

Service Architecture



1. Introduction.....	3
2. Concepts.....	4
2.1. SOA.....	4
2.2. ROA.....	6
2.3. Microservices.....	9
3. The volunteer application.....	15
3.1. What is the purpose.....	15
3.2. Development decisions and deployment instructions.....	15
3.3. Services.....	15
3.3.1. Database.....	15
3.3.2. Requests.....	17
3.3.3. Responses.....	19
3.3.4. Feedbacks.....	21
3.3.5. Administrators, Users and Volunteers.....	23
3.4. Our final application.....	24
3.4.1. Login.....	24
3.4.2. User.....	28
3.4.3. Administrator.....	32
3.4.4. Volunteer.....	34
3.5. DevOps.....	36
4. Conclusion.....	40

1. Introduction

In today's interconnected digital world, service architectures have become essential for building scalable and modular applications. Throughout this report, we delve into the concepts of service-oriented architectures (SOA), resource-oriented architectures (ROA), and microservices. Each of these approaches offers unique advantages and trade-offs, progressing from monolithic applications toward more flexible solutions. To solidify these concepts, we applied them concretely by developing a volunteer application aimed at bridging the gap between those in need and those willing to help. This project served as a practical exploration of architecture design, implementation, and deployment, incorporating key principles of scalability and reliability.

2. Concepts

In this section we will focus on the concepts that we have discovered in the MOOCs and Tutorials. We will start by introducing what a service-oriented architecture is, then move on to service-oriented architectures, and finally microservices.

2.1. SOA

A service-oriented architecture is a design pattern that allows different applications to call on services that may be common. These services will integrate one or more features that the application will need

Before the creation of this model, applications were very little modular and scalable because a feature often had to be adapted to the application that needed it. Moreover the modification of this feature could require modifying the code bricks that surrounded it accordingly.

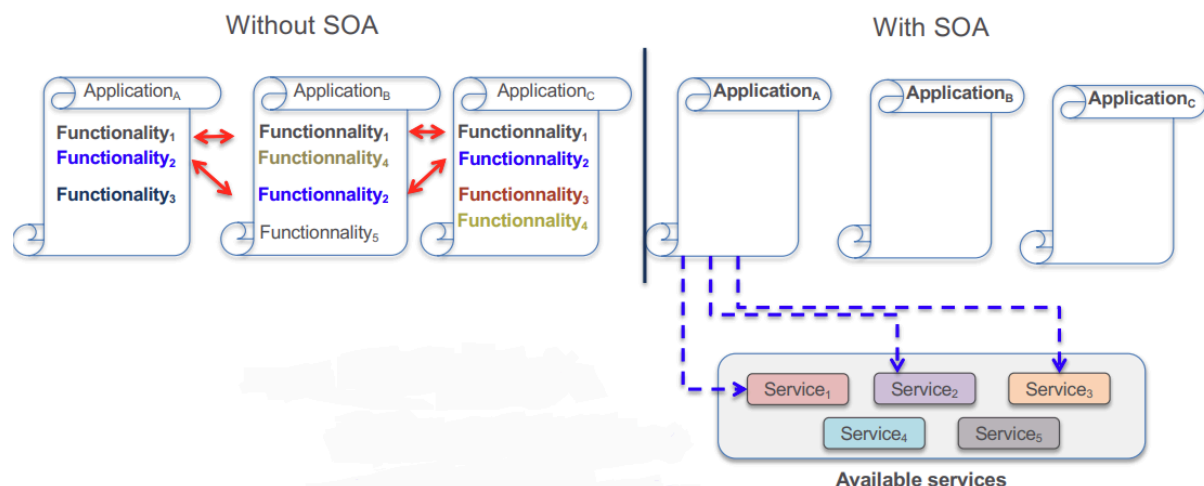


Figure 1: Without SOA and with SOA differences

The services we are going to be interested in are called WEB services since they are on servers accessible via the Internet. To access them we can use SOAP for Simple Object Access Protocol which is a protocol that defines the exchange of messages between the WEB service and the rest even if they are developed in different languages. SOAP uses WSDL which is an XML-based language that describes WEB services and how to call them from your application without needing to know the source code.

We have created a SOAP Web service in order to put this new knowledge into practice. Using Eclipse we have created a simple Java class that returns the length of the typed string:

```

1 package fr.insa.soap;
2
3 import javax.jws.WebMethod;
4 import javax.jws.WebParam;
5 import javax.jws.WebService;
6
7 @WebService(serviceName="analyser")
8 public class AnalyserChaineWS {
9     @WebMethod(operationName="compare")
10     public int analyser(@WebParam(name="chain") String chaine) {
11         return chaine.length();
12     }
13 }
14

```

Figure 2: Snapshot of our Java class

As we can see on the snapshot below, we find the description of our class with its name, its methods and parameters:

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```

<!-- Published by JAX-WS RI (https://github.com/eclipse-ee4j/metro-jax-ws). RI's version is JAX-WS RI 2.3.6 git-revision#d201abe. -->
<xs:schema xmlns:tns="http://soap.insa.fr/" xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0" targetNamespace="http://soap.insa.fr/">
  <xs:element name="compare" type="tns:compare"/>
  <xs:element name="compareResponse" type="tns:compareResponse"/>
  <xs:complexType name="compare">
    <xs:sequence>
      <xs:element name="chain" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="compareResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>

```

Figure 3: Snapshot of our class envelope

We can now access this method via a client described like this:

```

package fr.insa.soap;

import java.net.MalformedURLException;
import java.net.URI;
import java.net.URL;

import fr.insa.soap.wsdltojava.AnalyserChaineWS;
import fr.insa.soap.wsdltojava.Analyzer;

public class ClientOfAnalyzer {

    public static void main(String [] args) throws MalformedURLException {
        //l'adresse du service Web
        final String adresse="http://localhost:8089/analyzer";

        //Création de l'URL
        final URL url=URI.create(adresse).toURL();

        //Instanciation de l'image du service
        final Analyzer service= new Analyzer(url);

        //Création du proxy (en utilisant le portType) pour l'appel du service
        final AnalyserChaineWS port = service.getPort(AnalyserChaineWS.class);

        String chaine ="aaaa";
        //appel de la méthode compare via le port
        System.out.println("La taille de la chaine "+chaine+" est "+port.compare("aaaa"));
    }
}

```

Figure 4: Snapshot of the client code

The SOAP protocol works with what is called an envelope in which we can find a lot of information on the class that we want to address

2.2. ROA

The problem with SOA is that they are quite complex to implement because each request goes through a SOAP envelope that can take up a lot of space on a limited bandwidth. In addition, the XML language is very verbose, which does not reduce its size.

REST, which we will present, is part of resource-oriented architecture. This type of architecture is based on the fact that each element accessible via its own URL is identified as a resource that can be accessed and manipulated via HTTP methods such as GET, POST, etc. Here, REST is not limited to XML for its exchange between services since a resource is capable of returning XML, JSON and other types of data. REST is also much more flexible and makes interaction with the user more interesting with its various possible requests. This is also a weakness for REST since this “informality” makes it less reliable in terms of security.

In order to better highlight the difference between SOAP and REST in terms of complexity, let's take the following class as an example:

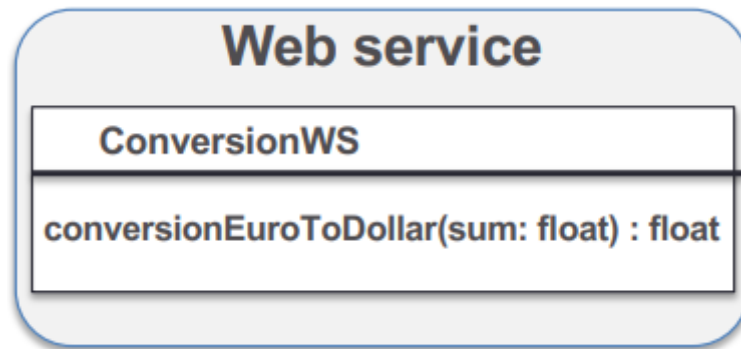


Figure 5: ConversionWS Java class

Now let's see how SOAP and REST make a request to the *conversionEuroToDollar* method:

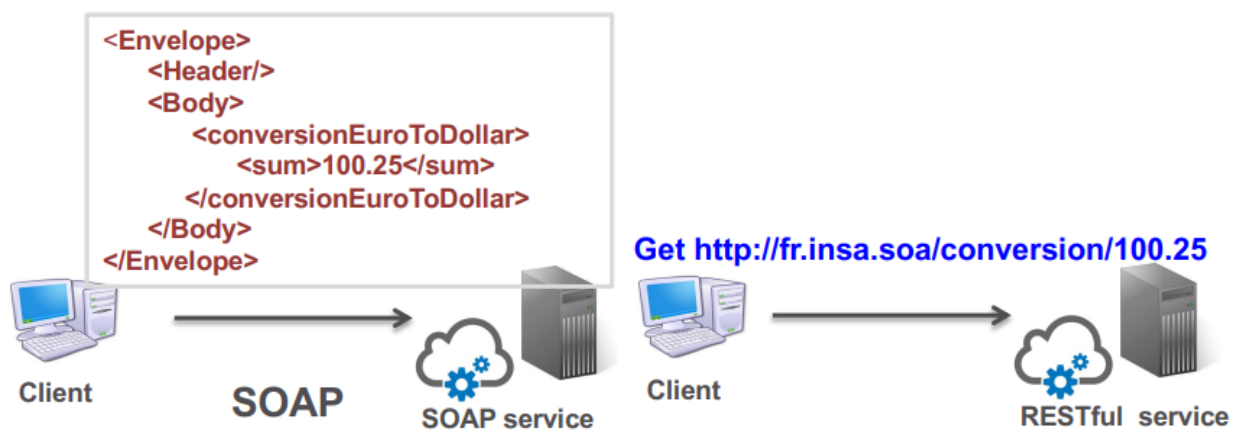


Figure 6: Request differences between SOAP and REST

As you can see, a SOAP request is very verbose because it requires an envelope, whereas REST only uses a simple HTTP request.

So we set up a web server on our localhost in order to test the REST architecture. As you can see below, we can access the different resources of our application via our "webapi" URL start:

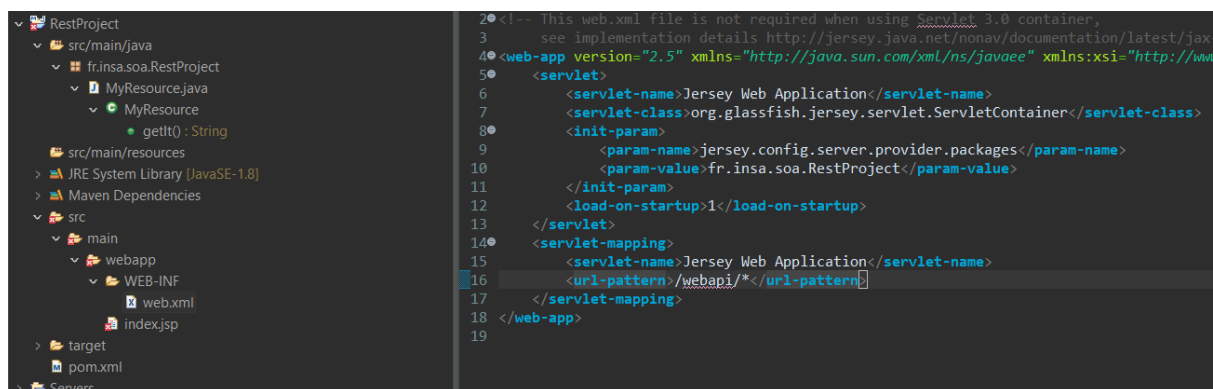


Figure 7: Snapshot of our service web.xml

This web.xml allow the connection between the URL and our service, which is why we can easily call our method:

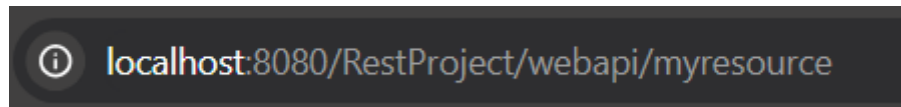


Figure 8: Our service URL

Here we make a very simple request for access to the site but we can also place a parameter for a method which, for example, returns the size of the character string given in the URL:

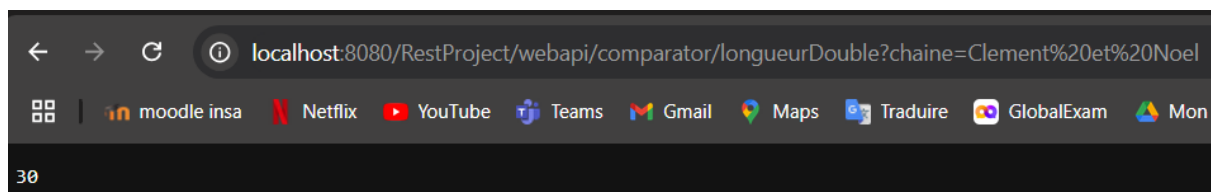


Figure 9: Snapshot of the method result for a parameter entered

Here the returned data is an integer but we can also retrieve as said earlier JSON with a student class and its method which simply returns the values of two parameters:

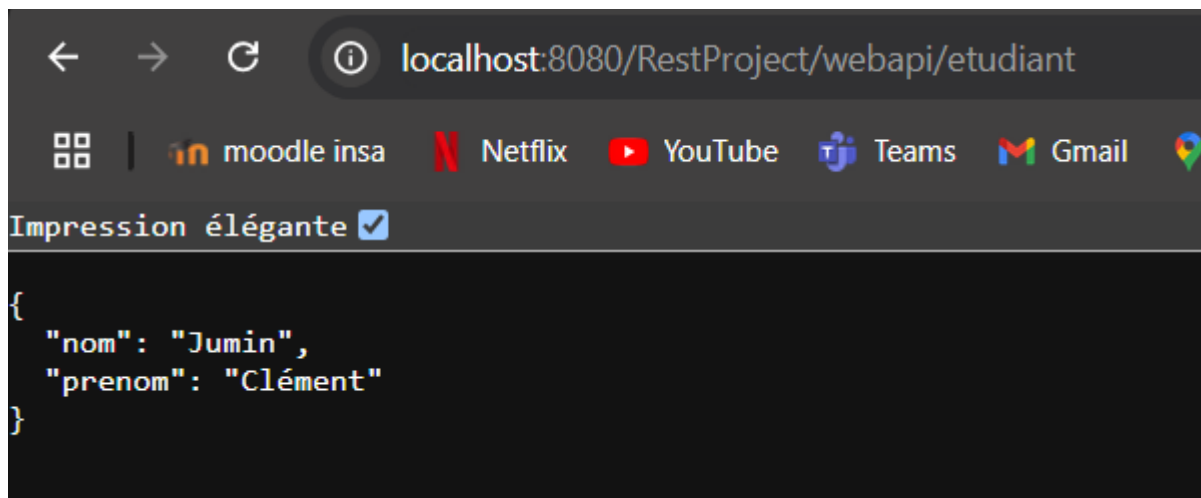


Figure 10: Snapshot of the method returning JSON data

Here the majority of HTTP options that we presented were GET but other options are available like POST which allows to update an existing data:

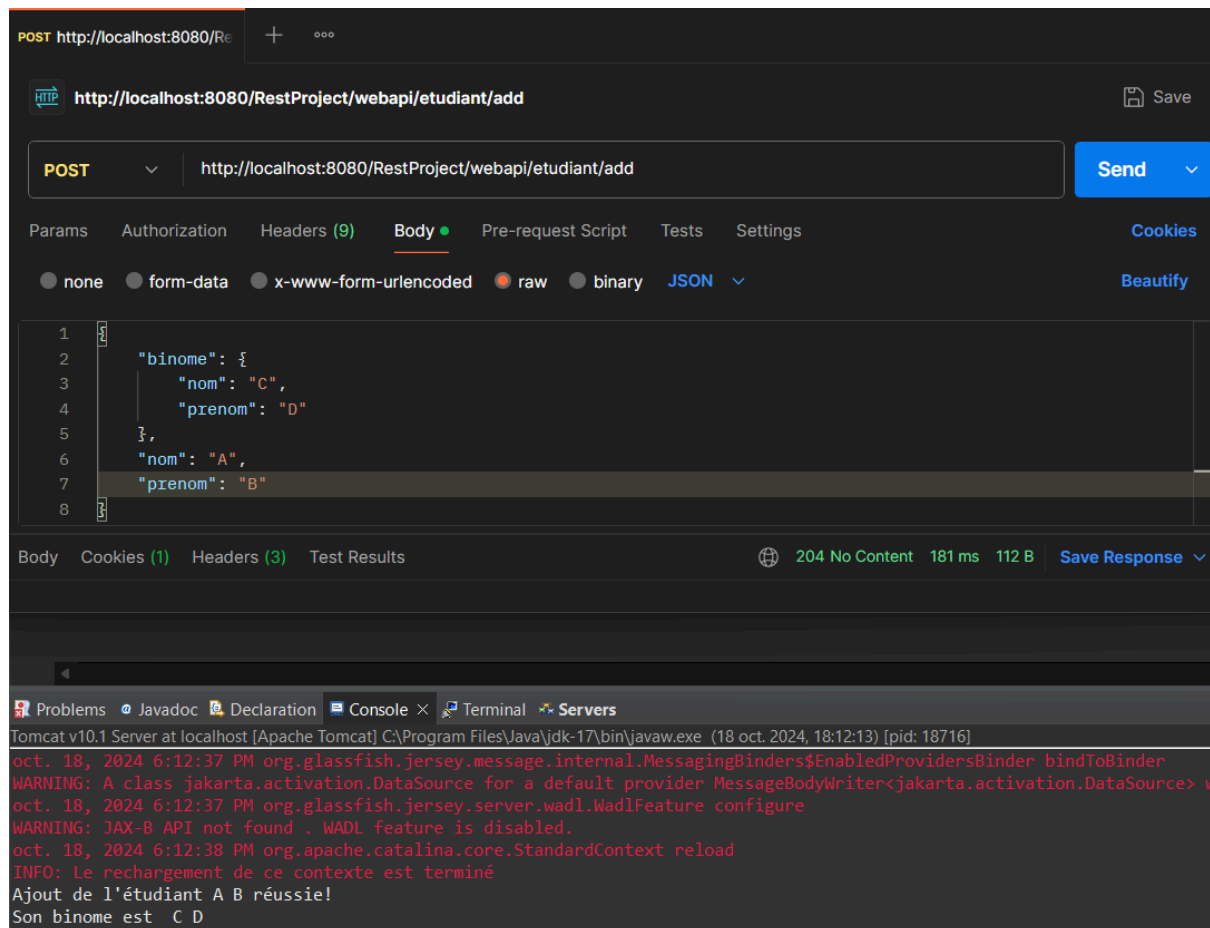


Figure 11: Posting JSON data to our service

Here on “Postman” we called the *add* method of our *student* class which allows us to add students. For this, we were able to add a new student and his partner using JSON.

2.3. Microservices

Earlier, we discussed why using a Service-Oriented Architecture is often better than relying on a Monolithic application. The main advantages include improved modularity of each service within the application and better interoperability.

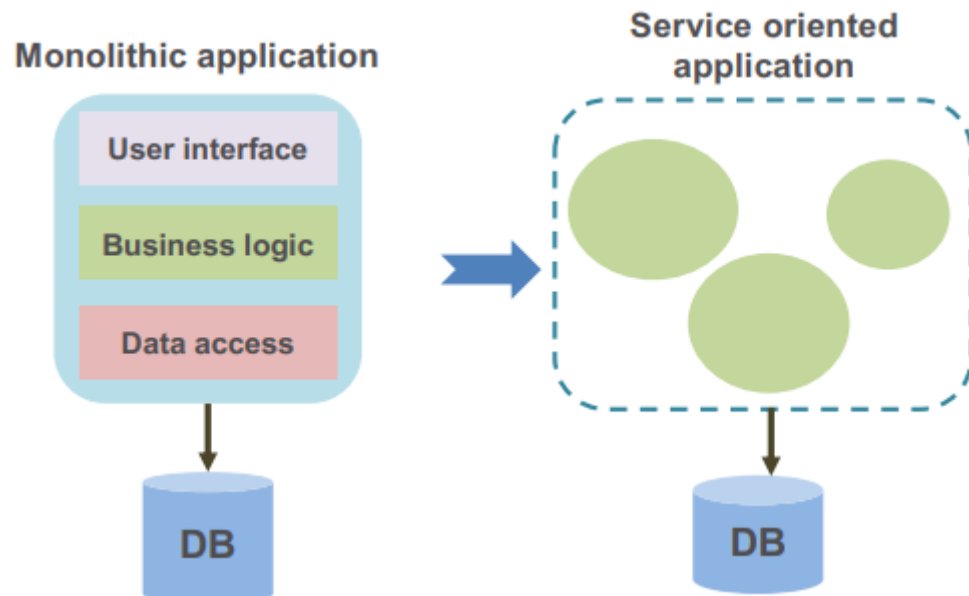


Figure 12: Difference between Monolithic and Service Oriented Application

However, SOA also has its limitations, particularly the complexity of SOAP, which uses envelopes for communication. This approach can also result in significant network overhead. An alternative solution is the Resource-Oriented Architecture, which is much simpler to implement and use. However, it lacks limits on service size and can make database access for individual services more challenging.

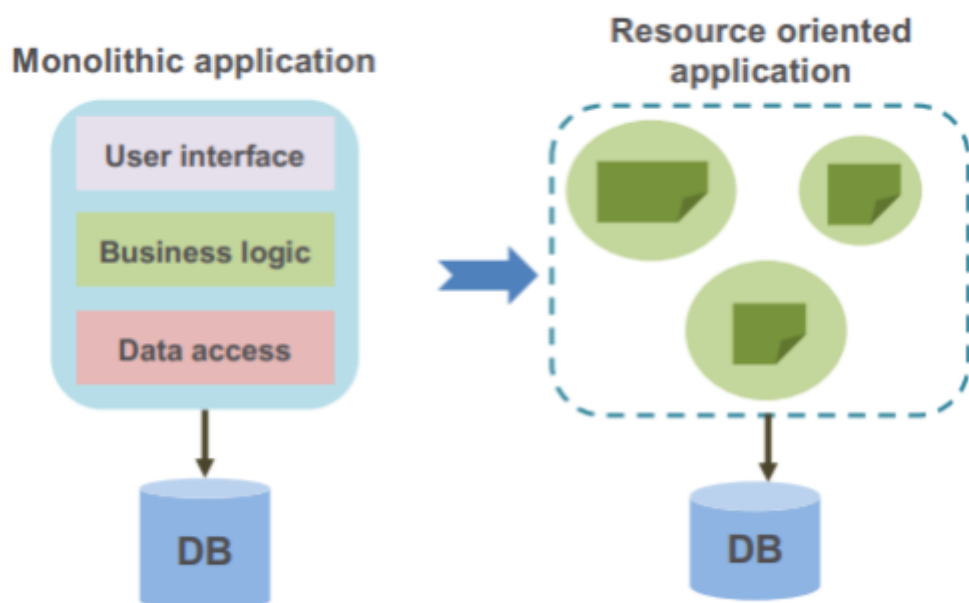


Figure 13: Difference between Monolithic and Resource Oriented Application

To address these drawbacks, a new approach based on REST architecture was introduced. This model divides the application into microservices, with each microservice dedicated to performing only one or a small number of specific tasks.

Each microservice has its own connection to the database and a dedicated port for addressing.

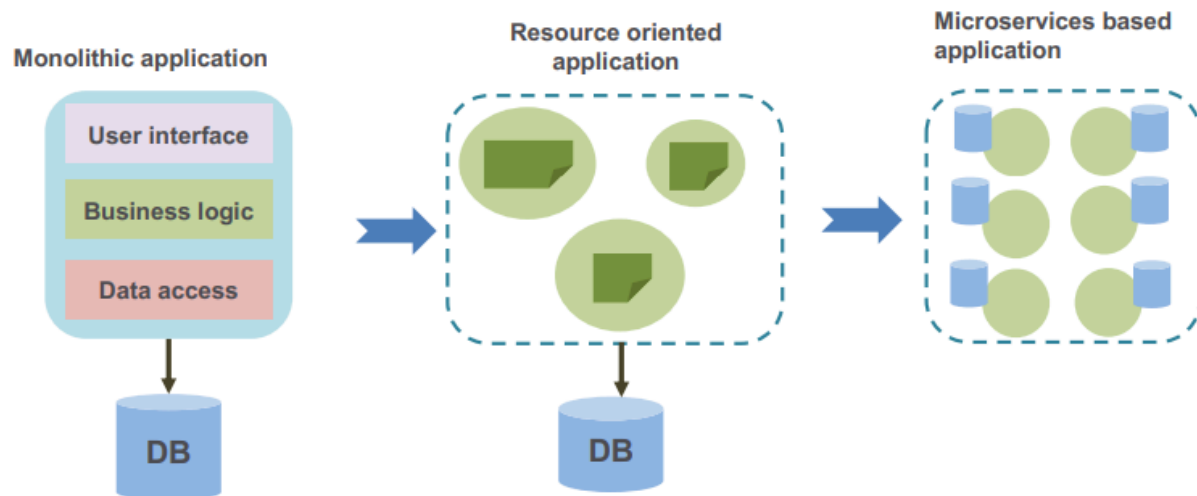


Figure 14: Difference between Monolithic and Resource Oriented and Microservices Application

To create a microservice, [Spring Initializr](#) can be used. This tool helps generate a microservice based on Spring Boot, which simplifies service setup and deployment.

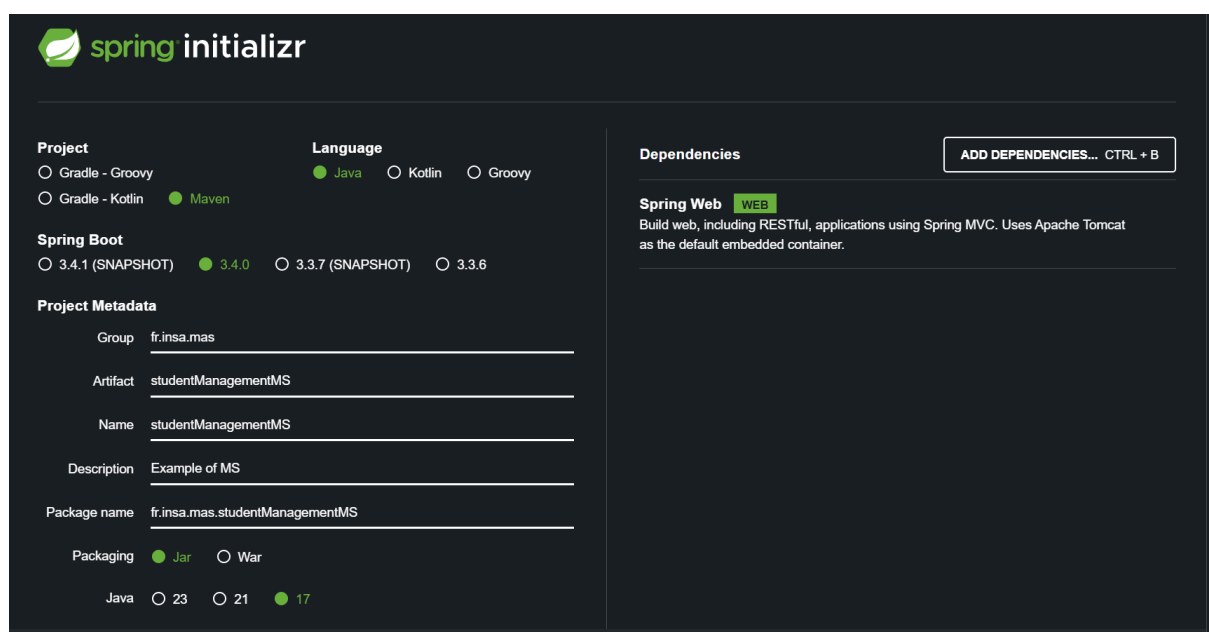


Figure 15: Spring initializr interface

Each microservice handles a specific function. For example:

- A "students" microservice might retrieve information about students.

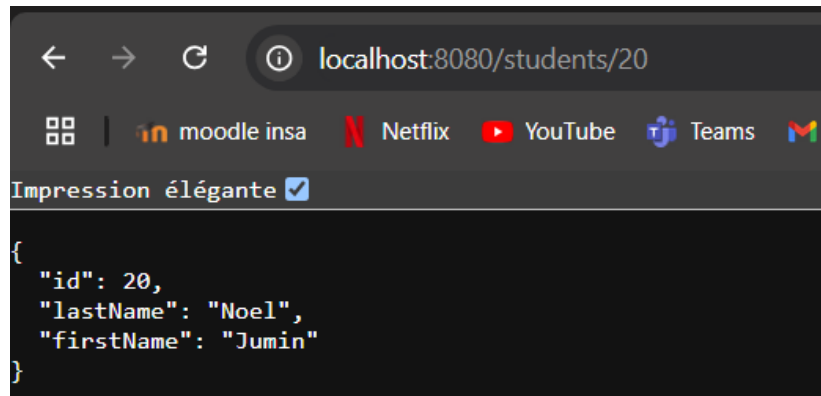


Figure 16: Student service response

- A separate "evaluation" microservice could provide student evaluation data.

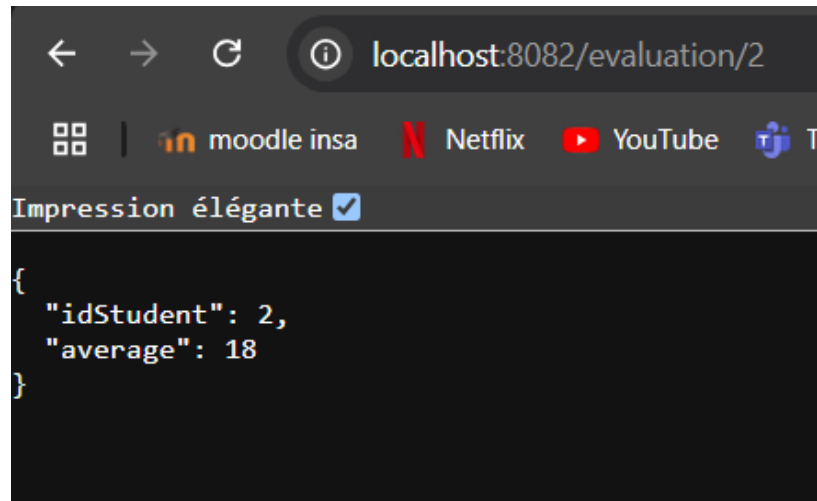
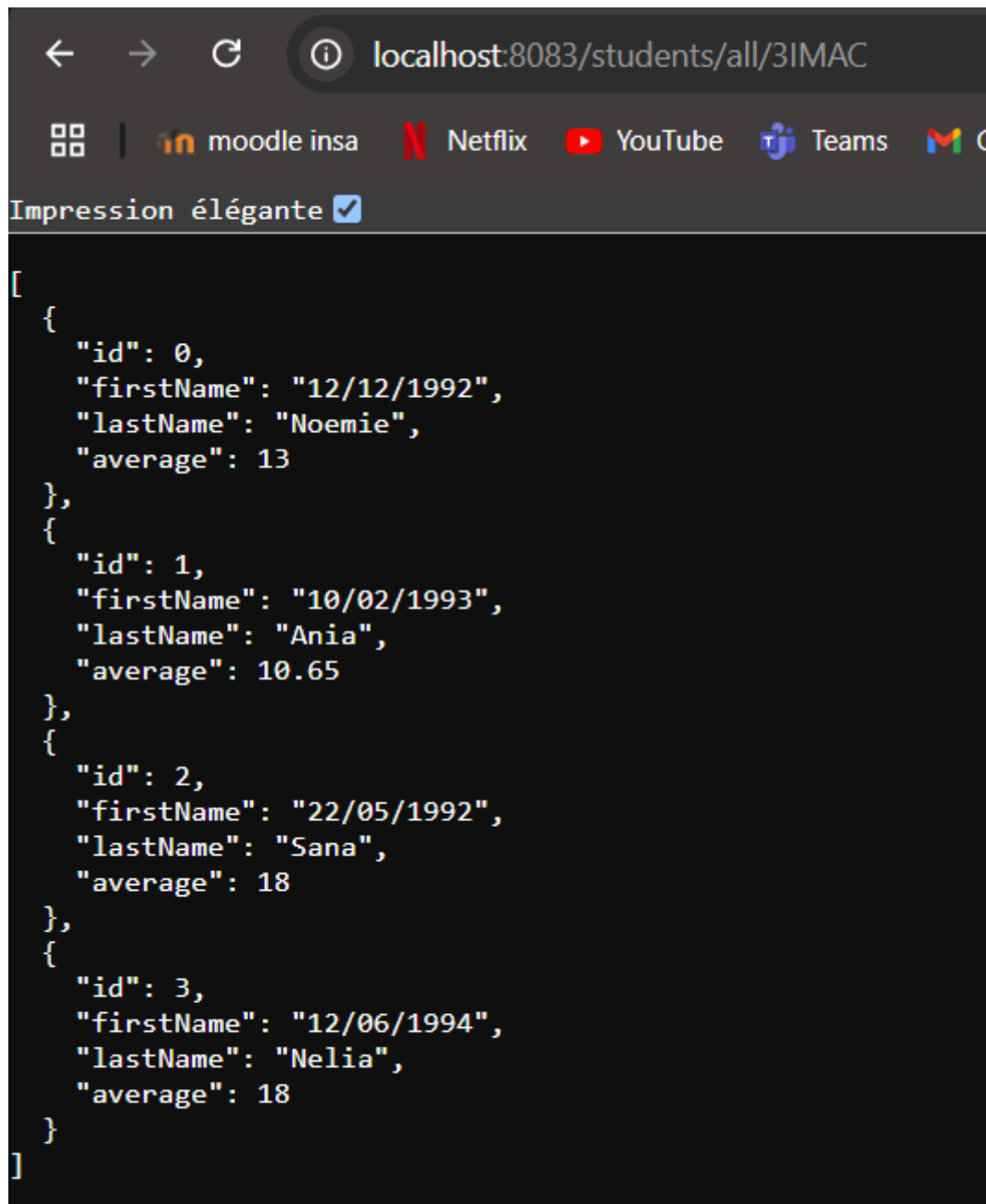


Figure 17: Evaluation service response

Finally, another microservice can be created to call these two services, establishing a connection that allows us to use the functions of both seamlessly.



```
[
  {
    "id": 0,
    "firstName": "12/12/1992",
    "lastName": "Noemie",
    "average": 13
  },
  {
    "id": 1,
    "firstName": "10/02/1993",
    "lastName": "Ania",
    "average": 10.65
  },
  {
    "id": 2,
    "firstName": "22/05/1992",
    "lastName": "Sana",
    "average": 18
  },
  {
    "id": 3,
    "firstName": "12/06/1994",
    "lastName": "Nelia",
    "average": 18
  }
]
```

Figure 18: Another micro service calling functions of the two above

As you can see, each microservice uses a different port for communication, allowing the functions of each microservice to be reused in different services.

3. The volunteer application

3.1. What is the purpose

The aim of this BE is to apply the concepts covered in the MOOC course. It takes the form of a volunteer application, a platform designed to connect people in need with volunteers who are willing to offer assistance. This mini-project would solve the problems of isolated individuals in society, such as those with dispersed families, disabling health problems, or long-term hospitalizations, by making it possible for them to request help for specific tasks, such as picking up packages or doing laundry.

The application will support a structured process for facilitating help. It will be possible for users to:

- Request Help: Users can add certain requests for assistance.
- Volunteer Support: The volunteer can answer the requests.
- Administrators: Admins moderate the requests that are published.

This application is designed based on different software architecture approaches like SOAP, REST, and microservices that make it scalable, reliable, and efficient.

3.2. Development decisions and deployment instructions

To demonstrate our ability to deploy the different types of service, we were asked to deploy a set of microservices communicating together.

We have chosen to develop these micro-services in REST due to two reasons:

- REST is way more simple to implement and use than SOAP
- Micro-services are based on REST

To run these services, we launch each service directly through a dedicated terminal, which means for the six implemented services, we use six terminal.

Another solution, had we had more time, would have been to launch a single Python file with a thread for each service, or to set up a Docker architecture with a Dockerfile for each service.

It is also essential to start the INSA VPN before launching the services.

3.3. Services

3.3.1. Database

In this application, we use a MySQL database which is accessed via the INSA VPN. We created an init.sql file that allows us to initialize our database by defining tables for each of the services.

```
mysql> SHOW TABLES;
+-----+
| Tables_in_projet_gei_037 |
+-----+
| administrators |
| feedbacks       |
| requests        |
| responses       |
| users           |
| volunteers      |
+-----+
6 rows in set (0.01 sec)
```

Each table has its own attributes, as shown below, where the users table contains four attributes: id, username, email, and password.

```
mysql> SELECT * FROM users;
+----+-----+-----+-----+
| id | username | email                | password |
+----+-----+-----+-----+
| 1  | user    | user@example.com     | password |
| 2  | user1   | user1@example.com    | password |
| 3  | User2   | user2@mail.com       | password |
| 4  | User3   | user3@mail.com       | password |
| 6  | User4   | user4@mail.com       | password |
| 10 | User10  | user10@mail.com      | password |
| 11 | User50  | user50@mail.com      | password |
| 17 | userjjj | user50@email.com     | password |
| 20 | user100 | user100@gmail.com    | test     |
| 21 | user1000 | test@gmail.com       | sss      |
| 22 | yoboujon | yo.boujon@gmail.com | noelleplusbeau |
+----+-----+-----+-----+
11 rows in set (0.01 sec)
```

Some tables are more complex, such as the feedbacks table, which depends on a user_id and a request_id. This makes it easy to retrieve the user and request associated with the feedback. Additionally, this setup allows all feedbacks and requests related to a user to be deleted if the user is removed.

```
mysql> SELECT * FROM feedbacks;
+----+-----+-----+-----+-----+
| id | user_id | response_id | text_of_feedback | validated |
+----+-----+-----+-----+-----+
| 9  | 1       | 8          | Non c'est très mal fait | 0 |
```



```
| 11 |      22 |      10 | oui |      1 |
+---+-----+-----+-----+-----+
2 rows in set (0.01 sec)
```

In the code, this is reflected by a key that references an attribute from another table and ensures cascading deletions when a related entry is removed.

```
-- Création de la table des réponses
CREATE TABLE IF NOT EXISTS feedbacks (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  user_id BIGINT NOT NULL,
  response_id BIGINT NOT NULL,
  text_of_feedback VARCHAR(500) NOT NULL,
  validated BOOLEAN NOT NULL,
  FOREIGN KEY (response_id) REFERENCES responses(id) ON DELETE CASCADE,
  FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

3.3.2. Requests

The Requests service is a fully independent micro-service that operates on port 8084 and implements five methods, which can be called through various HTTP requests.

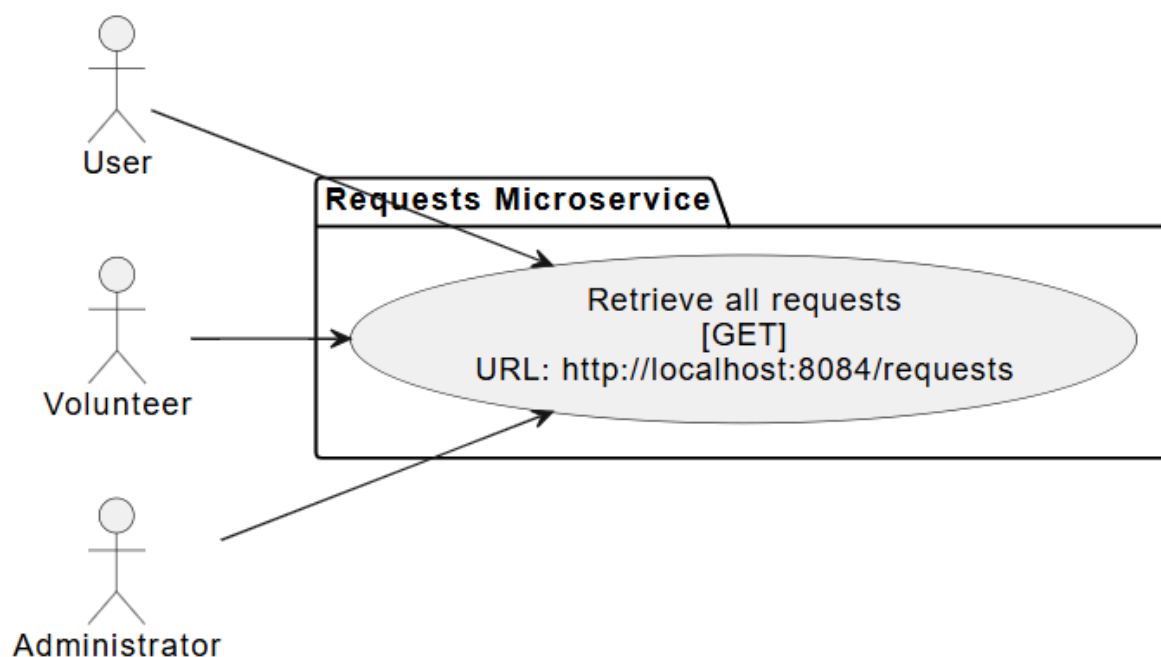


Figure 19: Get all requests method

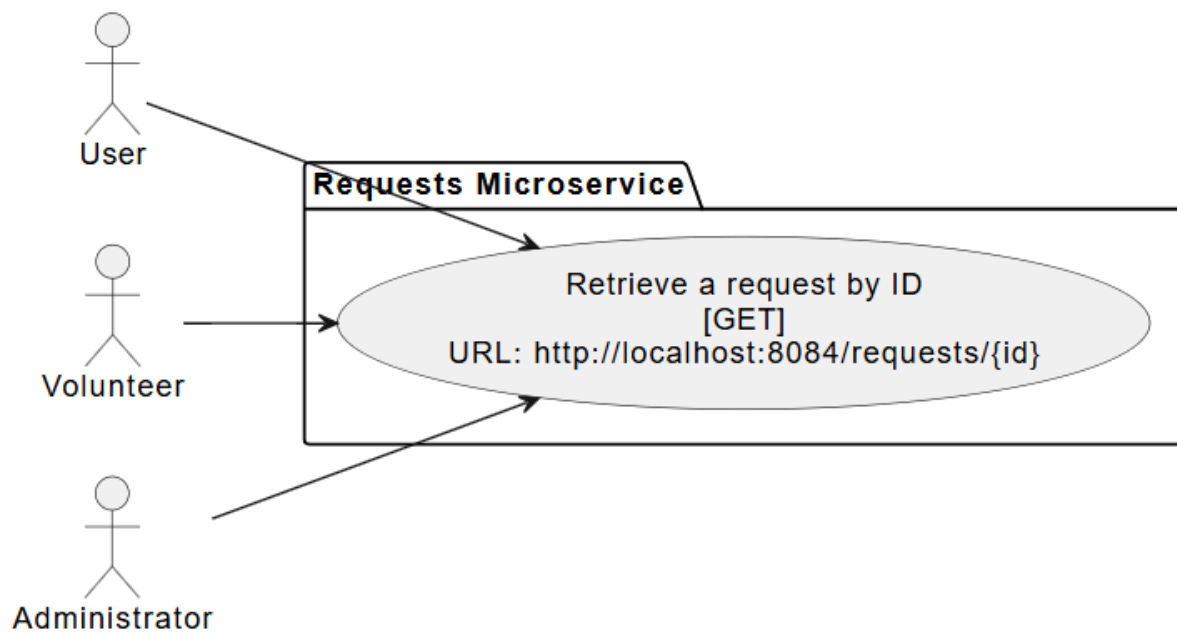


Figure 20: Get request by id method

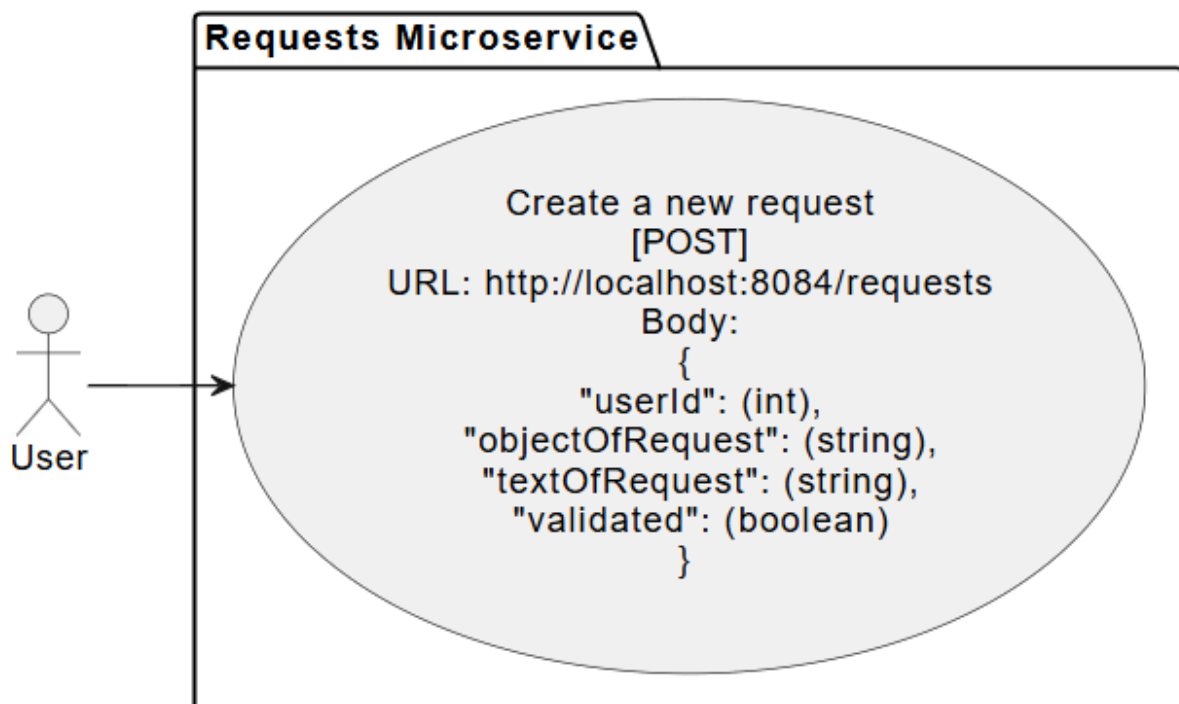


Figure 21: Post new request method

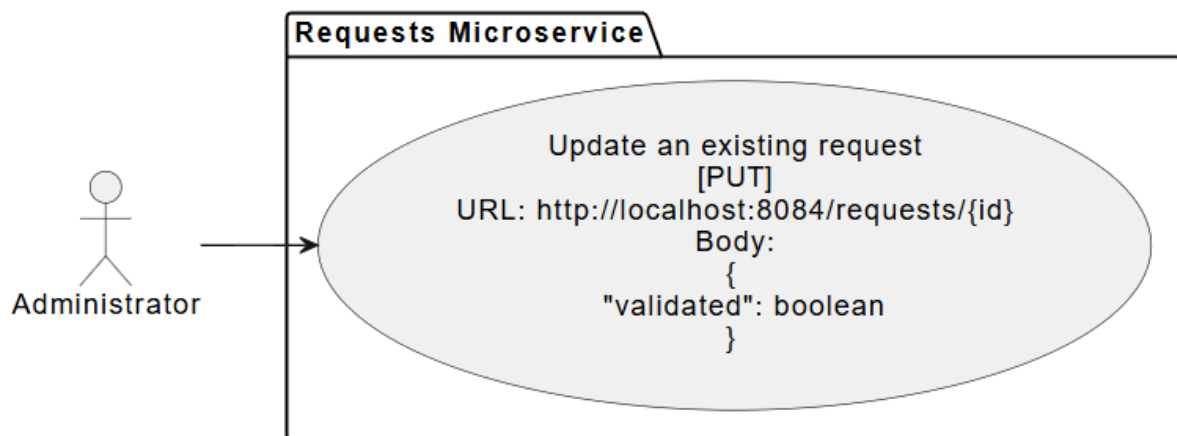


Figure 22: Put request method

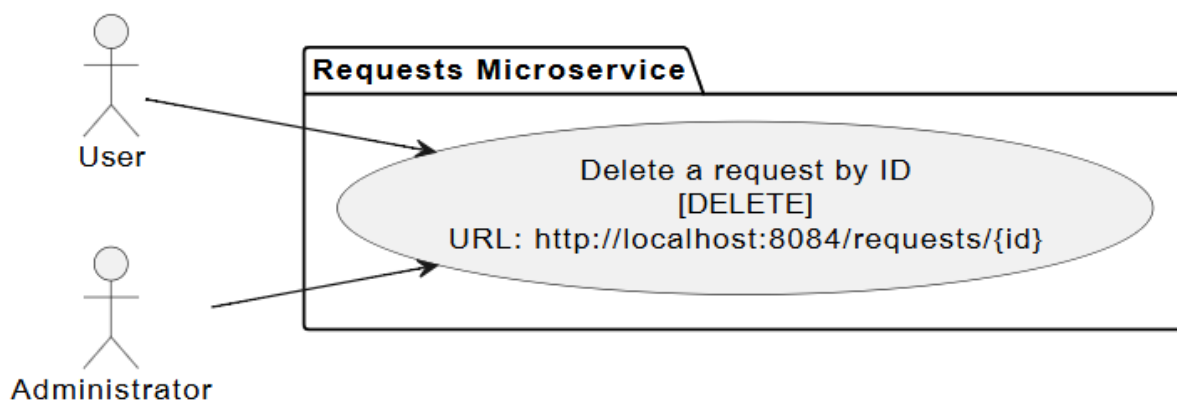


Figure 23: Delete request by id method

As you can see, the three different users have access to different methods of the Requests service. Each method communicates directly with the MySQL database.

3.3.3. Responses

The Responses micro-service is fully independent and operates on port 8085. It implements four methods, accessible through different HTTP requests.

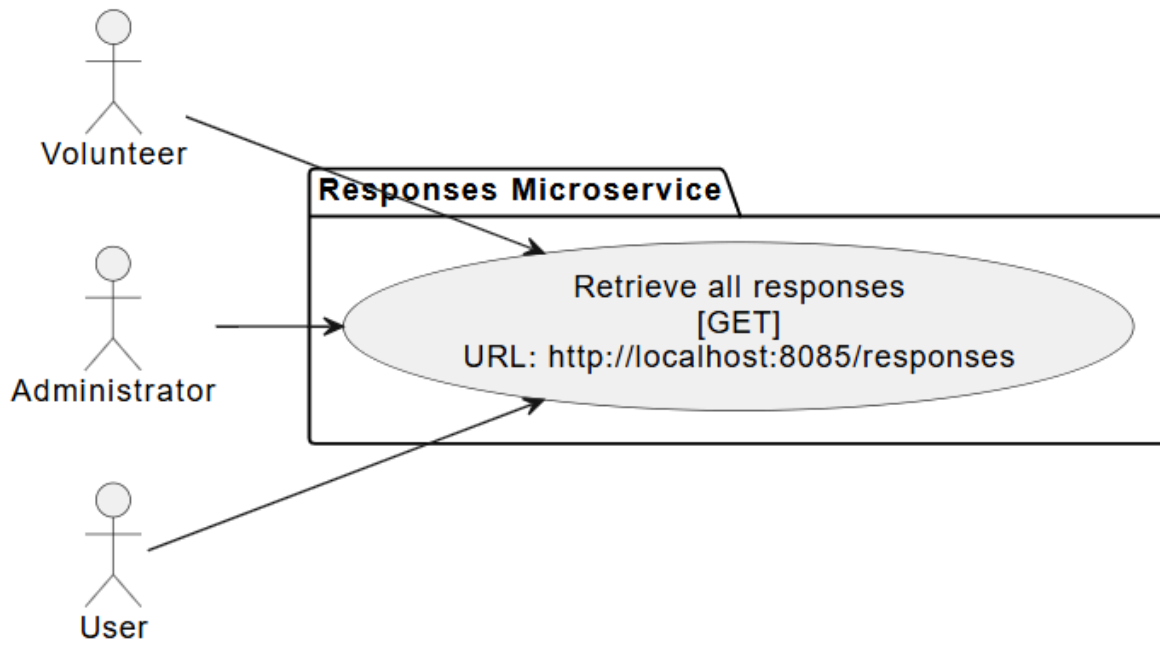


Figure 24: Get all responses method

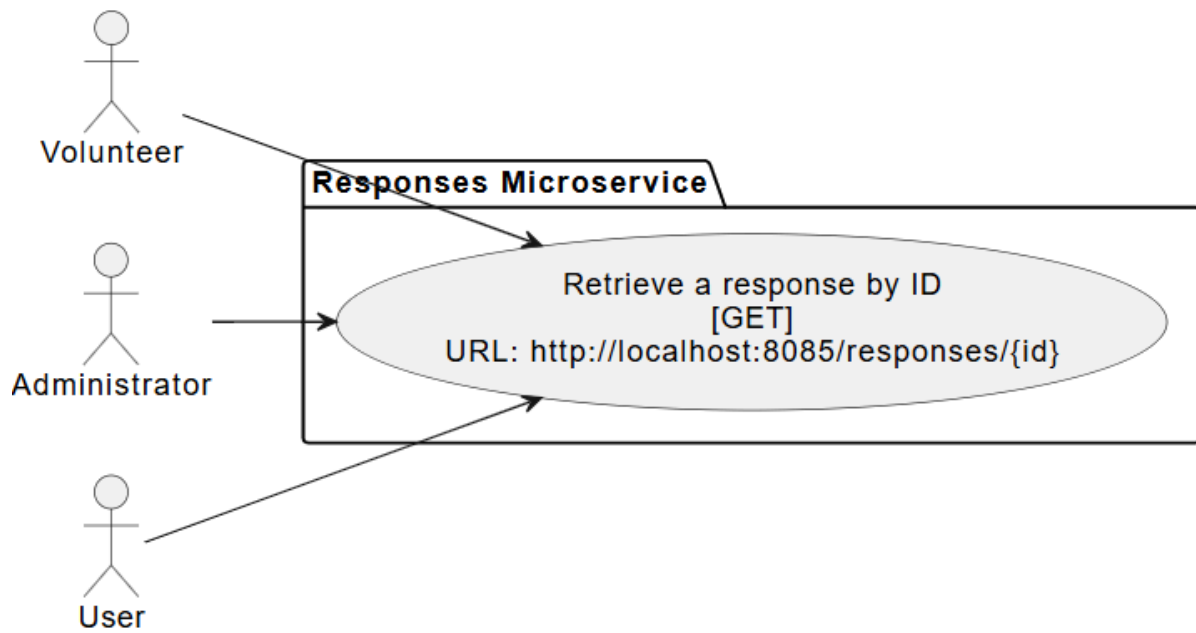


Figure 25: Get response by id method

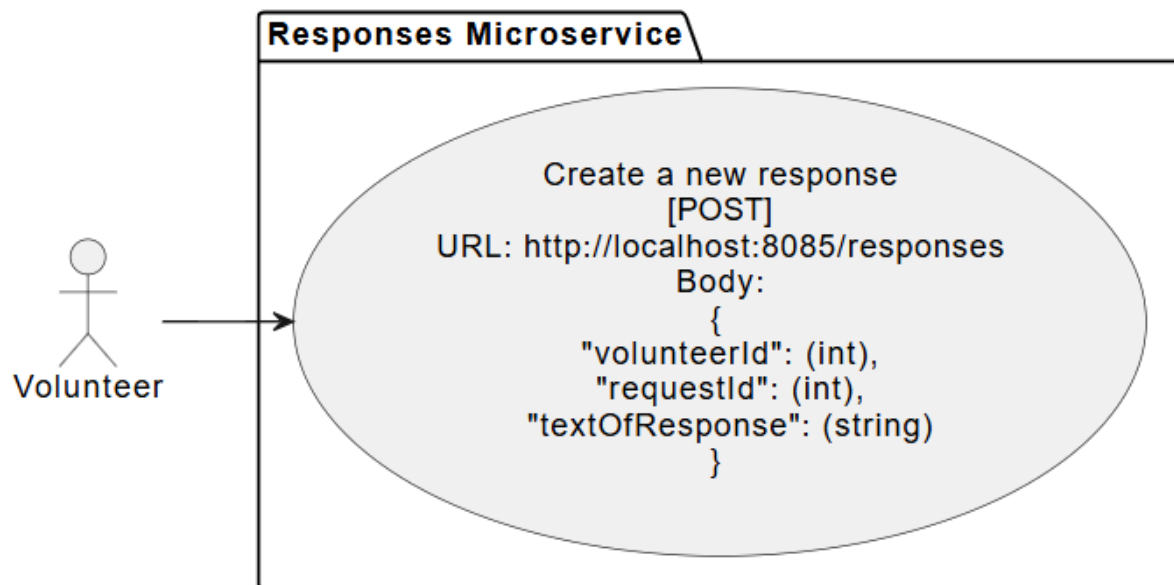


Figure 26: Post new response method

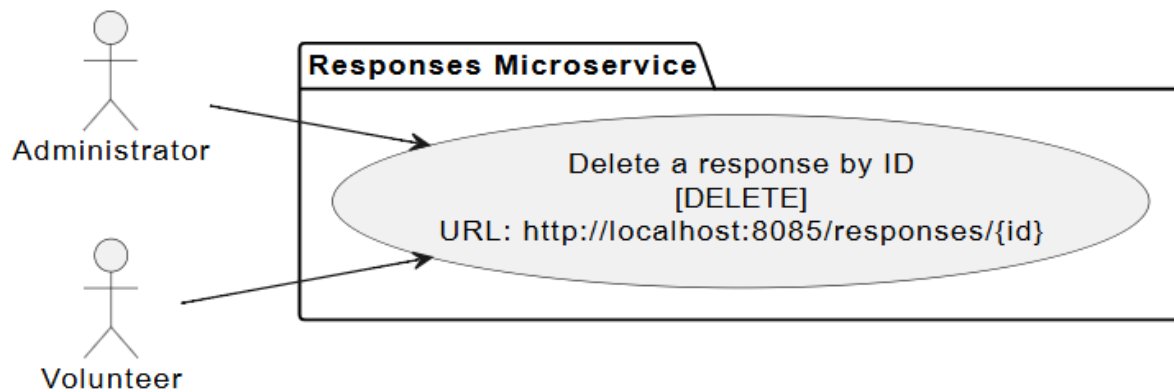


Figure 27: Delete response by id method

3.3.4. Feedbacks

The Feedbacks micro-service is fully independent and operates on port 8086. It provides four methods that can be called via different HTTP requests.

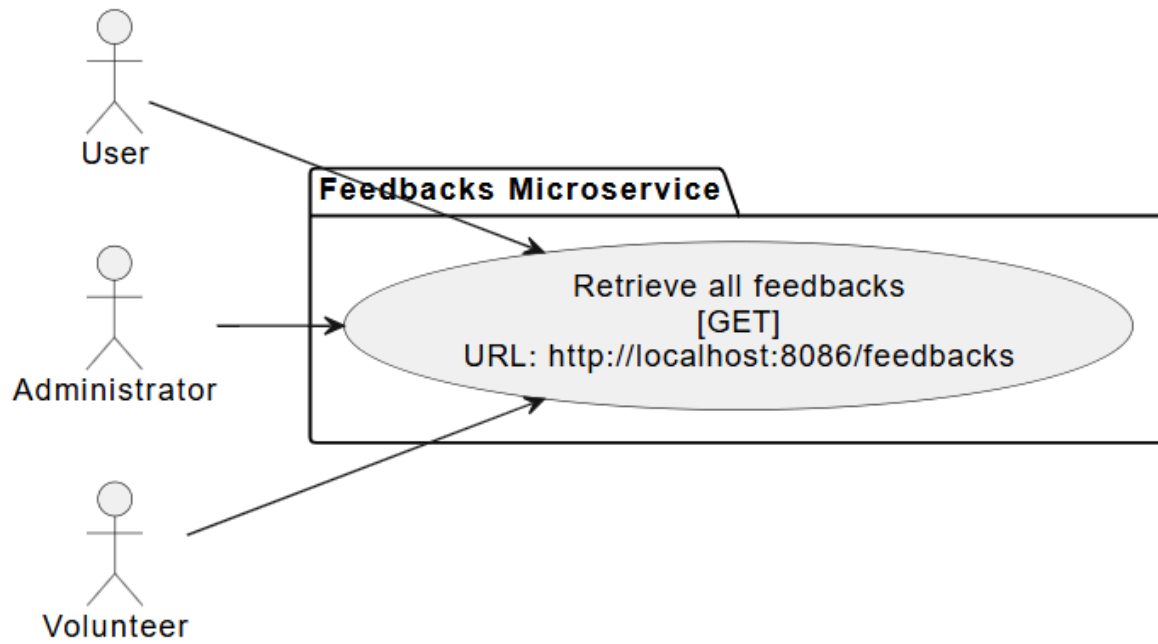


Figure 28: Get all feedbacks method

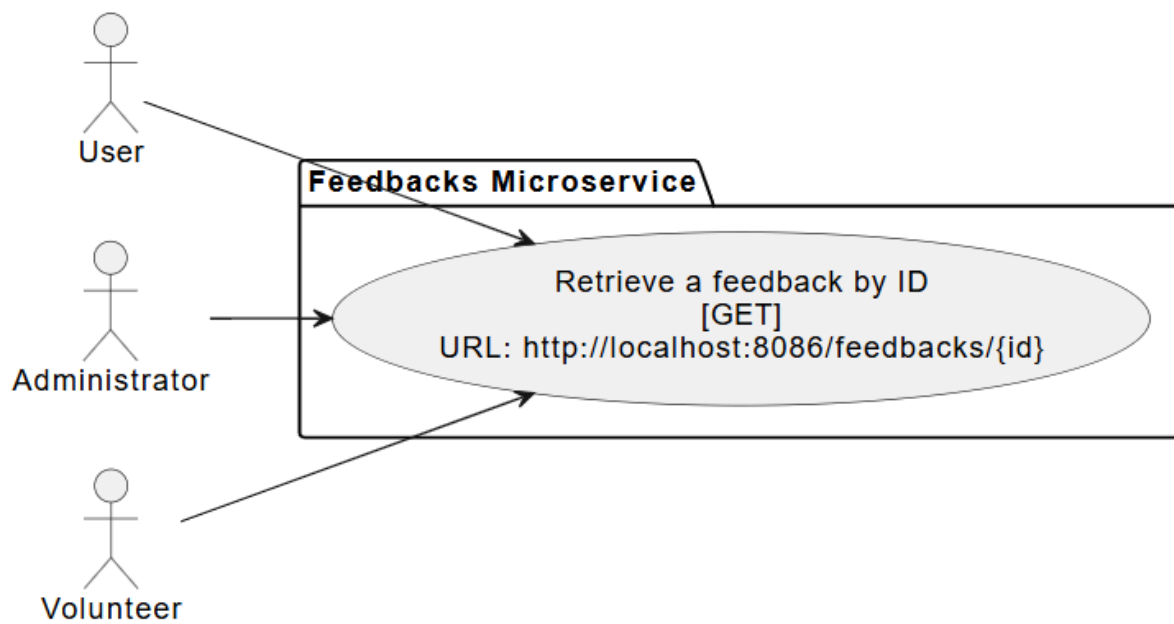


Figure 29: Get feedback by id method

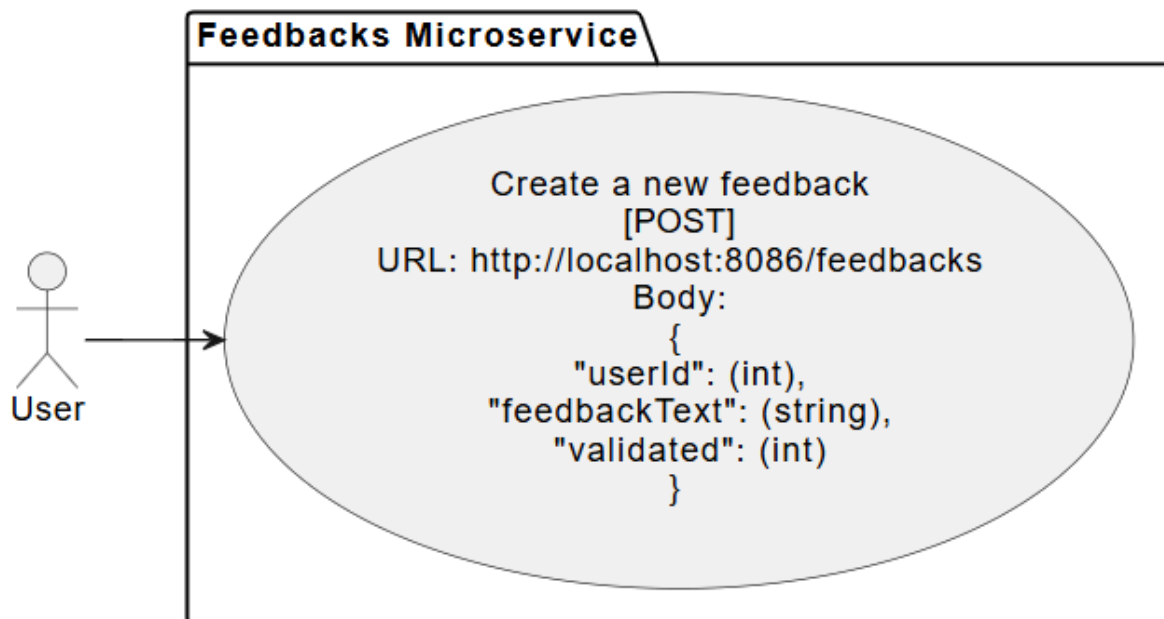


Figure 30: Post new feedback method

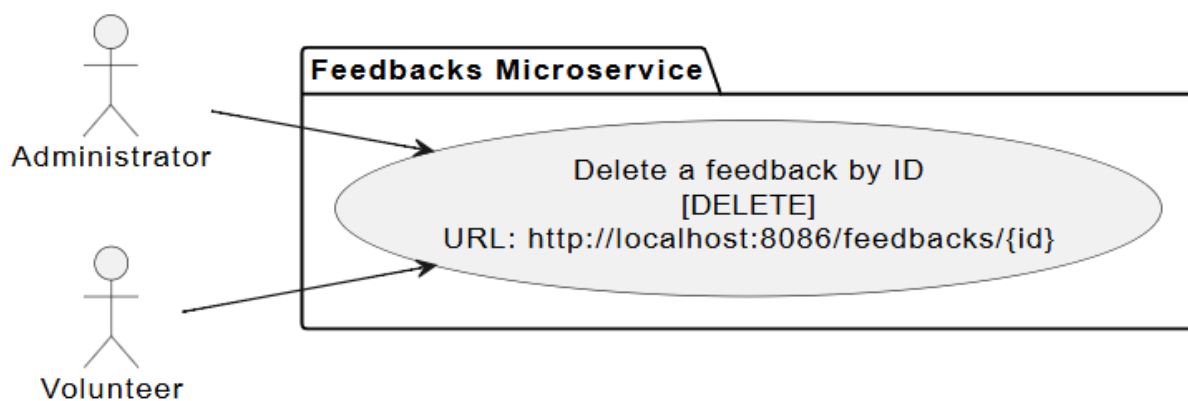


Figure 31: Delete feedback method

3.3.5. Administrators, Users and Volunteers

The three services shown in the diagram differ from the previous ones because they call the services above, as you can see in the diagrams. These are the services that will be accessed by the frontend, which we will discuss further below. Each of these services implements four methods, which are similar. The only difference is the call that is made. Let's take the Users service as an example.

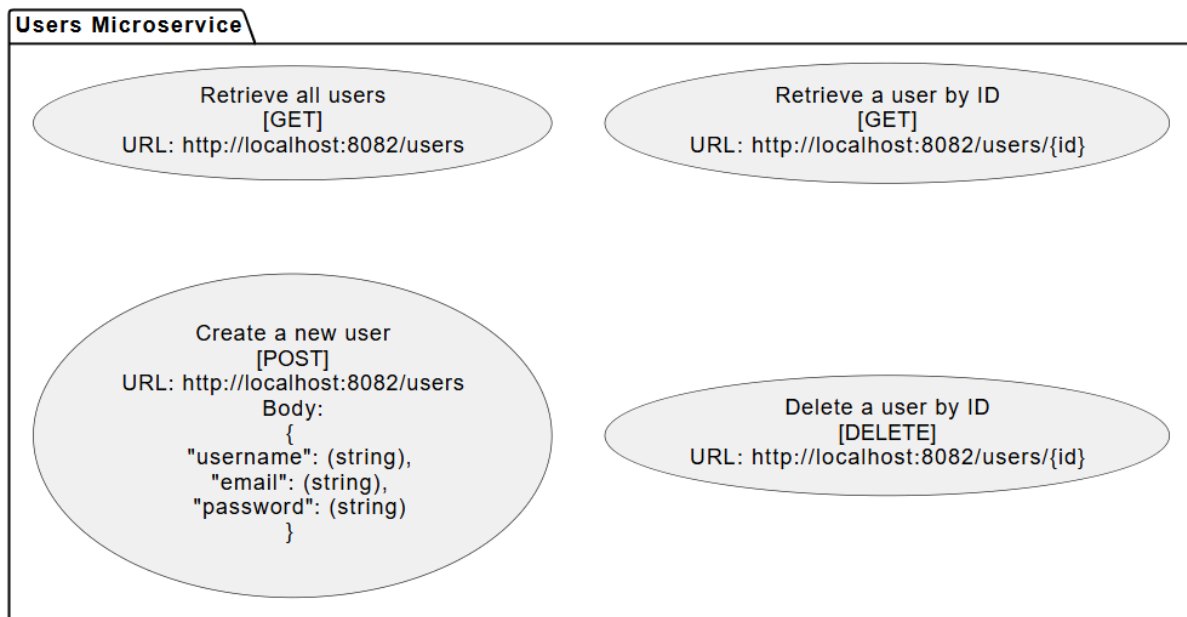


Figure 32: Different methods in Users

3.4. Our final application

Our final application integrates the various services mentioned above, including Users, Volunteers, Administrators, Requests, Responses, and Feedbacks. In addition to these services, we have a frontend, which serves as a web interface in HTML. It makes requests to the three main services: Users, Administrators, and Volunteers.

The frontend is built using HTML and CSS for the layout, featuring four pages: login, user, volunteer, and admin pages. Each page has its own JavaScript file, which handles HTTP requests to the services.

3.4.1. Login

On the Create Account page, there is a button for each user type (Users, Administrators, and Volunteers).

The screenshot displays a web interface for user authentication and account creation. At the top, there is a dark grey 'Login' header. Below it, three buttons labeled 'Administrator', 'User', and 'Volunteer' are arranged horizontally. The 'Create Account' section follows, featuring three input fields for 'User Username', 'User Email', and 'User Password'. A dark grey 'Register' button is positioned below these fields. At the bottom of the form, three buttons are provided: 'Create Admin Account', 'Create User Account', and 'Create Volunteer Account'.

Figure 33: User creation on login page

If the username or email is already in use, a rule in the database prevents the addition, as the UNIQUE constraint ensures these attributes must be unique in the table.

```
-- Création de la table des utilisateurs
CREATE TABLE IF NOT EXISTS users (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(255) UNIQUE NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE,
  password VARCHAR(255) NOT NULL
);
```

This will return an error to the calling service. Without any response from the service, the frontend will display the error.

Login

Administrator

User

Volunteer

Create Account

noel

noel@test.com

.....

Register

Registration failed: This username is already taken

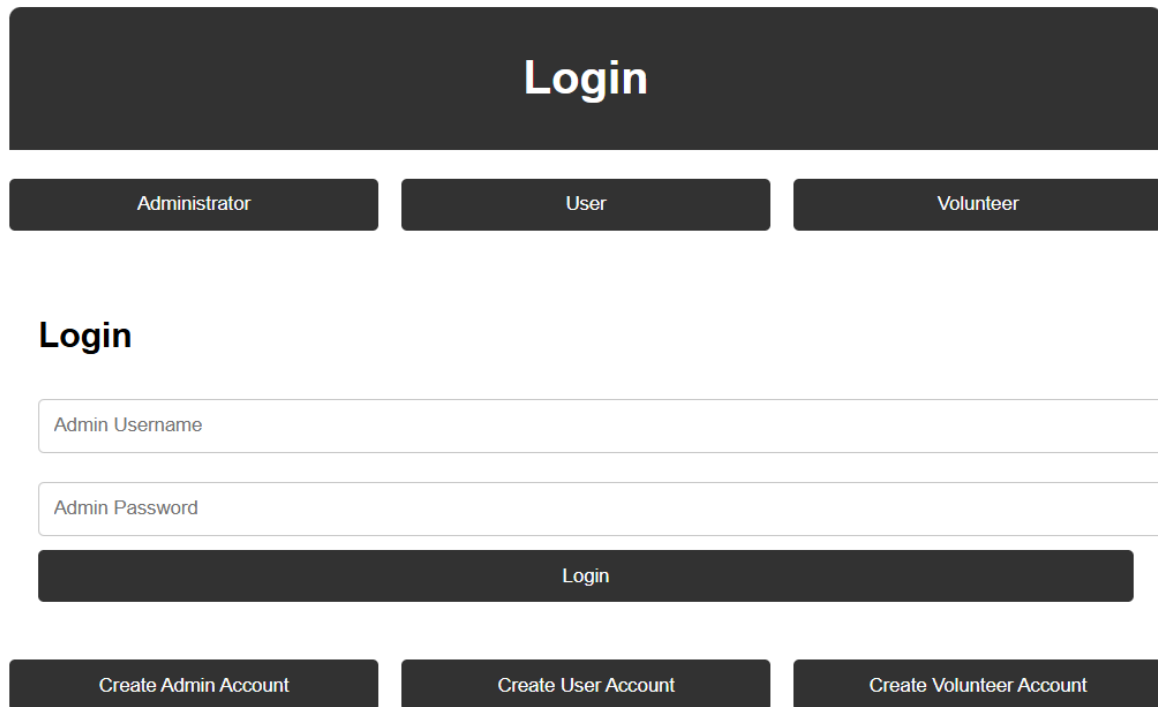
Create Admin Account

Create User Account

Create Volunteer Account

Figure 34: User creation error on login page

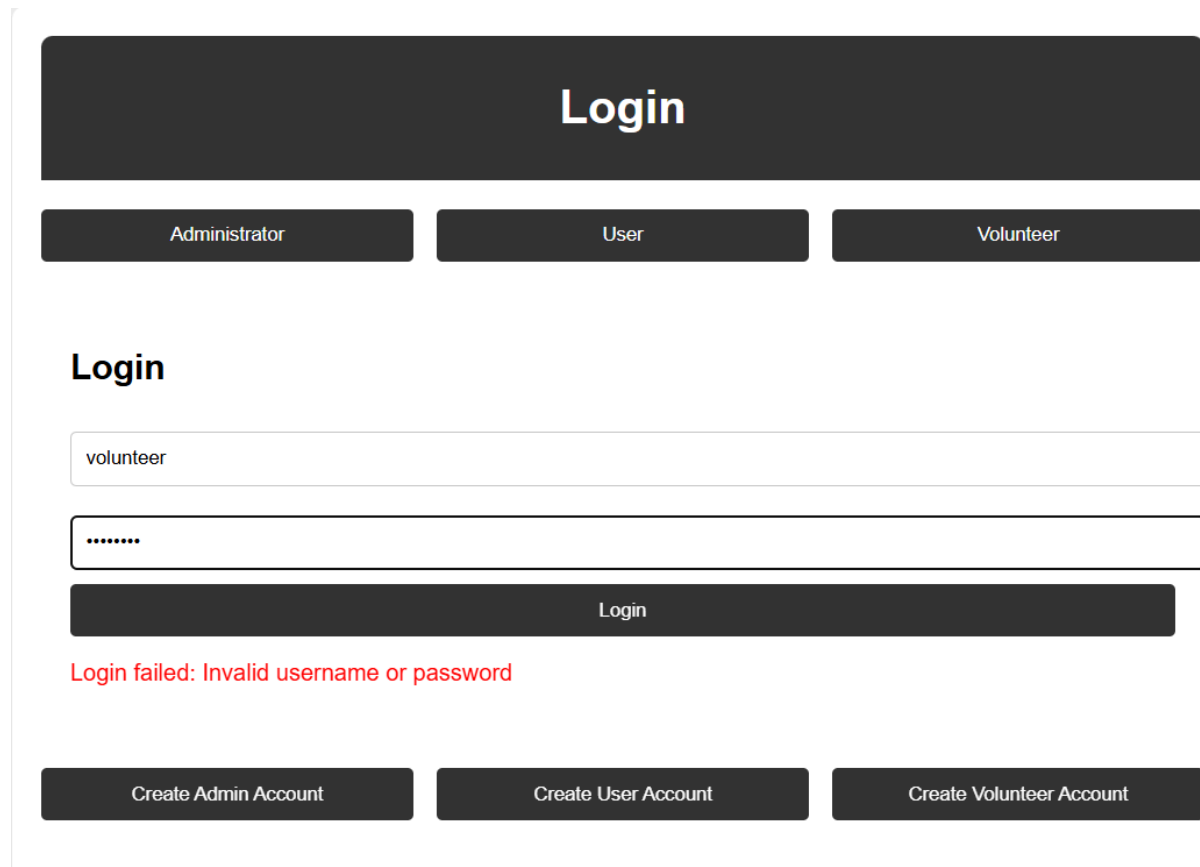
Additionally, login is possible by clicking one of the three top buttons.



The image shows a login interface for administrators. At the top, there is a large dark grey button labeled "Login". Below this, there are three smaller dark grey buttons labeled "Administrator", "User", and "Volunteer". The "Administrator" button is selected. Below these buttons, the word "Login" is displayed in a bold font. Underneath, there are two input fields: "Admin Username" and "Admin Password". Below the password field is a dark grey button labeled "Login". At the bottom, there are three dark grey buttons labeled "Create Admin Account", "Create User Account", and "Create Volunteer Account".

Figure 35: Administrator login on login page

If the entered username and password do not match any user, administrator, or volunteer, depending on the page, an error is returned.



The screenshot shows a web application's login interface. At the top, there is a dark grey header with the word "Login" in white. Below the header, there are three buttons: "Administrator", "User", and "Volunteer". The "Volunteer" button is selected. Below these buttons, there is a "Login" section with a text input field containing "volunteer", a password input field with masked characters "*****", and a "Login" button. Below the login button, a red error message reads "Login failed: Invalid username or password". At the bottom, there are three buttons: "Create Admin Account", "Create User Account", and "Create Volunteer Account".

Figure 36: Administrator login error on login page

If you enter the correct credentials, you will be redirected to one of the following three pages. In each of the JavaScript codes for the following pages, you will also find this small piece of code:

```
const loggedInUser = JSON.parse(localStorage.getItem('loggedInUser'));
if (!loggedInUser) {
  window.location.href = '../html/login.html';
  return;
}
const userId = loggedInUser.id;
const userName = loggedInUser.username;
```

This code retrieves the login information from the login page to determine the username of the logged-in user.

3.4.2. User

On the User login page, you can view all the requests that have been made.

user Dashboard

Logout

☒ All Requests ☐ My Requests ☐ Create Request

Request: Haircut

User: [user](#)

Text: I need a new haircut, some hairdresser available this afternoon

Delete

I can help you

Volunteer: [volunteer](#)

Add Feedback

I am the better for this

Volunteer: [volunteer2](#)

Add Feedback

Request: Plumber

User: [user1](#)

Text: I need a plumber for my bathroom.

Request: Driver

User: [User2](#)

Text: I need a driver to go to an appointment

I can help you if you need

Volunteer: [volunteer2](#)

Figure 37: All requests on user page

It is also possible to filter and view only the requests of the logged-in user.

user Dashboard

Logout

☐ All Requests ☒ My Requests ☐ Create Request

Request: Haircut

User: [user](#)

Text: I need a new haircut, some hairdresser available this afternoon

Delete

I can help you

Volunteer: [volunteer](#)

Add Feedback

I am the better for this

Volunteer: [volunteer2](#)

Add Feedback

Figure 38: My request on user page

Additionally, you can create a request visible to the administrator until it is validated.

User2 Dashboard

Logout

☐ All Requests ☐ My Requests ☒ Create Request

Create a New Request

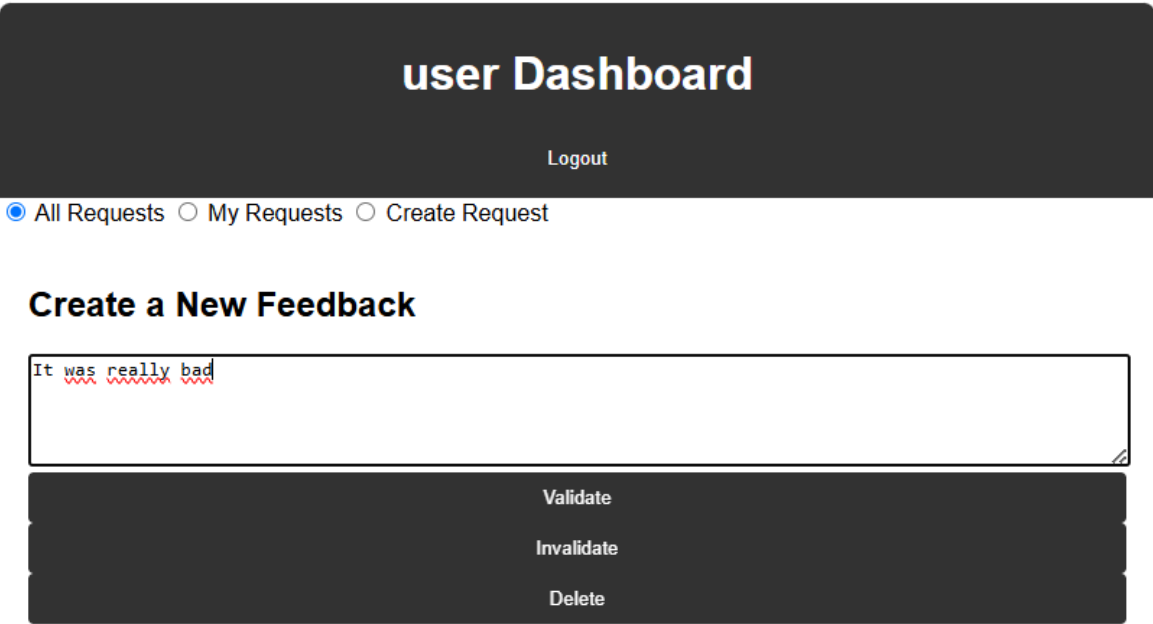
Driver

I need a driver to go to an appointment

Submit Request

Figure 39: Create a request on user page

For the responses to your requests, you can leave feedback and either approve or disapprove of the response.



The screenshot shows a dark-themed user dashboard. At the top, the text "user Dashboard" is centered in a large, bold, white font. Below it, a "Logout" link is centered in a smaller white font. Underneath the "Logout" link, there are three radio buttons: "All Requests" (which is selected), "My Requests", and "Create Request". Below these radio buttons, the heading "Create a New Feedback" is displayed in a bold white font. Under this heading is a text input field containing the text "It was really bad". Below the input field is a dark button labeled "Validate". Below the "Validate" button are two more dark buttons, labeled "Invalidate" and "Delete", stacked vertically.

Figure 40: Add a feedback on user page

Finally, you can see that you can only delete the requests and feedbacks from the user you're logged in as, as you do not have permission to delete other users' requests.

user Dashboard

Logout

☒ All Requests ☐ My Requests ☐ Create Request

Request: Haircut

User: [user](#)

Text: I need a new haircut, some hairdresser available this afternoon

Delete

I can help you

Volunteer: [volunteer](#)

It was perfect

Delete Feedback

I am the better for this

Volunteer: [volunteer2](#)

It was really bad

Delete Feedback

Request: Plumber

User: [user1](#)

Text: I need a plumber for my bathroom.

Request: Driver

User: [User2](#)

Text: I need a driver to go to an appointment

I can help you if you need

Volunteer: [volunteer2](#)

Figure 41: User page with requests, responses and feedbacks displayed

3.4.3. Administrator

The Administrator page grants the most privileges. It is possible to delete requests, responses, and feedback from any user, whether they are a user or a volunteer.

admin1 Dashboard

Logout

☒ Validated Requests ☐ Invalidated Requests

Request: Haircut
User: [user](#)
Text: I need a new haircut, some hairdresser available this afternoon

Delete Request

I can help you
Volunteer: [volunteer](#)

Delete Response

It was perfect

Delete Feedback

I am the better for this
Volunteer: [volunteer2](#)

Delete Response

It was really bad

Delete Feedback

Request: Plumber
User: [user1](#)
Text: I need a plumber for my bathroom.

Delete Request

Request: Driver
User: [User2](#)
Text: I need a driver to go to an appointment

Delete Request

I can help you if you need
Volunteer: [volunteer2](#)

Delete Response

Figure 42: All requests validated on Administrator page

You can also view unvalidated requests to either approve or delete them.

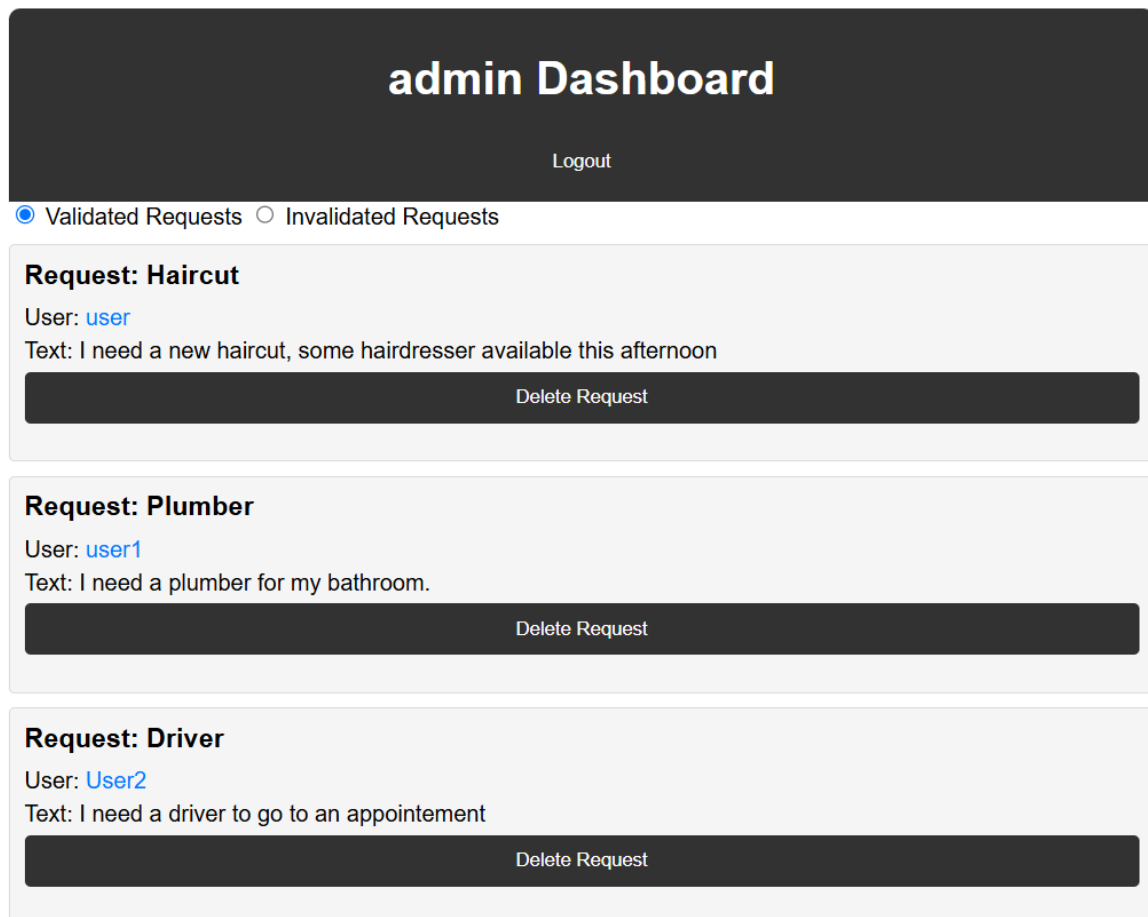
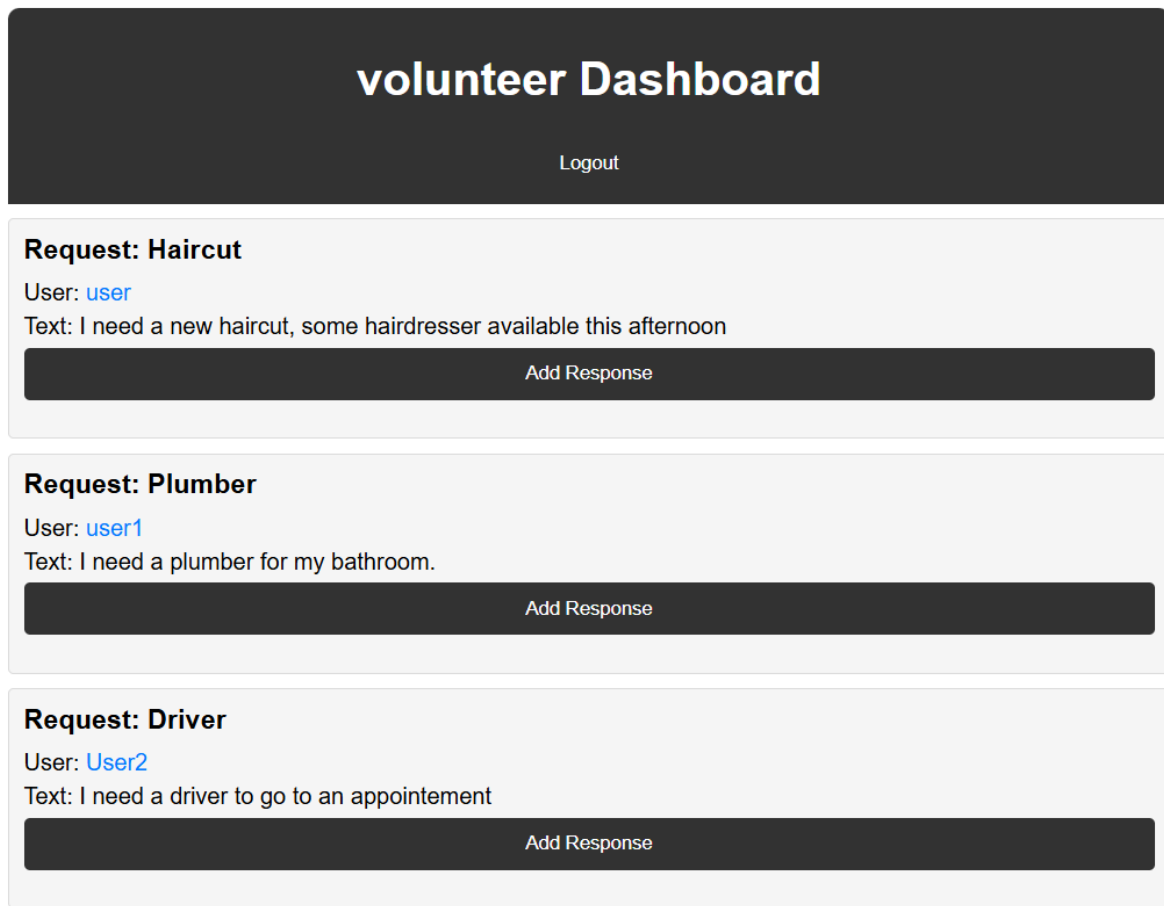


Figure 43: All requests not validated on Administrator page

3.4.4. Volunteer

The Volunteer page is the simplest. Like the other two pages, it allows you to view all validated requests.



volunteer Dashboard

Logout

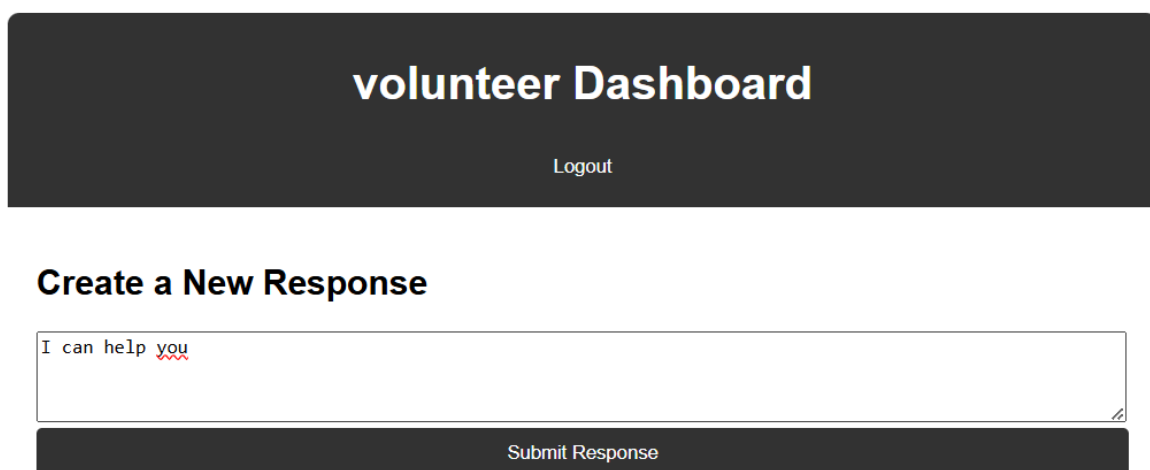
Request: Haircut
User: [user](#)
Text: I need a new haircut, some hairdresser available this afternoon
[Add Response](#)

Request: Plumber
User: [user1](#)
Text: I need a plumber for my bathroom.
[Add Response](#)

Request: Driver
User: [User2](#)
Text: I need a driver to go to an appointment
[Add Response](#)

Figure 44: All requests on volunteer page

It is also possible to leave a response to each request.



volunteer Dashboard

Logout

Create a New Response

I can help you

[Submit Response](#)

Figure 45: Create a response on volunteer page

You can see the responses from other volunteers and also view the feedbacks that you or others have left.

volunteer Dashboard

Logout

Request: Haircut

User: [user](#)

Text: I need a new haircut, some hairdresser available this afternoon

Add Response

I can help you

Volunteer: [volunteer](#)

Delete

It was perfect

I am the better for this

Volunteer: [volunteer2](#)

It was really bad

Request: Plumber

User: [user1](#)

Text: I need a plumber for my bathroom.

Add Response

Request: Driver

User: [User2](#)

Text: I need a driver to go to an appointment

Add Response

Figure 46: Volunteer page with requests, responses and feedbacks displayed

3.5. DevOps

We have developed a distributed software architecture based on the concept of services. Our next step is to implement a DevOps methodology. The goal is to demonstrate the implementation of Continuous Deployment (CD) alongside portions of Continuous Integration (CI) by automating the build and packaging process, which is a crucial component of DevOps (CI/CD).

Several Cloud solutions are available, such as Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft Azure. For this project, we have chosen Microsoft Azure as the deployment platform, specifically the free Education version. Our hands-on approach integrates GitHub for collaborative work and GitHub Actions for CI/CD automation, combined with Microsoft Azure for deployment.

To implement this, we created a resource group on Azure where we can deploy our services. Usually, one web application per service is created to properly test our application. Essentially, these deployments use virtual machines. Each GitHub repository typically includes one microservice, along with a configuration file to test and deploy the microservice to the Azure application on every push to the repository.

Here is a GitHub Actions workflow file example used for automating the deployment process:

```
# Docs for the Azure Web Apps Deploy action:
https://github.com/Azure/webapps-deploy
# More GitHub Actions for Azure: https://github.com/Azure/actions

name: Build and deploy JAR app to Azure Web App - User

on:
  push:
    branches:
      - main
  workflow_dispatch:

jobs:
  build:
    runs-on: windows-latest

    steps:
      - uses: actions/checkout@v4

      - name: Set up Java version
        uses: actions/setup-java@v4
        with:
          java-version: '17'
          distribution: 'microsoft'

      - name: Build with Maven
        run: mvn clean install -f Users/pom.xml

      - name: Upload artifact for deployment job
        uses: actions/upload-artifact@v4
```

```

with:
  name: java-app
  path: '${{ github.workspace }}/Users/target/*.jar'

deploy:
  runs-on: windows-latest
  needs: build
  environment:
    name: 'Production'
    url: '${{ steps.deploy-to-webapp.outputs.webapp-url }}'
  permissions:
    id-token: write #This is required for requesting the JWT

steps:
  - name: Download artifact from build job
    uses: actions/download-artifact@v4
    with:
      name: java-app

```

This workflow runs automatically on every push to the main branch.

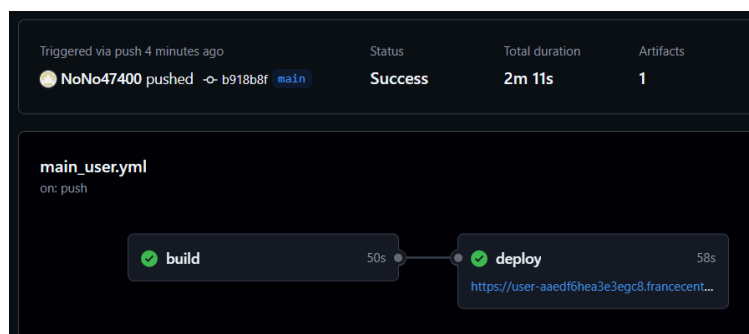


Figure 47: Github action interface

Once deployed, the services can be tested by accessing the Azure virtual machine (VM).

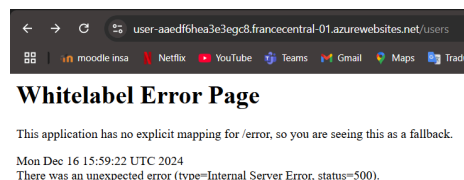


Figure 48: Azure application error

Currently, there is an issue: we do not have access to the database, so the deployed services will return errors due to this limitation. Expanding our implementation could

include incorporating database access and creating fully automated microservice testing within the CI/CD pipeline.

4. Conclusion

In conclusion, this project offered an invaluable opportunity to explore and apply modern architectural paradigms, such as SOA, REST, and microservices. By developing a volunteer platform, we highlighted the strengths and limitations of microservice type when addressing real-world problems. The created application not only emphasized modularity, scalability, and efficient service communication but also demonstrated the importance of aligning architectural decisions with specific project goals. As technology advances, mastering such architectures will remain critical for building adaptive, future-ready solutions.