Database env variable**, and the fact that the two services containers are going to depend on the database container.

Exposing the two services ports allows the services to communicate with each other.

The last service is the **PostgreSQL** database, which uses a standard image and environment variables. We also define a **volume** for the **data persistency** of the database.

```
services:
  user-service:
    build: ./user
    ports:
      - "8000:5000"
    environment:
      - DATABASE_URL=postgresql://db_user:db_password@postgres-db/mydatabase
    depends_on:
      - postgres-db

  order-service:
    build: ./order
    ports:
      - "8001:5001"
    environment:
      - DATABASE_URL=postgresql://db_user:db_password@postgres-db/mydatabase
    depends_on:
      - postgres-db

  postgres-db:
    image: postgres:13
    environment:
      POSTGRES_USER: db_user
      POSTGRES_PASSWORD: db_password
      POSTGRES_DB: mydatabase
    volumes:
      - postgres_data:/var/lib/postgresql/data

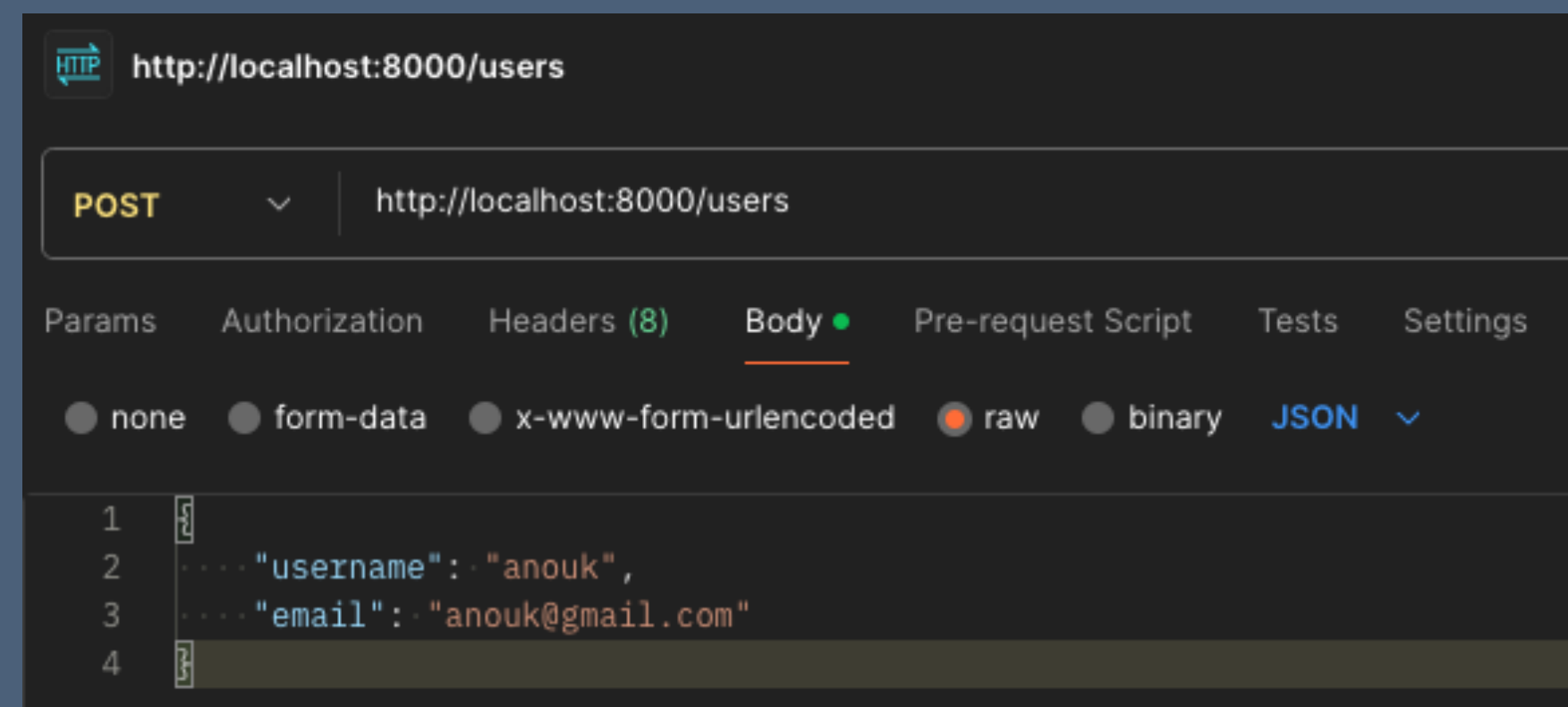
volumes:
  postgres_data:
```

Testing with Postman

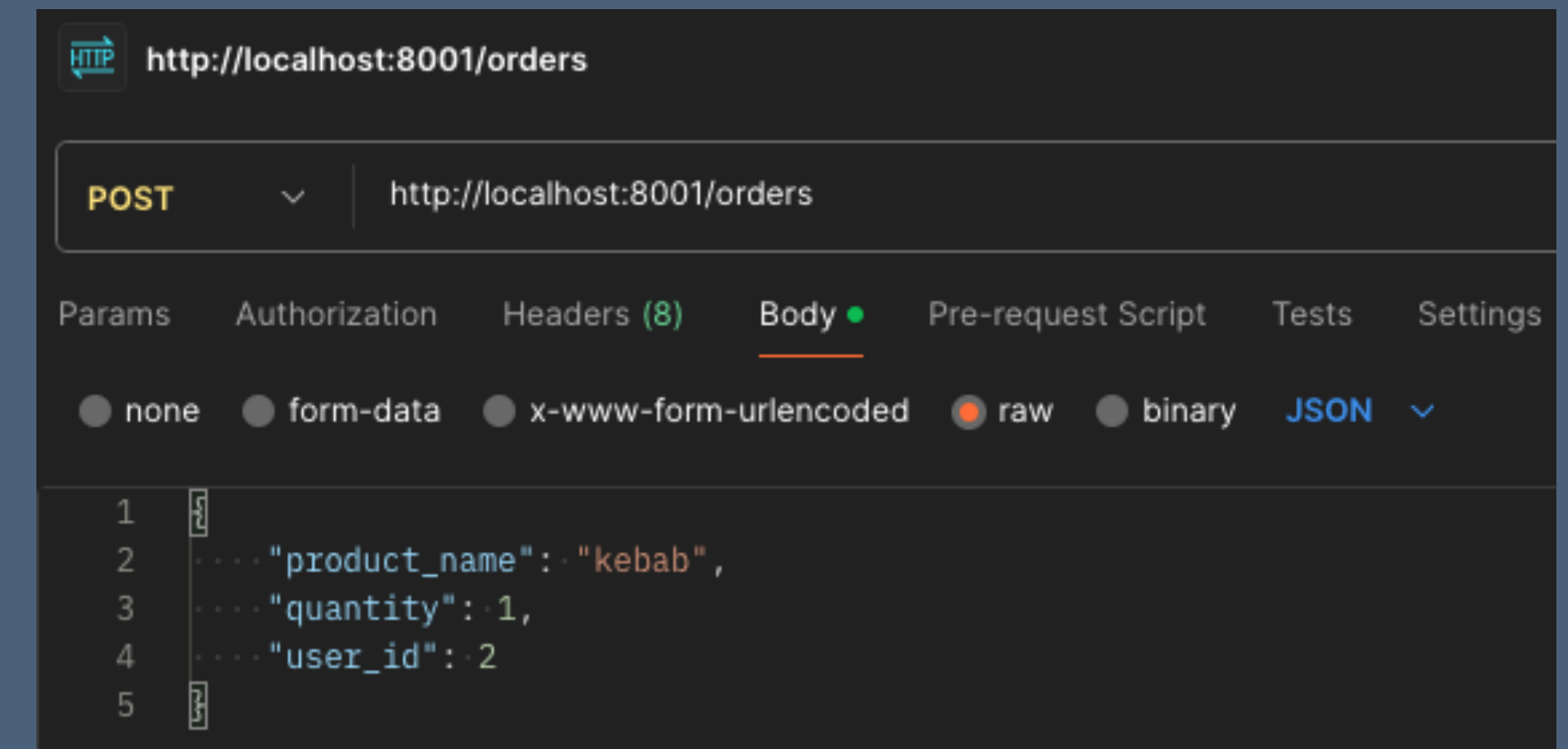
After running the **docker-compose** file using the *docker-compose up* command, everything was up and running.

To test the **REST API's** we exposed, we used **Postman**.

Beside are some examples of Postman calls with our endpoints.



```
1 {
2   "user": {
3     "email": "anouk@gmail.com",
4     "id": 1,
5     "username": "anouk"
6   }
7 }
```



```
1 {
2   "orders": [
3     {
4       "id": 1,
5       "product_name": "pizza",
6       "quantity": 2,
7       "user_id": 1
8     },
9     {
10      "id": 2,
11      "product_name": "kebab",
12      "quantity": 1,
13      "user_id": 2
14    },
15    {
16      "id": 34,
17      "product_name": "kebab",
18      "quantity": 1,
19      "user_id": 2
20    }
21  ]
22 }
```


How to monitor this services

Monitoring ensures visibility into the health and performance of your system. You can implement **observability** by focusing on three key pillars:

- **Metrics:** tools like *Prometheus* and *Grafana* can track vital metrics such as CPU/memory usage, API response times, error rates, and database performance. **Alerts** can notify teams of potential issues before they escalate.
- **Logging:** centralized logging with tools like *ELK* (Elasticsearch, Logstash, Kibana) or *Fluentd* provides insight into application behaviors, errors, and database interactions. These logs are invaluable for debugging and analyzing trends.
- **Tracing:** distributed tracing tools like *Jaeger* or *Zipkin* can map the flow of requests across services, helping identify latency issues or bottlenecks in the User-Order communication chain.

Health checks (e.g., */health* endpoints) combined with service monitoring tools or a service mesh enable proactive service management.

PostgreSQL-specific tools like pgAdmin or Datadog can monitor query performance and database health.

How to scale this services

Scaling ensures your architecture can handle increased traffic or workload spikes:

- **Horizontal Scaling:** increase the number of replicas for your microservices, distributing traffic via a **load balancer**. In Docker compose, scaling can be defined using replica settings, but orchestration tools like *Kubernetes* provide more robust scaling through features like the *Horizontal Pod Autoscaler*.
- **Vertical Scaling:** allocate more resources (CPU, memory) to containers for temporary performance boosts. For PostgreSQL, vertical scaling can improve query performance until horizontal options like read replicas or sharding become necessary.
- **Optimizations:** reduce load with caching to store frequently accessed data and use message queues for asynchronous tasks.

Thank you!