Exercise 7

Distributed Software Systems — Prof. Paolo Ciancarini
Università di Bologna
A.Y. 2023/2024
Lorenzo Campidelli, Riccardo Scotti

Introduction

We present here a microservice system for an e-commerce application. It consists of two microservices (in a real application, they would be of course way more):

- User service: to manage user data
- Order service: to manage the orders of products by users

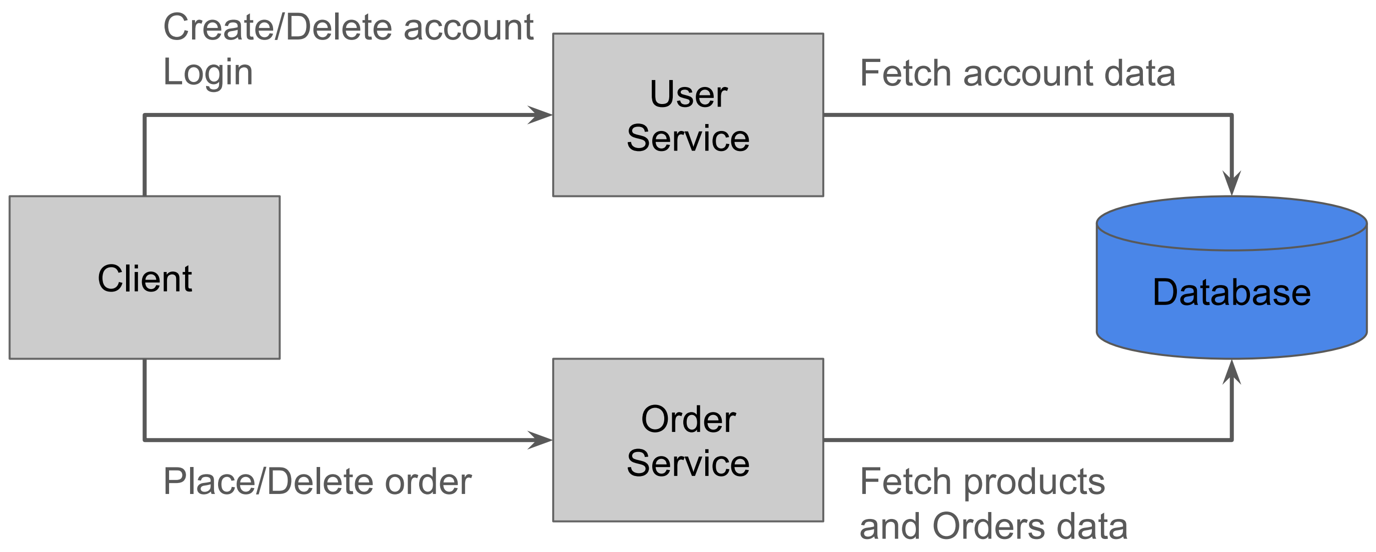
Implementation details

We implemented a service each.

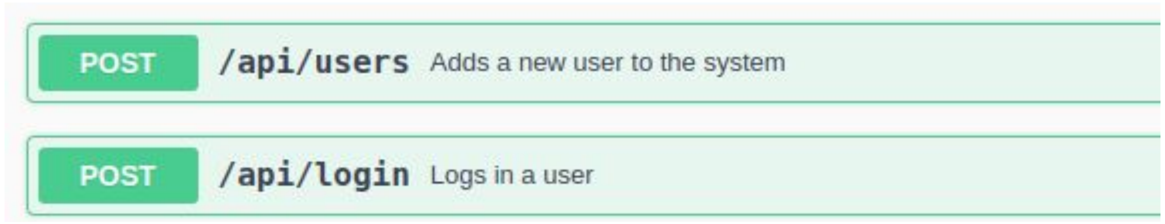
Both services are implemented in Python with **Flask**. **Flask-RESTX** has been used for RESTful API design and documentation.

Persistency has been implemented with a **Sqlite3** central database, accessed by both services via the library **Flask-SQLAlchemy**.

System architecture



User service APIs - Account management

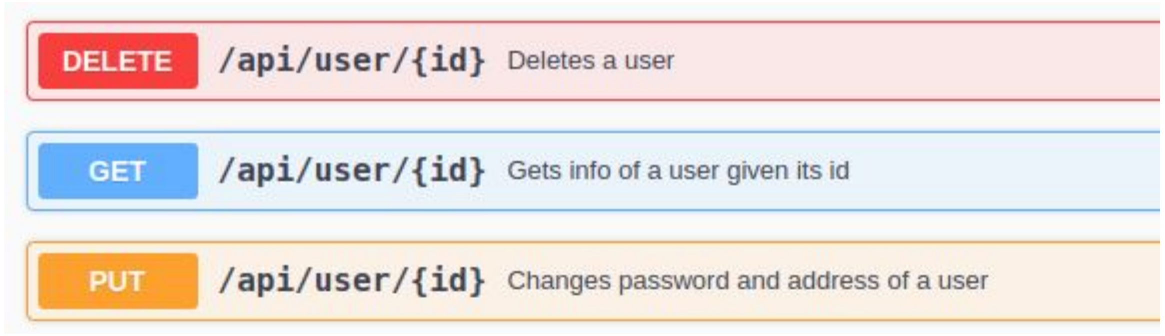

POST /api/users Adds a new user to the system

POST /api/login Logs in a user

- /users - creates a new user with username, password and address given in the payload.

- /login - logs the user into the system, returns 200 if username and password combination is correct, else 401.

User service APIs - /api/user/{id}

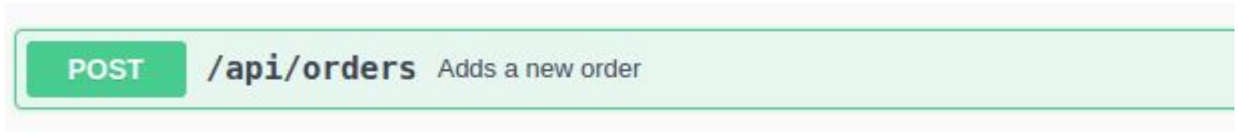

DELETE /api/user/{id} Deletes a user

GET /api/user/{id} Gets info of a user given its id

PUT /api/user/{id} Changes password and address of a user

- DELETE method - deletes the give user from the database.
- GET method - returns the information related to the user (username, password and address).
- PUT method - changes password and/or address of the user given as payload parameters.

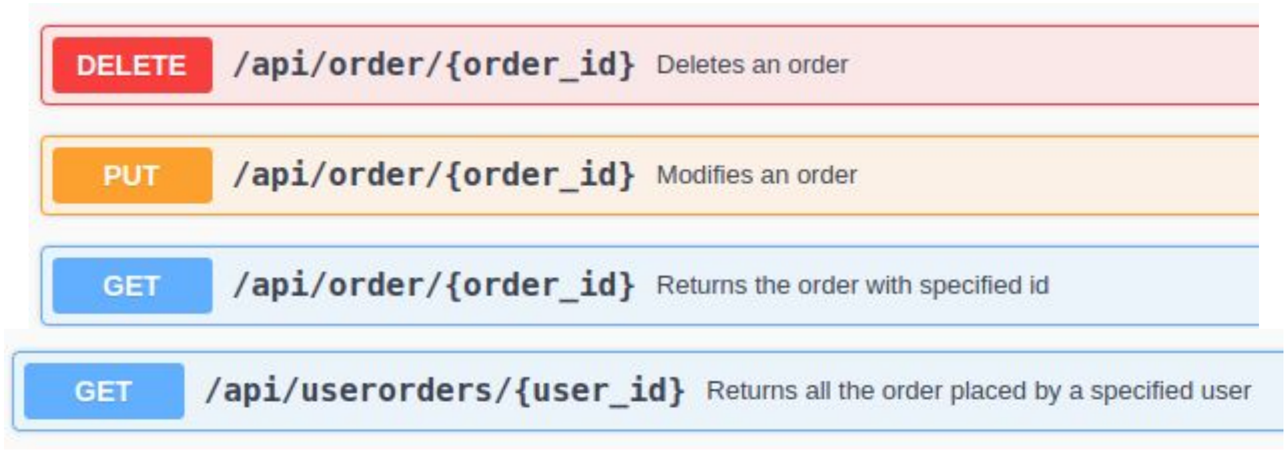
Order service APIs – Placing orders


POST /api/orders Adds a new order

Places a new order by specifying user id, product id, quantity and date of order.

The method performs checks regarding the availability of a certain product in that given quantity, returning an error code in case of failure.

Order service APIs – Placing orders



DELETE /api/order/{order_id} Deletes an order

PUT /api/order/{order_id} Modifies an order

GET /api/order/{order_id} Returns the order with specified id

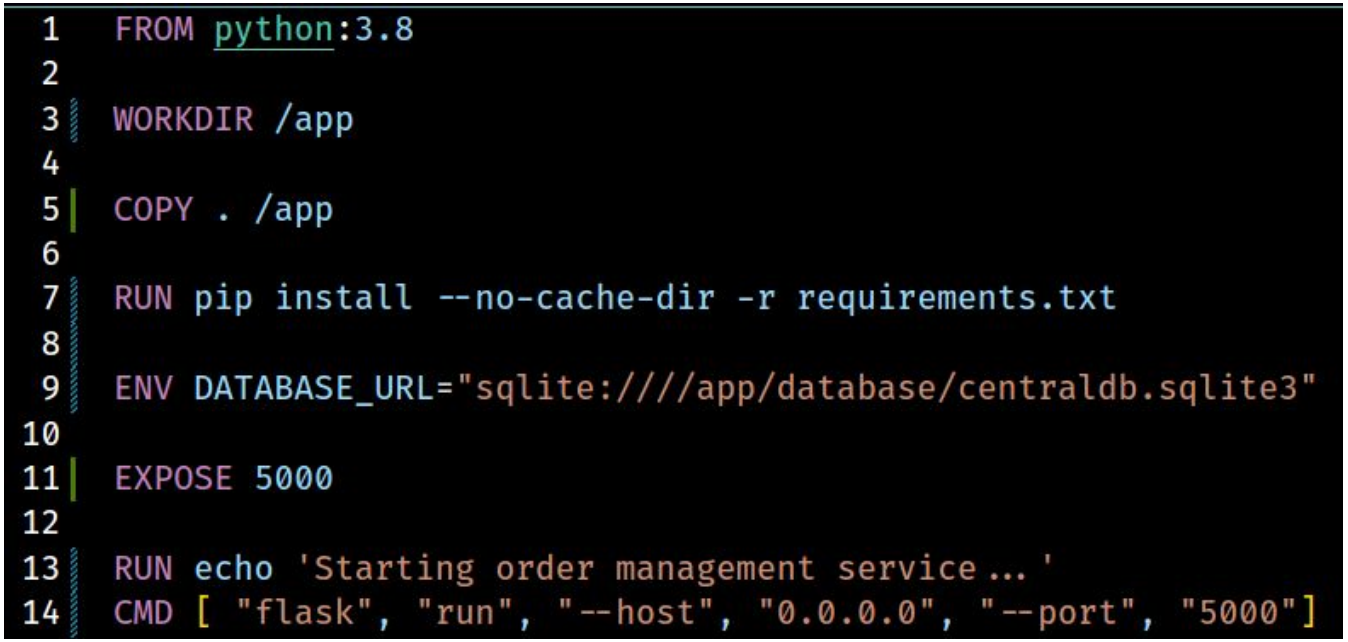
GET /api/userorders/{user_id} Returns all the order placed by a specified user

Methods to delete, modify and retrieve orders.

The delete methods also restores the available quantity of a product.

Containerization

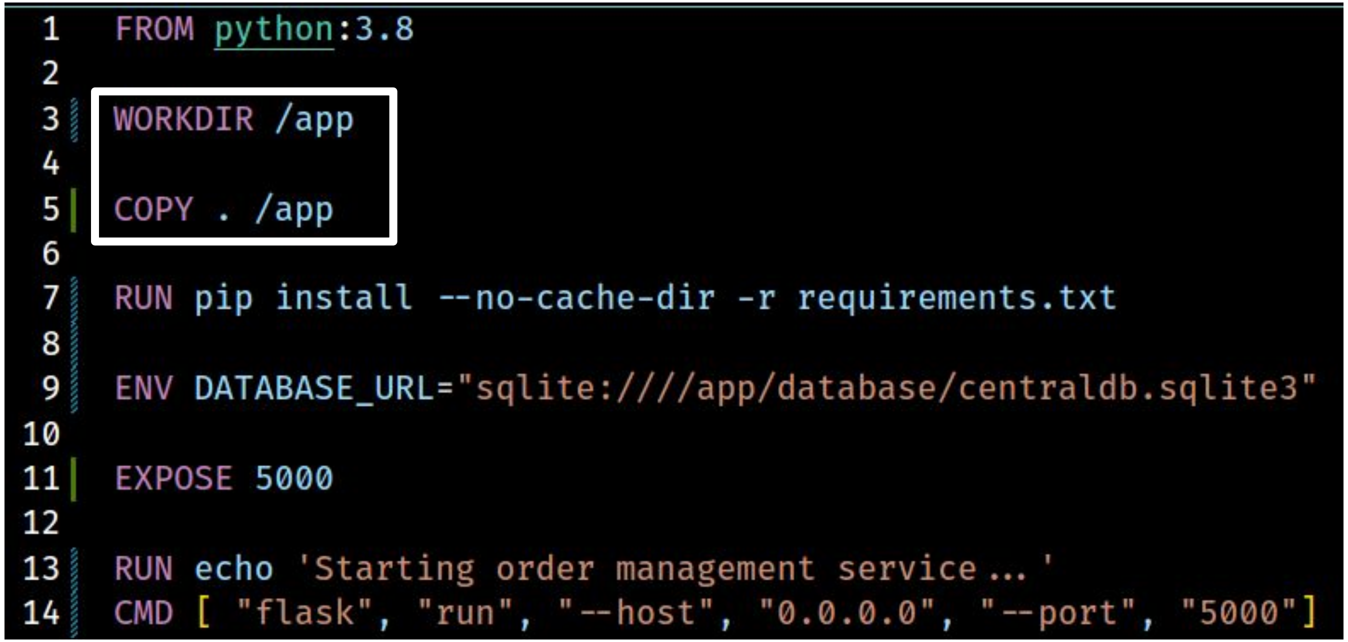
Both services run in a separated Docker container. Here we see a snippet of the configuration file (Dockerfile) for one of the services.



```dockerfile
FROM python:3.8

WORKDIR /app

COPY . /app

RUN pip install --no-cache-dir -r requirements.txt

ENV DATABASE_URL="sqlite:////app/database/centraldb.sqlite3"

EXPOSE 5000

RUN echo 'Starting order management service... '
CMD [ "flask", "run", "--host", "0.0.0.0", "--port", "5000"]
```

Containerization

Both services run in a separated Docker container. Here we see a snippet of the configuration file (Dockerfile) for one of the services.

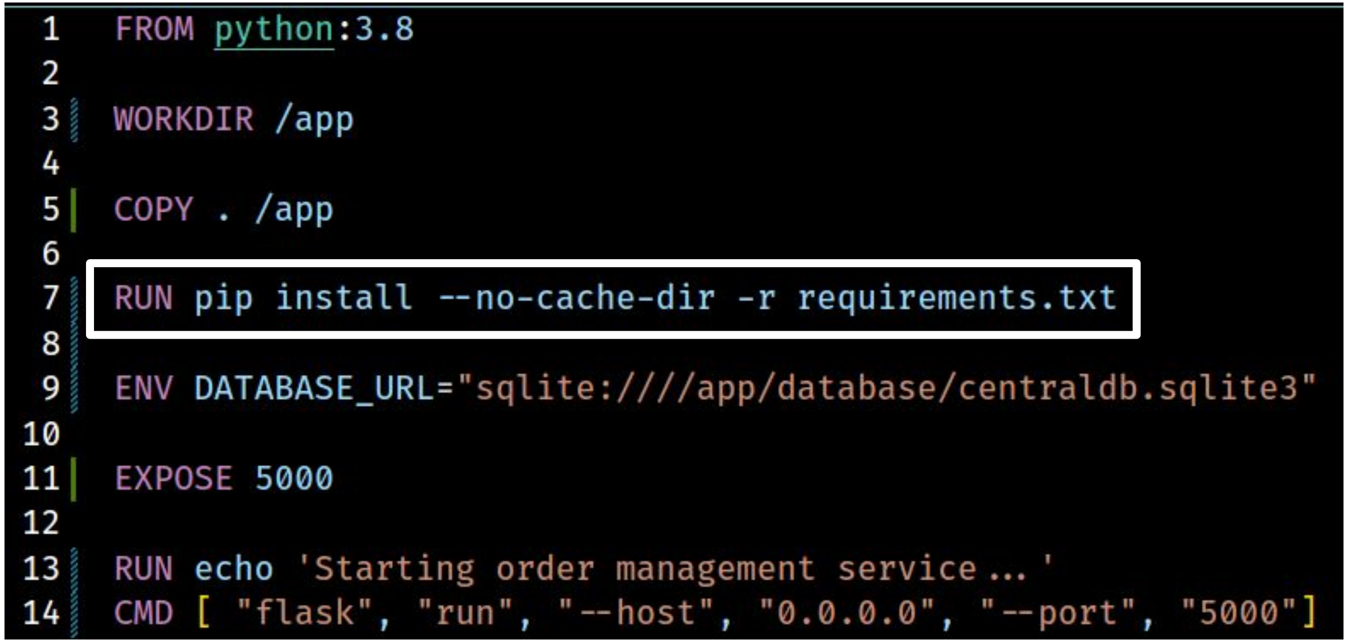


```
FROM python:3.8

WORKDIR /app

COPY . /app

RUN pip install --no-cache-dir -r requirements.txt

ENV DATABASE_URL="sqlite:////app/database/centraldb.sqlite3"

EXPOSE 5000

RUN echo 'Starting order management service... '
CMD [ "flask", "run", "--host", "0.0.0.0", "--port", "5000"]
```

creating a working environment with the code for the service

Containerization

Both services run in a separated Docker container. Here we see a snippet of the configuration file (Dockerfile) for one of the services.

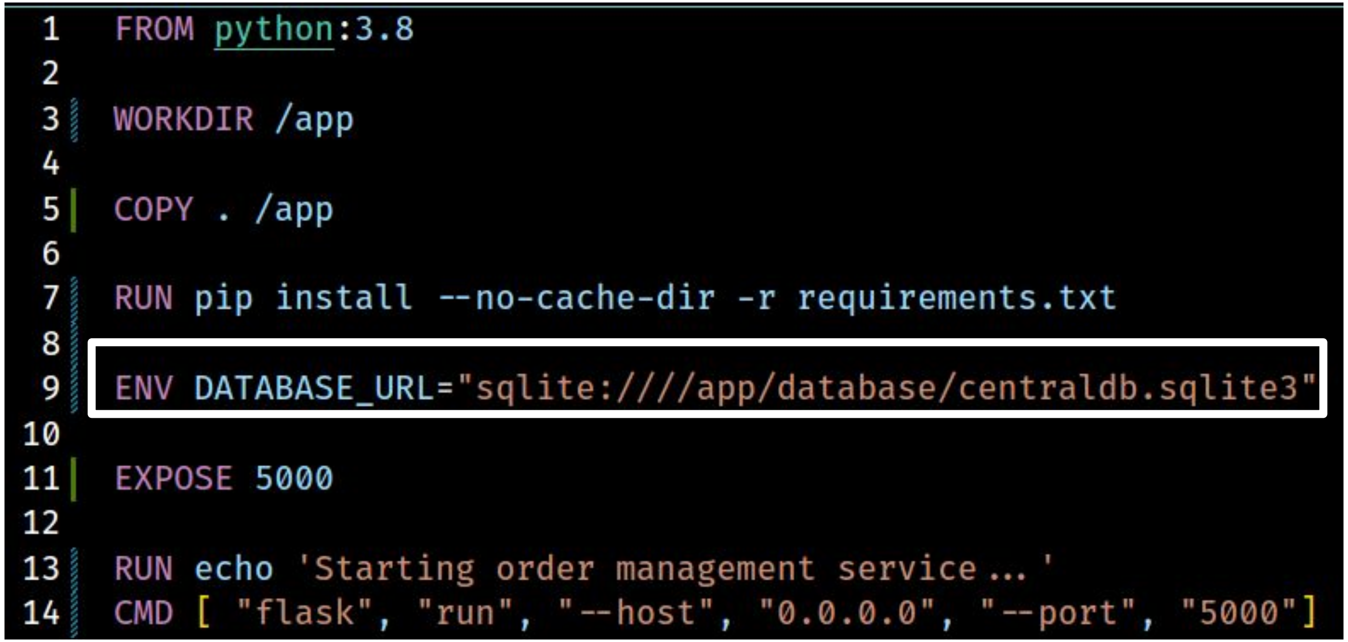


```
FROM python:3.8

WORKDIR /app

COPY . /app

RUN pip install --no-cache-dir -r requirements.txt

ENV DATABASE_URL="sqlite:////app/database/centraldb.sqlite3"

EXPOSE 5000

RUN echo 'Starting order management service... '
CMD [ "flask", "run", "--host", "0.0.0.0", "--port", "5000"]
```

installing dependencies

Containerization

Both services run in a separated Docker container. Here we see a snippet of the configuration file (Dockerfile) for one of the services.

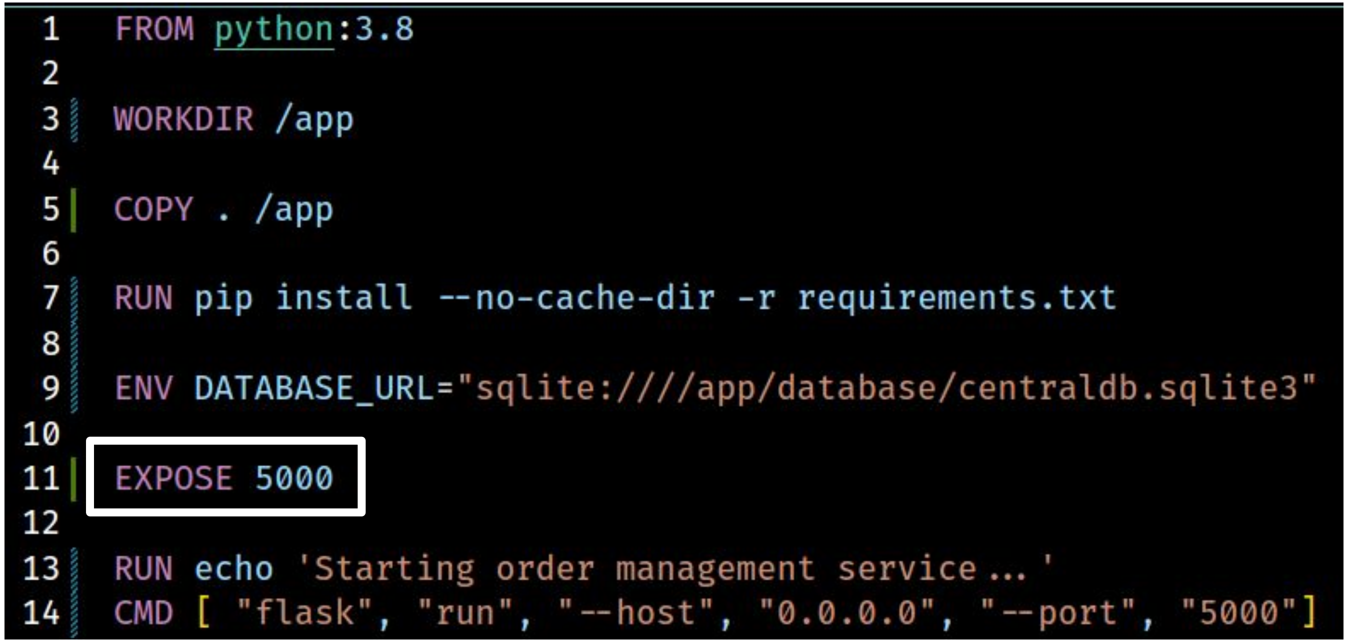


```
FROM python:3.8

WORKDIR /app

COPY . /app

RUN pip install --no-cache-dir -r requirements.txt

ENV DATABASE_URL="sqlite:////app/database/centraldb.sqlite3"

EXPOSE 5000

RUN echo 'Starting order management service... '
CMD [ "flask", "run", "--host", "0.0.0.0", "--port", "5000"]
```

setting environment variable for DB location

Containerization

Both services run in a separated Docker container. Here we see a snippet of the configuration file (Dockerfile) for one of the services.

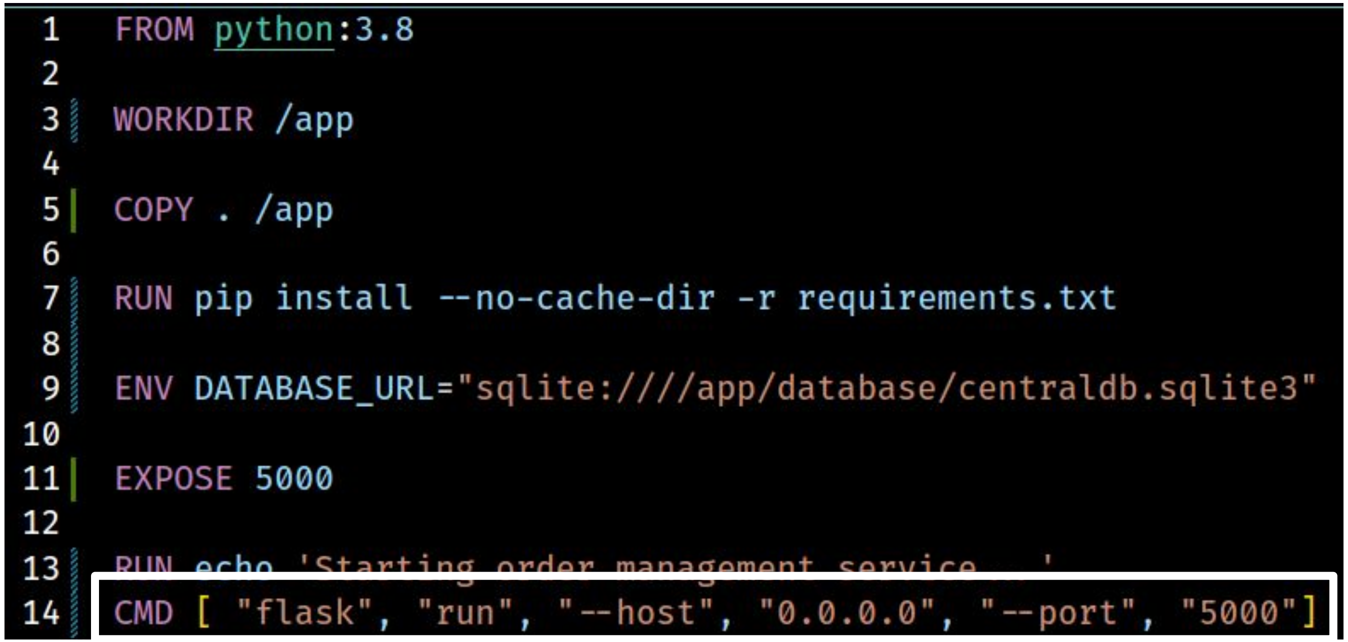


```
FROM python:3.8

WORKDIR /app

COPY . /app

RUN pip install --no-cache-dir -r requirements.txt

ENV DATABASE_URL="sqlite:////app/database/centraldb.sqlite3"

EXPOSE 5000

RUN echo 'Starting order management service... '
CMD [ "flask", "run", "--host", "0.0.0.0", "--port", "5000"]
```

setting exposed port(s)

Containerization

Both services run in a separated Docker container. Here we see a snippet of the configuration file (Dockerfile) for one of the services.

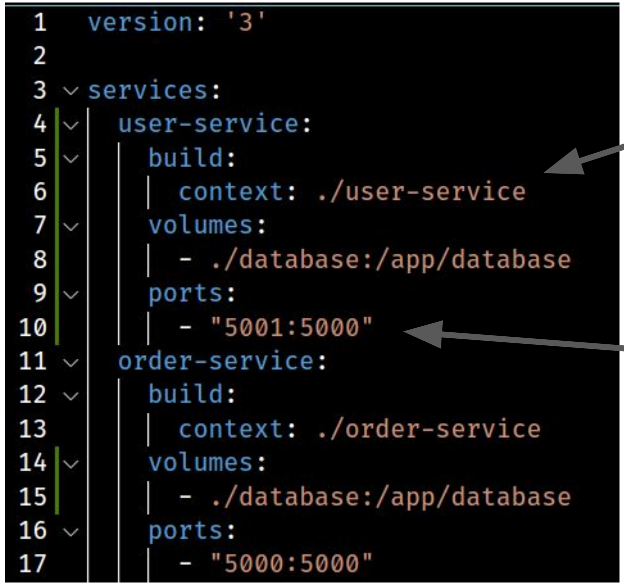


```
FROM python:3.8

WORKDIR /app

COPY . /app

RUN pip install --no-cache-dir -r requirements.txt

ENV DATABASE_URL="sqlite:////app/database/centraldb.sqlite3"

EXPOSE 5000

RUN echo 'Starting order management service...'
CMD [ "flask", "run", "--host", "0.0.0.0", "--port", "5000"]
```

running the entry point of the service

Orchestration

Docker Compose has been used to orchestrate the two services. It takes care of building and running them, managing connection with the database and port mapping.



```
version: '3'

services:
  user-service:
    build:
      context: ./user-service
    volumes:
      - ./database:/app/database
    ports:
      - "5001:5000"
  order-service:
    build:
      context: ./order-service
    volumes:
      - ./database:/app/database
    ports:
      - "5000:5000"
```

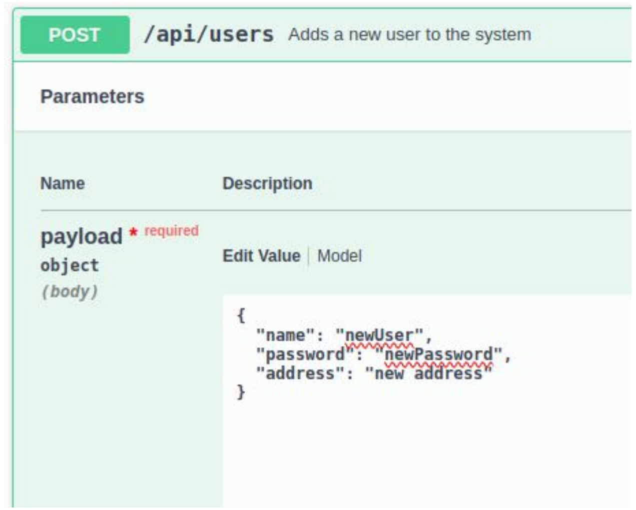
we set a database external to the container of the service as a volume, to make it accessible to both services

this service, which runs on port 5000 in its container, will be accessible at port 5001

Testing

We will test the system with the API webpage provided automatically by Flask-RESTX; it serves both as a documentation reference and as a testing platform. It is by all means equivalent to sending requests from a client or from specialized apps such as Postman or Insomnia.

Testing – adding a new user



POST /api/users Adds a new user to the system

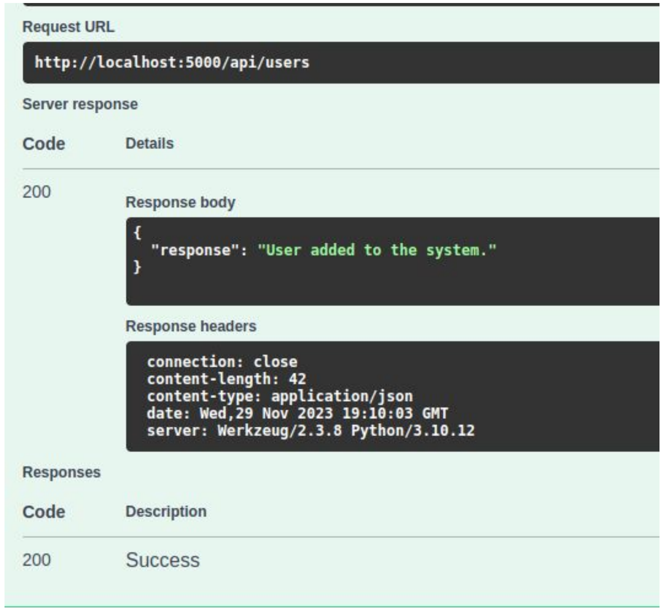
Parameters

Name | Description

payload * required
object
(body)

Edit Value | Model

```
{
  "name": "newUser",
  "password": "newPassword",
  "address": "new address"
}
```



Request URL

```
http://localhost:5000/api/users
```

Server response

Code | Details

200

Response body

```
{
  "response": "User added to the system."
}
```

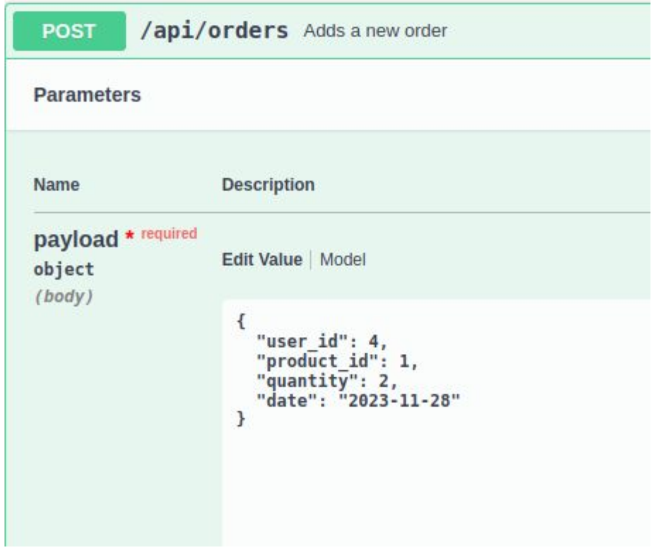
Response headers

```
 connection: close
 content-length: 42
 content-type: application/json
 date: Wed,29 Nov 2023 19:10:03 GMT
 server: Werkzeug/2.3.8 Python/3.10.12
```

Responses

Code	Description
200	Success

Testing – placing an order for a user

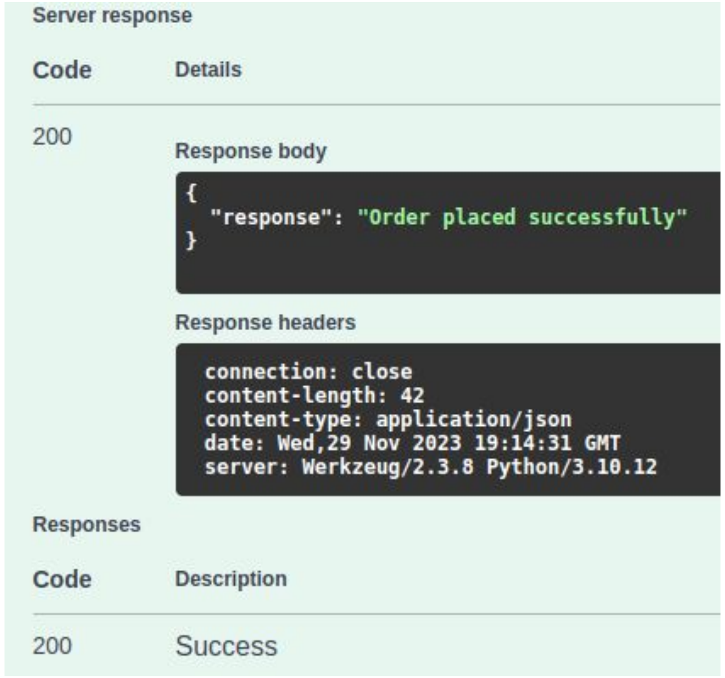


POST `/api/orders` Adds a new order

Parameters

Name	Description
payload * required object *(body)*	Edit Value | Model

```
{
  "user_id": 4,
  "product_id": 1,
  "quantity": 2,
  "date": "2023-11-28"
}
```



Server response

Code	Details
200	Response body

```
{
  "response": "Order placed successfully"
}
```

Response headers

```
connection: close
content-length: 42
content-type: application/json
date: Wed,29 Nov 2023 19:14:31 GMT
server: Werkzeug/2.3.8 Python/3.10.12
```

Responses

Code	Description
200	Success

Conclusions and further improvements

- the microservice architecture allows the system to be extended seamlessly
  - adding an inventory microservice, to manage products, would simply require to design other APIs and connect to the already present DB, without any other form of configuration for the other microservices
  - the DB offers an interface to allow the different services to communicate easily
- if one service fails, the other can still run as it restarts
- orchestration tools such as Docker compose or Kubernetes can make the system scale up automatically, by means of specification in the configuration files
- they also offer monitoring tools, to determine which microservices and by what amount need to be scaled up