INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

INNOVATIVE SMART SYSTEMS

Service Architecture

by JAULHIAC Paul
VASSEUR Cyril

16/12/2024
Year 2024/2025
5A Innovative Smart Systems

Introduction :

This document has been prepared as part of a practical workshop (TD) at INSA Toulouse, focusing on Service-Oriented Architecture (SOA), REST architecture, and microservices. Its objective is to provide an overview of the core concepts and practical techniques required to develop modern software systems based on these paradigms.

The workshop is structured around three main themes:

1. **Service-Oriented Architecture (SOA)**
   An introduction to web service development through a bottom-up approach, followed by the creation of clients to interact with these services using libraries like Apache CXF.
2. **REST Architecture**
   A deep dive into the design and development of RESTful services, covering the setup of environments like Tomcat, integration with JSON, and the implementation of advanced principles such as HATEOAS.
3. **Microservices Architecture**
   A practical study of breaking down applications into independent microservices, including database management, inter-service communication through gateways, and deployment on Azure servers.

Summary:

1/ Objectives of the course on Service Oriented Architecture

The objective of this course are to:

- Understand the Architecture of SOA (Service-Oriented Architecture)
- Understand Web Services
- Understand the Key Standards for Web Services:
  - WSDL (Web Services Description Language)
  - SOAP (Simple Object Access Protocol)
  - UDDI (Universal Description Discovery and Integration)
- Learn How to Develop, Describe, and Deploy Web Services
- Master the Use of SOAP Messages
- Grasp the Process of Service Discovery
- Understand the Concrete Description of Web Services

The goal is to provide a comprehensive understanding of how SOA, WSDL, and SOAP work together to facilitate the creation, discovery, and usage of web services in distributed environments.

To get to this point, we went through a series of technical tutorials, which allowed us to practice our skills as follows:

Bottom-Up Web Service (Tuto 1)

During the first tutorial we had to create a Web Service from a simple Java class and try to understand the WSDL (Web Services Description Language) generated from the service. And finally, we tested the Web Service using a browser.

1. Let's first answer a few questions:

- What is a Service-Oriented Architecture (SOA)?
  By definition, it is a software design approach where functionalities are encapsulated as reusable, self-contained services. These services communicate with each other over a network, often using standard protocols like SOAP or REST, enabling interoperability between different platforms, applications, and programming languages. The main goal of SOA is to allow some flexibility, scalability, and reusability, so systems are more maintainable and adaptable to change.

- What is a Web Service?

  A Web Service is a software application that is accessible over the Internet or a private network using SOAP or REST. It allows interoperable communication between different systems, platforms, and programming languages. A Web Service exposes functions or operations that clients (applications or users) can invoke remotely, typically via HTTP requests. The service's operations, input/output messages, and communication protocols are described using WSDL (Web Services Description Language). This enables dynamic discovery and interaction with the service without prior knowledge of its internal implementation.

- What are WSDL and SOAP?

  WSDL is an XML-based language used to describe the functionalities of a Web Service. It defines the operations offered by the service, the data types for inputs/outputs, the binding protocols (like HTTP or SOAP), and the service endpoint URL. WSDL allows clients to discover how to interact with a service without needing to know its internal implementation.

  SOAP is a messaging protocol that defines how data is structured and exchanged between a client and a Web Service. It uses XML to format messages and typically relies on HTTP or SMTP for transport. A SOAP message consists of an envelope (the root element), an optional header (for metadata like authentication), and a body (which contains the actual data or function call).

2. Practical work done:

We began by setting up a Maven project called AnalyseProject. After creating the project, we edited the pom.xml file to include the necessary dependencies, specifically Java 11 and the jaxws-rt (version 2.3.3) library, which enabled us to create SOAP Web Services.

```
10    <properties>
11            <java.version>17</java.version>
12    </properties>
13            <dependencies>
14                    <!-- https://mvnrepository.com/artifact/com.sun.xml.ws/jaxws-rt -->
15                    <dependency>
16                    <groupId>com.sun.xml.ws</groupId>
17                    <artifactId>jaxws-rt</artifactId>
18            <version>2.3.6</version>
19            </dependency>
20    </dependencies>
```

Next, we developed the Web Service logic by creating a Java class called AnalyserChaineWS in the package fr.insa.soap. The goal was to expose a method that receives a string as input and returns its length. To achieve this, we annotated the class with @WebService to declare it as a Web Service and gave it the name analyzer. We then exposed the method "analyser(String chaine)" as an operation called compare using @WebMethod, and we defined the input parameter using @WebParam(name="chain"). The method's logic was simple: it calculated and returned the length of the input string.

```
7     @WebService(serviceName="analyzer")
8     public class AnalyserChaineWS {
9             @WebMethod (operationName="compare")
10            public int analyser(@WebParam(name="chain") String chaine) {
11                    return chaine.length();
12            }
```

Fig. 2: AnalyserChaineWS java class

Once the logic was complete, we moved on to the deployment phase. We created a launcher class called AnalyserChaineApplication. In this class, we used the Endpoint.publish() method to deploy the service locally at http://localhost:8089/analyzer. Running the application allowed us to access the service and its WSDL file. This WSDL file describes the Web Service, detailing its available operations, input and output message types, and communication protocols.

```
public class AnalyserChaineApplication {
        public static String host="localhost";
        public static short port=8089;

        public void demarrerService() {
                String url="http://"+host+":"+port+"/";
                Endpoint.publish(url, new AnalyserChaineWS());
        }

        public static void main(String [] args )throws MalformedURLException{
                new AnalyserChaineApplication().demarrerService();
                System.out.println("service a démarré");
        }


}
```

Fig. 3: AnalyserChaineApplication launcher class

With the service deployed, we proceeded to the testing phase. We accessed the WSDL file in a web browser to review the description of the service and its operations. To test the service, we used the Web Service Explorer. We launched the explorer, provided the WSDL URL (http://localhost:8089/analyzer?wsdl), and executed the compare operation. We tested it with different string inputs, such as "Hello", and observed that the service correctly returned the string's length. We also reviewed the SOAP requests and responses exchanged with the service. This allowed us to see how SOAP messages are structured, including the envelope, header, and body.

In summary, we successfully developed a bottom-up Web Service starting from a simple Java class, exposed it as a SOAP Web Service, deployed it locally, and tested it using the Web Service Explorer. Through this process, we deepened our understanding of SOAP messaging, the role of WSDL in describing services, and how to use annotations like @WebService, @WebMethod, and @WebParam to expose methods as Web Service operations.

Web Service Client with CxF (Tuto2-Client-WS-cxf.pdf)

1. Objective:

This tutorial focuses on creating a **SOAP Web Service client** that consumes an existing Web Service. By the end, you should be able to:

- **Consume an existing Web Service** using its **WSDL** file.
- **Generate client code** from the WSDL using **Apache CXF**.
- **Invoke operations** on the Web Service from a Java program.

2. Key steps:

We started by creating a Maven project called ClientWS. We configured the pom.xml file to include the necessary dependencies, including Java 11, jaxws-rt (version 2.3.3), and the CXF Codegen Plugin (version 3.4.2). This plugin allowed us to generate Java classes directly from a WSDL file, making it easier to create a client to interact with the Web Service.

```
<?xml version='1.0' encoding='UTF-8'?><!-- Published by JAX-WS RI (https://github.com/eclipse-ee4j/metro-ja
<types>
<xsd:schema>
<xsd:import namespace="http://soap.insa.fr/" schemaLocation="http://localhost:8089/?xsd=1"/>
</xsd:schema>
</types>
<message name="compare">
<part name="parameters" element="tns:compare"/>
</message>
<message name="compareResponse">
<part name="parameters" element="tns:compareResponse"/>
</message>
<portType name="AnalyserChaineWS">
<operation name="compare">
<input wsam:Action="http://soap.insa.fr/AnalyserChaineWS/compareRequest" message="tns:compare"/>
<output wsam:Action="http://soap.insa.fr/AnalyserChaineWS/compareResponse" message="tns:compareResponse"/>
</operation>
</portType>
<binding name="AnalyserChaineWSPortBinding" type="tns:AnalyserChaineWS">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
<operation name="compare">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
</binding>
<service name="analyzer">
<port name="AnalyserChaineWSPort" binding="tns:AnalyserChaineWSPortBinding">
<soap:address location="http://localhost:8089/"/>
</port>
</service>
</definitions>
```

Fig. 4: analyser.wsdl file, copied and modified from Tuto 1

Once the project was ready, we retrieved the WSDL file for the Web Service we had created in Tuto 1. We copied the WSDL content into a file named analyser.wsdl and placed it in the directory src/main/resources/wsdl. To convert this WSDL file into usable Java classes, we configured the CXF Codegen Plugin in the pom.xml. The plugin's configuration specified that the WSDL file should be used to generate Java classes under target/generated-sources/cxf. After running the Maven generate-sources phase, we verified that the required Java classes and interfaces were created, matching the operations and types defined in the WSDL. For example, the portType in the WSDL was mapped to an interface named AnalyserChaineWS.

With the client-side classes ready, we proceeded to develop a client application to interact with the Web Service. We created a class called ClientOfAnalyzer in src/main/java. This class was responsible for connecting to the service, invoking its operations, and processing the responses. We instantiated the AnalyserChaineWSService class and used it to get a reference to the AnalyserChaineWS port, which allowed us to call the compare operation. We tested the method by sending a string input, such as "Hello", and confirmed that the service correctly returned the length of the string.

```
10  public class ClientOfAnalyzer {
11
12          public static void main(String[] args) throws MalformedURLException {
13                  // l'adresse du service web
14                  final String adresse="http://localhost:8089/analyser";
15                  // création URL
16                  final URL url = URI.create(adresse).toURL();
17                  // instanciation de l'image de service
18                  final Analyzer service = new Analyzer(url);
19                  // création du proxi
20                  final AnalyserChaineWS port = service.getPort(AnalyserChaineWS.class);
21                  String chaine="aaaaaaa";
22                  // appel de méthode compare via le port
23                  System.out.println("la taille de la chaine "+chaine+" est "+port.compare(chaine));
24
25          }
26
27  }
```

Fig. 4: ClientOfAnalyzer class

To ensure everything was working correctly, we ran the ClientOfAnalyzer application. The application established a connection to the Web Service, sent a SOAP request containing the input string, and displayed the response. We observed how the SOAP request and response were exchanged, paying attention to the message structure (envelope, header, body) and the data being transmitted.

In summary, we successfully developed a SOAP client that consumes a Web Service using CXF and the WSDL-to-Java approach. We also observed how SOAP requests and responses were exchanged.

2/ Objectives of the course on REST Architecture

The practical works for the REST course guided us through the development of **RESTful Web services**, from server setup to creating fully **HATEOAS-compliant APIs**. Below is a summary of each tutorial, including the main objectives, key concepts, and practical tasks we completed.

1. Setup of Tomcat (Setup-Tomcat.pdf)

To begin, we focused on setting up the environment required to deploy and test our RESTful web services. We installed Apache Tomcat 9, a web server that serves as the backbone for our web services. After downloading and extracting the binaries, we configured Eclipse IDE to recognize Tomcat as a server. This setup allowed us to easily start, stop, and restart the server from the Eclipse interface. Finally, we validated our setup by starting the server and checking for successful log messages in the console. With this preparation complete, we had a stable development environment ready to support the next stages of our work.

2. Introduction to REST (Tuto1-Rest-Intro.pdf)

This first tutorial introduced us to the fundamental principles of RESTful Web services and guided us through the process of creating and deploying a simple RESTful service.

We started by creating a Maven project using Jersey, a framework for developing RESTful web services in Java. We edited the pom.xml file to add essential dependencies, including Jersey libraries and the Maven War plugin. The web.xml file was updated to specify the path where our service would be accessible, typically /webapi/.

```
28        <dependencyManagement>
29            <dependencies>
30                <dependency>
31                    <groupId>org.glassfish.jersey</groupId>
32                    <artifactId>jersey-bom</artifactId>
33                    <version>${jersey.version}</version>
34                    <type>pom</type>
35                    <scope>import</scope>
36                </dependency>
37            </dependencies>
38        </dependencyManagement>
39
40        <dependencies>
41            <dependency>
42                <groupId>org.glassfish.jersey.containers</groupId>
43                <artifactId>jersey-container-servlet-core</artifactId>
44                <!-- use the following artifactId if you don't need servlet 2.x compatibility -->
45                <!-- artifactId>jersey-container-servlet</artifactId -->
46            </dependency>
47            <dependency>
48                <groupId>org.glassfish.jersey.inject</groupId>
49                <artifactId>jersey-hk2</artifactId>
50            </dependency>
51            <!-- uncomment this to get JSON support
52            <dependency>
53                <groupId>org.glassfish.jersey.media</groupId>
54                <artifactId>jersey-media-json-binding</artifactId>
55            </dependency>
56            -->
57        </dependencies>
```

Fig. 5: pom.xml/dependencies

Our first service involved creating a Comparator class annotated with @Path("comparator"). Inside this class, we implemented simple REST methods annotated with @GET, @POST, and @PUT, exposing them as API endpoints. Each method was linked to a specific HTTP request type, and we tested the service using Postman. For example, we accessed the service using a GET request to the URL http://localhost:8080/RestProject/webapi/myresource, and it returned a simple text response.

We extended the service by adding logic to compute the length of a string provided in the URL, with an endpoint like /webapi/longueur/Toulouse, where "Toulouse" was treated as a parameter. This exercise introduced us to the concept of URI-based parameters. We also learned to handle PUT requests using Postman, since browsers can only send GET requests by default. By the end of this tutorial, we had created, deployed, and tested a working RESTful service.

```
12  @Path("comparator")
13  public class Comparator {
14          @GET
15          @Produces(MediaType.TEXT_PLAIN)
16          public String getLongueur() {
17                  return "Bonjour!";
18          }
19          @GET
20          @Path("longueur/{chaine}")
21          @Produces(MediaType.TEXT_PLAIN)
22          public int getLongueur(@PathParam("chaine")String chaine) {
23                  return chaine.length();
24          }
25          @GET
26          @Path("longueurDouble")
27          @Produces(MediaType.TEXT_PLAIN)
28          public int getLongueurDouble(@QueryParam("chaine")String chaine) {
29                  return chaine.length()*2;
30          }
31          @PUT
32          @Path("/{idEtudiant}")
33          @Consumes(MediaType.TEXT_PLAIN)
34          public int updateEtudiant(@PathParam("idEtudiant")int id) {
35                  System.out.println("mise à jour réussie!!!");
36                  return id;
37          }
```

Fig. 6: Content of Comparator class

3. Building a REST Client (Tuto2-Client-Rest.pdf)

After successfully creating a RESTful service, we moved on to building a REST client. The objective of this tutorial was to teach us how to consume a REST API from a client-side application. Our task was to develop a client capable of sending requests to the REST API and receiving responses.

To achieve this, we wrote a Java program to send GET, POST, PUT, and DELETE requests to the service. The Jersey client allowed us to programmatically build HTTP requests and capture the server's response. We verified the client's ability to interact with the service by checking the server logs for incoming requests. This helped us understand the communication between REST clients and REST servers and how to manage request/response cycles.

```java
public class ClientRest {

        public static void main(String[] args) {
                //instanciation de client, necessaire pour l'exécution des requétes et la consomation des réponses.
                Client client=ClientBuilder.newClient();
                //appel du service Rest, invocation de la méthode get correspondant à l'url
                Response response=client.target("http://localhost:8080/RestProject/webapi/comparator/longueur/Toulouse").request().get();
                //lecture de la réponse récupérée
                System.out.println(response.readEntity(String.class));
        }

}
```

Fig. 7: Client Java program

4. Working with JSON (Tuto3-Rest-JSON.pdf)

(Cf. RestProject/src/main/java/fr/insa/ws/soap/RestProject/ , in GitHub)

This tutorial expanded our understanding of data serialization in REST. While our previous services only returned text responses, we now learned to send and receive JSON data. JSON, being more lightweight and human-readable, is the standard format used in RESTful web services.

We updated our RestProject from Tutorial 1 by creating two new Java classes: Etudiant and Binome. These classes model the structure of the data we intended to send and receive. To ensure our service returned JSON responses, we used the @Produces(MediaType.APPLICATION_JSON) annotation on the methods.

```java
@GET
@Produces(MediaType.APPLICATION_JSON)
public Stage getStage (int idEtudiant) {
        Stage stage=new Stage();
        stage.SetEvaluation(16);
        stage.SetCompetence("SOA, Rest");
        stage.SetAnnée(2021);
        return stage;
}
```

Fig. 8: Applying JSON format

We started by building a GET method to retrieve a list of students, and the output was formatted as JSON. We also created a POST method to allow users to add a new student by sending a JSON payload. Here, we configured the method to consume JSON using the @Consumes(MediaType.APPLICATION_JSON) annotation.

The workflow to add a new student involved sending a POST request with the following JSON payload:

```
{
   "binome": {
      "nom": "Jaulhiac",
      "prenom": "Paul"
   },
   "nom": "Vasseur",
   "prenom": "Cyril"
}
```

We used Postman to test these requests, and the server responded with confirmation messages. This process taught us how to serialize and deserialize JSON data, allowing us to seamlessly transfer structured data between the client and server.

5. Implementing HATEOAS (Tuto4-Rest-HateOAS.pdf)

(Cf. RestProject/src/main/java/fr/insa/ws/soap/RestProject/ , in GitHub)

The final tutorial focused on HATEOAS (Hypermedia As The Engine Of Application State), a key principle of REST that allows API clients to discover actions they can perform based on the links embedded in responses.

We extended our Etudiant class to include an association with a Stage (internship). This meant that when we queried for an Etudiant, we also wanted to see the related Stage. To achieve this, we created a Link class to define hypermedia links. Each Link object included the URL of the related resource, the HTTP method to access it, and the relationship (rel) to the main resource.

In our EtudiantRessource class, we updated the GET method so that each response returned hypermedia links. For example, when querying a student with GET /etudiant/1, we received a JSON response like this:


```
{    "id": "1",
    "nom": "Vasseur",
    "prenom": "Cyril",
    "_links": [
        {
            "rel": "self",
            "uri": "http://localhost:8080/RestProject/webapi/etudiant/1"
        },
        {
            "rel": "stage",
            "uri": "http://localhost:8080/RestProject/webapi/stage/1"
        }
    ]
}
```

This response informed us that, in addition to viewing the student's details, we could also access their Stage details via the provided link. This approach allows for creating discoverable REST APIs where clients can navigate resources dynamically, without prior knowledge of the API's endpoints. We tested this by making GET requests via Postman and confirmed that the responses contained the expected hypermedia links.

Reflection on the Practical Works

These practical works gave us a comprehensive approach to understanding and building RESTful web services. We began with the server setup (Tomcat) and progressed through service development, client development, JSON serialization, and HATEOAS. Each tutorial introduced new concepts and practical tasks, allowing us to experience the complete lifecycle of a REST service.

To recap, we learned how to:

1. **Set up a server** and configure it for REST service deployment.
2. **Create RESTful services** with endpoints for **GET, POST, PUT, and DELETE**.
3. **Build a REST client** to consume services programmatically.
4. **Send and receive JSON data**.
5. **Implement HATEOAS** to make APIs discoverable and self-descriptive.

3/ Microservice architecture for a Service Application

1. MicroService Architecture

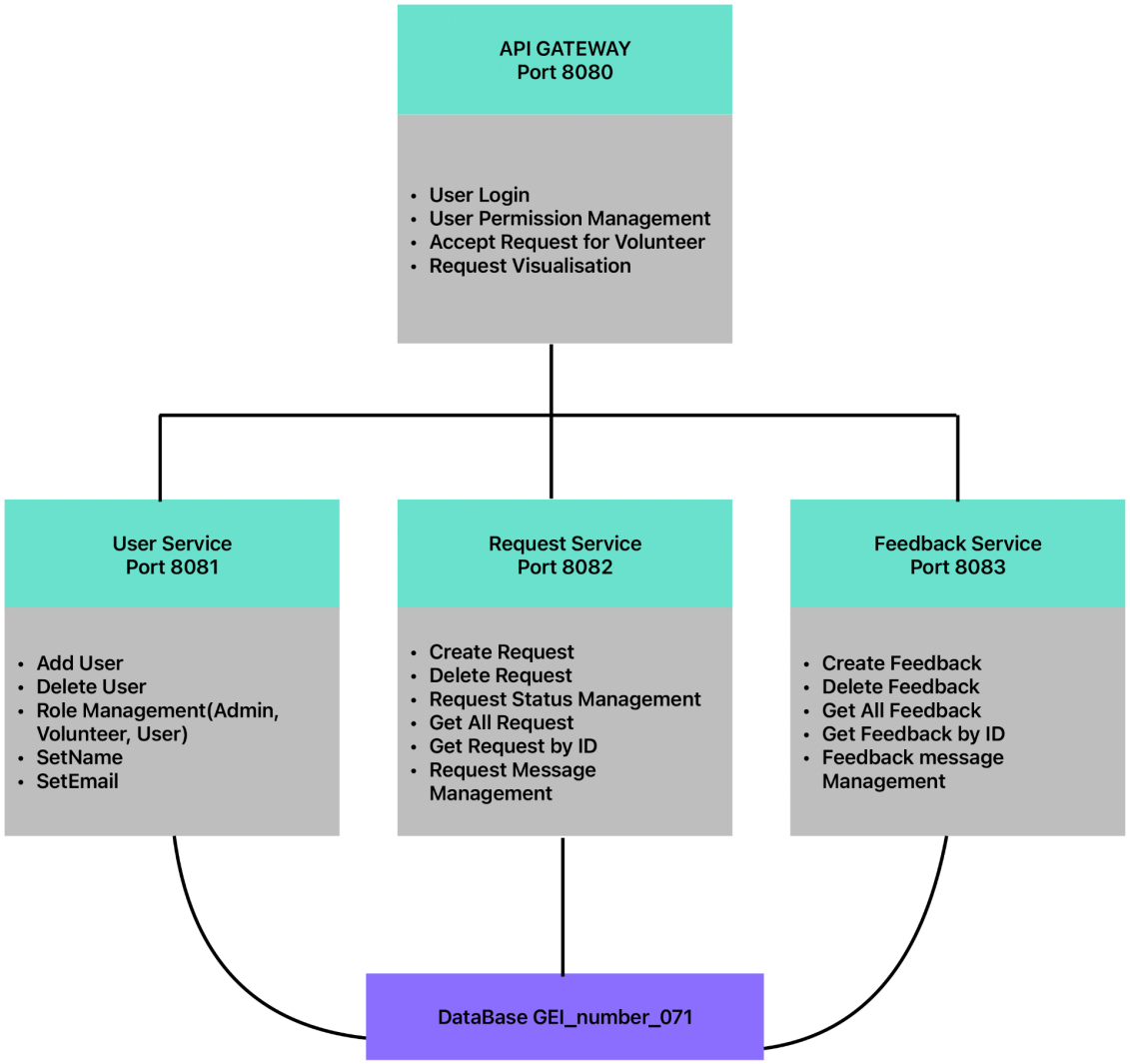

Fig. 9: Microservice Architecture build on VS code

As you can see in the picture above, we are now planning on implementing a microservice architecture based on 3 microservices and 1 gateway that can access each microservice which are independent. The only thing that is common on each microservice is the database located at INSA Toulouse. The database is using JSON data format to save the data received from a microservice.

2. How to reach the DataBase

```
# URL de la base de données MySQL
spring.datasource.url=jdbc:mysql://srv-bdens.insa-toulouse.fr/projet_gei_071
spring.datasource.username=projet_gei_071
spring.datasource.password=OoShees4
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Spécification de la dialecte MySQL
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

# Gestion des entités JPA
spring.jpa.hibernate.ddl-auto=update

# Affichage des requêtes SQL
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Fig. 10: Set-up code to access the database

The code shown above corresponds to the part where we set-up the database information, to be able to access it and store our data on it. A specific base was given to each group. Ours was “projet_gei_071”, which we could connect to using our password, just like in the first lines above. It is important to mention that, to reach the Database, we had to be connected to the VPN at INSA Toulouse.

3. The User Service Microservice

```
@RestController
@RequestMapping("/users")
public class Usercontroller {

	@Autowired
	private Userservice userService;

	@GetMapping
	public List<User> getUsers() {
	return userService.getAllUsers();
	}
```

```
	@PostMapping
	public User createUser(@RequestBody User user) {
	return userService.createUser(user);
	}

	@DeleteMapping("/{id}")
	public ResponseEntity<String> deleteUser(@PathVariable Long id) {
	boolean isDeleted = userService.deleteUser(id);
	if (isDeleted) {
		return ResponseEntity.ok("User with id " + id + " deleted
successfully.");
	} else {
		return ResponseEntity.status(HttpStatus.NOT_FOUND).body("User
with id " + id + " not found.");
	}
	}
}
```

Fig. 11: User service, on VS code

We used a REST application to create all the user management functions. As you can see here, we have all the user creation and deletion system for our service request application. The objective was also to be able to have different types of users, such as administrators, volunteers to answer the requests, and a simple user, able only to request for help.

4. The Request Service Microservice

```
@RestController
@RequestMapping("/requests")
public class RequestController {

   @Autowired
   private RequestService requestService;

   @GetMapping
   public List<Request> getAllRequests() {
       return requestService.getAllRequests();
   }

   @GetMapping("/{id}")
   public Optional<Request> getRequestById(@PathVariable Long id) {
       return requestService.getRequestById(id);
   }
```

```
    @PostMapping
    public Request createRequest(@RequestBody Request request) {
        return requestService.createRequest(request);
    }

    @DeleteMapping("/{id}")
    public void deleteRequest(@PathVariable Long id) {
        requestService.deleteRequest(id);
    }
}
```

Fig. 12: Request service, on VS code

Here we have another REST application, to allow us the creation of "service requests". This is what a user would typically create to request assistance from a volunteer, under validation of an administrator. We were hoping to implement a status for each request too: "WAITING, VALID/UNVALID, AFFECTED, COMPLETED". Unfortunately, we didn't manage to make it in time.

5. The Feedback Service Microservice

```
@RestController
@RequestMapping("/feedbacks")
public class FeedbackController {

    @Autowired
    private FeedbackService feedbackService;

    @GetMapping
    public List<Feedback> getAllFeedbacks() {
        return feedbackService.getAllFeedbacks();
    }

    @GetMapping("/{id}")
    public Optional<Feedback> getFeedbackById(@PathVariable Long id) {
        return feedbackService.getFeedbackById(id);
    }

    @PostMapping
    public Feedback createFeedback(@RequestBody Feedback feedback) {
        return feedbackService.createFeedback(feedback);
    }
```

```
    @DeleteMapping("/{id}")
    public void deleteFeedback(@PathVariable Long id) {
        feedbackService.deleteFeedback(id);
    }
}
```

Fig. 13: Feedback service, on VS code

This REST app is how we implemented the possibility to leave a comment, on a request. This is how any kind of user is able to give feedback on his experience using the app, or while helping someone or receiving help. The feedback is associated with the user ID, to be able to trace them easily.

6. The Gateway to use the Microservices and visualize them with a html.

```
@RestController
@RequestMapping("/gateway")
public class GatewayController {

    @Autowired
    private UserService userService;

    // Créer un utilisateur
    @PostMapping("/createUser")
    public String createUser(@RequestParam String name, @RequestParam String email,
@RequestParam String role) {
        return userService.createUser(name, email, role);
    }
    // Créer une demande d'aide
    @PostMapping("/requests")
    public String createRequest(@RequestParam String userEmail, @RequestParam String
description) {
        return userService.createRequest(userEmail, description);
    }
    // Créer un feedback
    @PostMapping("/feedbacks")
    public String createFeedback(@RequestParam String userName, @RequestParam String
feedbackText) {
        return userService.createFeedback(userName, feedbackText);
    }
}
```

Fig. 14: Gateway implementation, on VS code

Fig. 15: User interface, on a web page

Above, in figure 14, we have the code for our gateway. It is this system which allows us to link all the services and to have a working interface to put them to use. Although it is very simple, it is functional and allows us to create users by entering their name, email address and function inside the application (Admin, User, Volunteer). We can then create service requests by completing the user's ID (email @) and giving a description of the kind of help needed. Finally, it is possible for a user to give feedback on his experience by giving his ID again. All the data created via this application, through the interface, is uploaded to our database and fully readable and accessible from there.

7. Setting-up an Azure server to automatize microservices

As part of our project, we tried to deploy a Spring Boot service on Microsoft Azure as a web application. Our approach incorporated key DevOps principles, specifically focusing on Continuous Integration (CI) and Continuous Deployment (CD). By leveraging the capabilities of Microsoft Azure and GitHub Actions, we automated the entire process of building, packaging, and deploying the service. The service we chose to deploy was a Spring Boot microservice named **request-service**.

The first step in the process was to set up the necessary resources on Azure. We created a **resource group** named **MsDeployment-group**, which allowed us to logically organize and manage all the resources associated with our project.

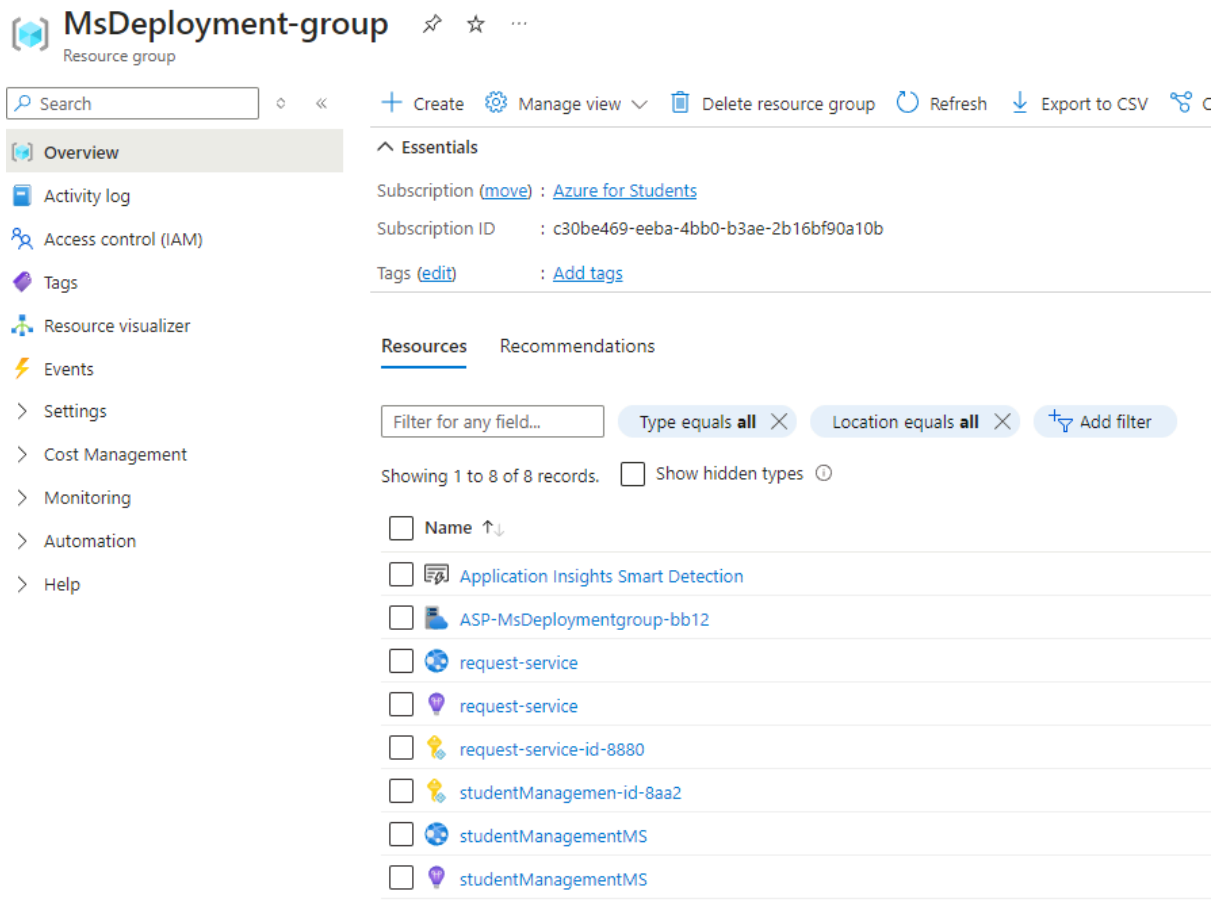

Fig. 3: Azure deployment group

Once the resource group was ready, we moved on to setting up our web application. We used Azure's **App Services** to create a new **Web Application**. During the configuration phase, we associated the app with our previously created resource group, gave it a unique name — **request-service** — and specified the Java Development Kit (JDK) version required for our application, which in this case was JDK 11. Once we validated the configuration, we just had to let Azure create the app.

With the web application in place, we shifted our focus to integrating it with our GitHub repository. Our goal was to create a direct link between our source code and Azure's deployment pipeline. To achieve this, we accessed the **Deployment Center** of our web app, where we specified the **source** of our microservice code. We opted to use our GitHub repository as the source and granted Azure permission to access the repository.

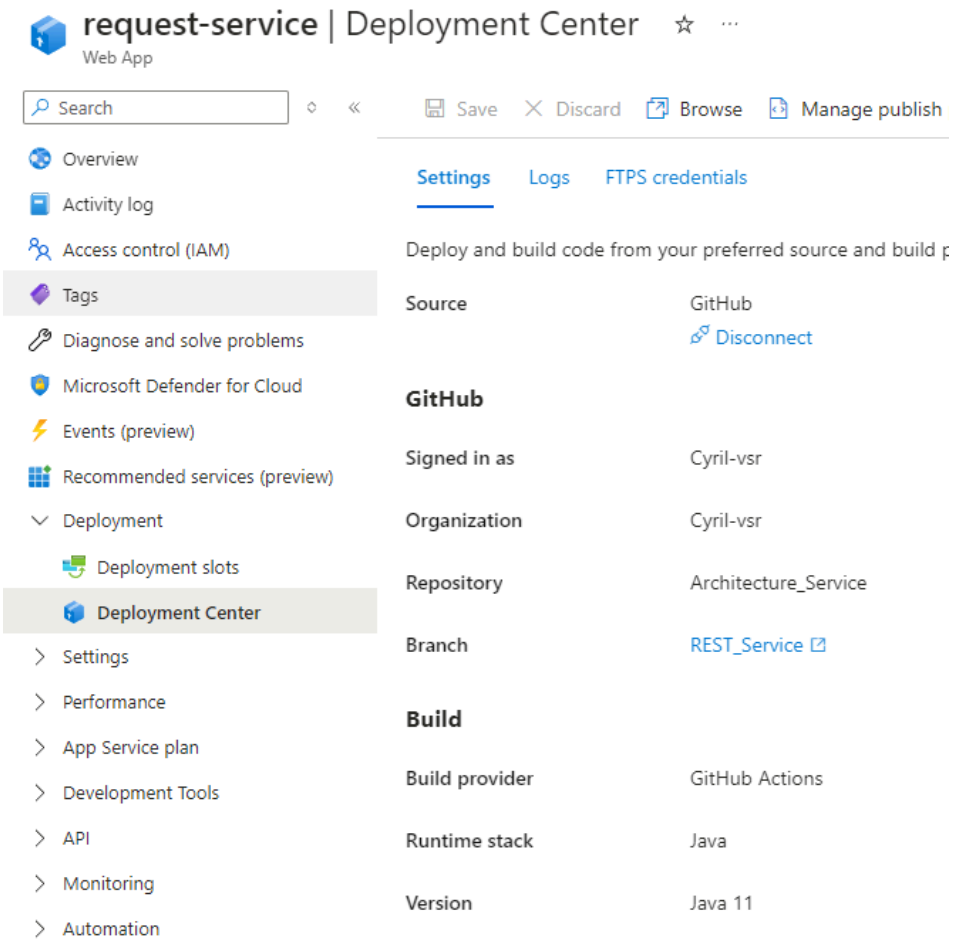

Fig. 4: Azure deployment center and link to GitHub

Once the connection between Azure and GitHub was established, we selected the specific GitHub repository, branch, and directory that contained our Spring Boot microservice project. At this stage, Azure automatically generated a YAML script that outlined the process for building and deploying the service. We reviewed this script to ensure it aligned with our project requirements. The YAML script consists of a clear structure that defines the steps of the CI/CD process. The first step is a trigger, which is set to activate every time a **push** is made to the chosen branch of the repository. The second step is a **build process**, where Maven commands (mvn clean install) are executed to verify the integrity and build success of the project. Finally, if the build step is successful, the third step is the **deployment process**, where Azure automatically deploys the resulting JAR file to the App Service.

This YAML script was saved in our GitHub repository under **.github/workflows/rest_service_request-service.yml**. We were able to monitor the execution of the workflow directly through the **GitHub Actions** tab. Here, we could view a detailed log of each action, from the build phase to the deployment phase, and verify that each step had been executed successfully.

Fig. 5: Actions tab in GitHub, with logs for every commit

Unfortunately, we couldn't have a live version of our microservice running on Azure, because of an error appearing each time the code tried to run "mvn clean install" command. Otherwise, we would have been provided with a public URL (request-service-eaf0dscjb3hgcygj.francecentral-01.azurewebsites.net), allowing us to access and test the microservice in real-time. By adding specific API paths to the URL.

This project still allowed us to gain hands-on experience with Azure's cloud platform and the CI/CD pipeline offered by GitHub Actions. It gives us a previous experience in DevOps field, which will prove valuable for us, in the future, in the industry.

CONCLUSION:

To conclude on this lab, we didn't succeed to implement all the functions in the gateway application due to the time restriction but we still have the architecture already thought out. The principal methods are already designed and implemented. We are optimistic that we would be able to produce a cleaner and more complete version of this application, with more time ahead of us. Whatever the current result, we are happy with the work we could produce, because we believe that it helped us understand better how Microservice Architectures are made. This is why we focused on providing a basic application, with the main elements of this course, to harden our skills in Service Architectures.