Service Architecture

MARIN-MULLER Robin, BOUJON Yohan

Institut National des Sciences Appliquées de Toulouse

25 janvier 2025

Table des matières

1 Introduction

Repository Link : https://github.com/Lemonochrme/service-architecture

Moodle Link : https://moodle.insa-toulouse.fr/course/view.php?id=1044

The goal of this project is to build an application to assist vulnerable individuals through a micro-service-based architecture. The core functionalities will be around user, authentication, requests and feedbacks. To make this application use the micro-service principle, we will focus on building a RESTful API. With this design, flexibility will be the core, and each independent services can be used in any way possible while keeping user integrity.

In Section 2 we will be focusing on is the principle of both SOAP and REST. The choice between one of the two protocol to share data between application is essential. While SOAP is complex and rigid, REST is the norm on the modern internet and is around the evolution of what we call the Web 2.0. Performance and scalability are masters in this work. The shift from a monolithic architecture to using multiple micro-services can be quite confusing at first, but can enhance development greatly. By adding multiple layers of abstractions, developers can use pure data without taking care of programming languages or complex bindings.

In Section 3 will focus on how we designed our back-end using Sprint Boot. Each micro-service functionalities, and how it can be used for a complete front-end. We will focus on security for the users and how they can interact with requests and feedbacks.

The Section 4.1 will be focused on the way we develop our project, how we compile and run everything. Using CI/CD technology we will be showing how to correctly design a good working flow, from the raw compilation to the full deployment on a server.

2 Starting the project

The project aims to be an application to help vulnerable people, the goal would be to create multiple micro-services with a system of users, requests, feedbacks. The main focus would be to create full REST API which would enable anyone to make a frontend application with the service we provide.

2.1 SOAP principles

Before getting into the REST API. We need to first understand SOAP. SOAP or *Simple Object Access Protocol* has been created in 1998 by a team at Microsoft. SOAP was the first protocol to enable the Web 2.0 we know today, it added structure to communication on the web, making it the first standard when it comes to sharing data. It uses **WSDL** or *Web Service Description Language*, a **XML**[1] based language, to format its data. The advantage of such format is its ability to have strictly defined rules with the help of an XSD file. In itself, XML is quite slow, verbose and has an enormous size compared to modern alternatives such as **JSON**[2] or **YAML**[3]. Overall, SOAP has the advantages of being rigid, so users don't do anything with its data, it can still be flexible because it is only bytes transported through the internet. However, the formatting is large, parsing XML is slower than alternatives, SOAP has to respect each languages independently and it doesn't provide good protection concerning software legacy (versioning).

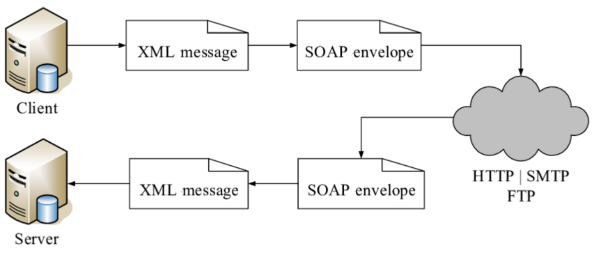


FIGURE 1 – [1] SOAP Workflow

SOAP is close to its end of life, with the gaining popularity of RESTful paradigm, and its complexity, software tends to switch to a more simple approach. Like in the OSI[4] model, each layer has to abstract its behaviour.

2.2 REST principles

REST or *REpresentational State Transfer* has been designed in 2000 by Roy Fielding in a paper focusing on the HTTP protocol and how it could enable multiple applications to communicate without the barrier of programming languages. REST focuses on 6 constraints : performance, extensibility, simplicity, scalability, portability and reliability. Using HTTP to send values to the server, clients can cache the data gathered to keep the communication with the server. The advantage of such a system is that the server and the client do not need to be constantly communicating. REST is closely tightened with **URI**[5] which enables frontend application to communicate with easy to read, human-readable requests such as *GET, POST, PUT, DELETE*. The goal is to remove any ambiguity.

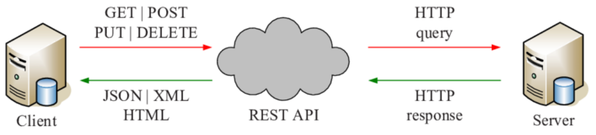


FIGURE 2 – [1] REST Workflow

REST does not have the ability to tell the user which data its need to actually make the call. This let the user entierly on their own and has to test if some parameters work, hopefully a good documentation can help. In the other end, because SOAP is strictly defined, with WSDL, it provides directly to the user what data to send. However, REST and RESTful applications seems to be a better choice, it uses a message format way less dense than XML because it can be entierly customized by the server *(usually JSON)*. REST is easy to use for any developper so gathering data from a REST application is fast. It has been built around the open paradigm which enhance its portability all over the internet for any application.

2.3 Micro-service Architecture

We made during all our school years monolithic applications. These are easy to make at first but can quickly be hard to maintain, scaling up such software requires to rebuild the whole application and can break things. SOA *(Service-Oriented Architecture)* is designed around making sets of independently deployable services. It allows scaling because only the required part can be upgraded. Such an architecture can benefit the developer and business-driven development. Enabling companies to decentralize multiple services and let the customize choose what would be the best for him, whereas monolithic architecture gives the user a full bundle directly.

1. **XML** : Extensible Markup Language
2. **JSON** : Java Script Object Notation
3. **YAML** : Yet Another Markup Language
4. **OSI** : Open Systems Interconnection
5. **URI** : Universal Resource Identifiers

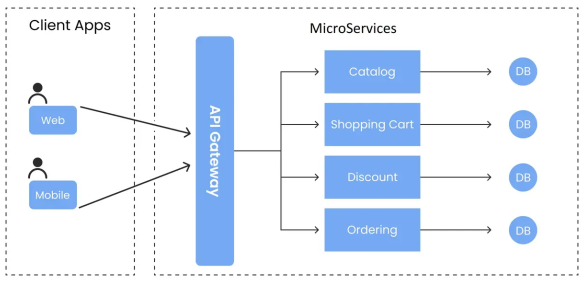


FIGURE 3 – Micro-service Architecture

The modularity of such a system enhance portability, technology heterogeneity, resilience, scaling, ease of deployment and organisational alignment *(multiple little teams can work on different micro-services, whereas in monolithic application can only be maintained by a big team)*. Micro-services are usually designed around the Web because it is the only platform that can exchange data without any language barriers. This is why our application will focus on REST. We will use the **Java** programming language but each micro-service could be designed around other languages : the only rule is to send the data in the same format.

3 Micro-services

3.1 Why Spring Boot ?

Spring Boot is a Java Framework used to simplify the configuration of big Java project that uses web functionalities. It embarks *Apache Tomcat*, the famous Java HTTP web server environment and builds over the *Maven* Java toolchain.

This framework is oriented around SOA and micro-services, it can packages multiple *.jar* from the same project. It focuses mainly on REST/SOAP, perfectly fitting our project needs. Furthermore, we do not have much time to finish this project, Spring Boot can greatly enhance the development time and therefore make every micro-service fully work.

It is possible to make a Full-stack application using only Java and Spring Boot. However we did not go into that route. Because programming and understanding the Backend in full Java can be time consuming, we made our Frontend entirely in pure *HTML/Javascript* : gaining time while making everything work accordingly.

3.2 Manager : *Database*

As part of the apprenticeship formation of the Automatic and Electronic, we did not have any course around designing a data base. Hopefully this was not the first time we did this work around databases so we came up with a diagram that could work with the desired application.

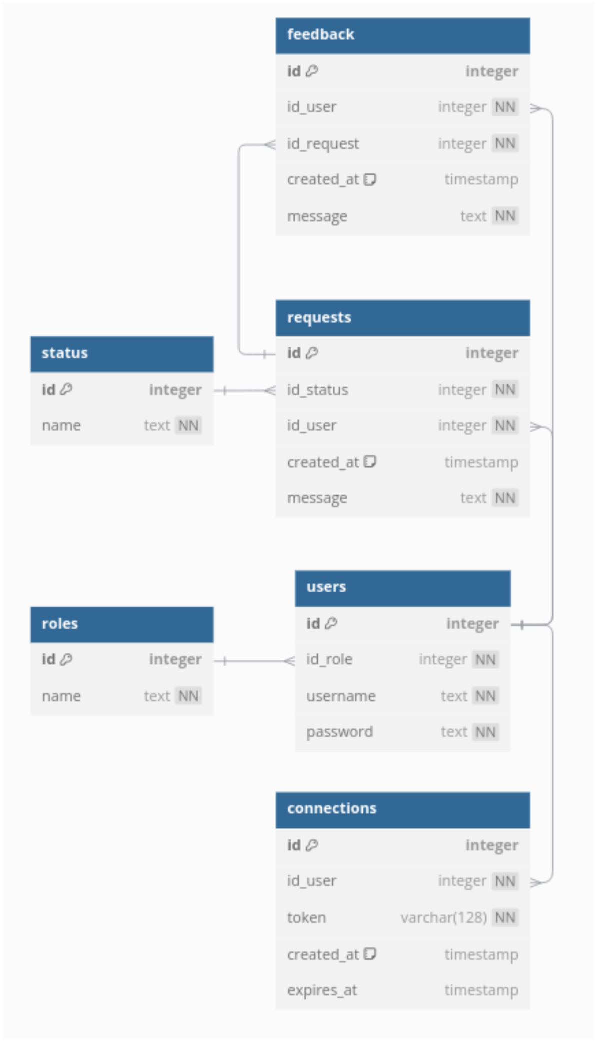


FIGURE 4 – Database diagram

As we can see in the diagram above, **Users** are linked with a foreign key *role*. There are 3 roles that are defined in the subject :

- **Users** : Whose can create a request, respond to their request and set it to *"Finished"*
- **Admins** : They can change to any status, respond, remove any request. They are the one responsible to *"Validate"* or *"Reject"* a request from a user.
- **Volunteers** : They can switch a request from *"Validated"* to *"Selected"*. Once they did their duty they can respond to the request.

Each User with the role *"User"* can create a **Request** with a given status. Once created any *"Admin"* can modify its status to either *"Validated"* or *"Rejected"*. As we can see the Request has 2 foreign keys : Status, which has been explained already, and Users which is the author of the request. There are multiple status which are defined in a static data base file :

- **Waiting** : The default status for any request. Notifying every administrator that they have to make choice.
- **Validated** : Status reserved for administrators, once validated, a volunteer can select this request.
- **Rejected** : Status reserved for administrators, the request is rejected.
- **Selected** : Volunteers can select a request that have been previously validated.
- **Finished** : Once selected, the author of the request can change the status to finished.

Feedback can be done by either the author of the request or any user who volunteered.

Connections is a set of users with a given token and a timestamp. The token can be gathered when connecting and has to be stored on the browser. They last until the expiration given in the database. When doing any action regarding requests, feedbacks, the token has to be sent. This is our solution to user authentication.

To add communication between the database and our Spring Boot instance, we need to use *Jakarta Persistence API. JPA* is an interface from databases like MySQL to Java, using **JPQL**[6] it can be easy to design a class entirely based on its database counterpart. A set of predefined functions can directly communicate with our MySQL database. Here is an example with the Role object :

```
@Entity
@Table(name = "roles"
, schema = "service-architecture")
public class Role {
@Id
@GeneratedValue(strategy
= GenerationType.IDENTITY)
    private int id;
@Column(name = "name", nullable = false
, unique = true)
    private String name;

    // Getters and Setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

6. **JPQL** : Java Persistence Query Language

Which pretty similar to our database integration :

```
CREATE TABLE `roles` (
  `id` integer UNIQUE PRIMARY KEY AUTO_INCREMENT,
  `name` text UNIQUE NOT NULL
);
```

As we can see, with the different Annotations such as *Table, Column, GeneratedValue, Id* : *Jakarta Persistence* can easily understand how our database is designed and will be doing SQL requests with our JPQL functions. To instantiate a *Jakarta Persistence* JPQL instance we need to create the following object, inherited from *JpaRepository* :

```
public interface RoleRepository
extends JpaRepository<Role, Integer> {
    //List<Role> findById(int id);
}
```

By calling simple functions such as *findById*, we are making a JPQL which is directly translated into an SQL query. We can even make custom functions, for example *findByIdRequest* in the *FeedbackRepository* class. Comparing to regular backend using other languages such as *Javascript* or *GO*, *Java* here make it really simple by removing already defined queries for the database while keeping it customizable for complex more complex use cases.

Once everything is set we can simply bundle the manager to its own *.jar* file. Micro-services will be using all these classes to enable a communication with the **REST API**. The only goal of this manager is to abstract the database layer which is none of our concern. Services uses the database as key values and a *JSON* file could do the same.

3.3 Service : *Role*

"Role" defines the way a connected user can interact with all the other API resources. It checks the permissions it can deliver and can be dynamically changed. We could have made so the backend server would have this table, however with this being inside the database we can make foreign key calls directly for each user.

This service only delivers the **"/get_roles"** *GET* endpoint, which returns the roles defined inside the database. This is useful in the frontend part to show a dropbox of all the roles available when creating an account. Here is the output :

```
{
"id": 1,
"name": "User"
},
{
"id": 2,
"name": "Volunteer"
},
{
"id": 3,
"name": "Admin"
}
```

We only need to gather the json parsing both the name and the id, and create for each value gathered an *<option/>* element in the HTML.

3.4 Service : *User*

The User service manages everything from the account perspective. The creation of a user, gathering the username/role from an id, connect using the username and password.

The **"/create_user"** is a *POST* endpoint asks for an *idRole*, *username* and a *password*. it can only fail if the username has already been taken. it can either return the user created with its id :

```
{
  "id": 4,
  "idRole": 1,
  "username": "yoboujon",
  "password": "1234"
}
```

Or an internal server error (500) showing that the username has already been taken.

The **"/get_user"** *GET* endpoint asks for a user *id* and returns its username and role. Because the browser will be storing only the id, this could be useful to show the username and the role on other parts of the website.

The **"/connect"** *POST* endpoint asks for a *username* and a *password*. It will directly communicate with the *Connections* table from the database by asking for a new token. The first step is to verify if the username exists, which if it doesn't will send back a bad request error (400) following by the string "Username not known". If the password is not correct, it will send back the Unauthorized error (401) with the following string "Invalid password.". If both parameters are correct, the server will send back a token in the following format :

```
{
  "userId": 4,
  "expiresAt": "2024-12-25T12:50:20",
  "token": "GTEMCc45T..."
}
```

The given token can be used later to send requests and feedbacks, so it has to be stored inside the browser for later use. An expiration date has been given, for now, the token expires 1 day after its creation. When reconnecting, the token can be changed if it expired, otherwise, it will send the same generated from the day prior.

3.5 Service : *Request*

Before getting into the request's logic, we need to go back into the Database Manager. A class named *AdministrationService* has two methods which can be used in every following REST API. The first one being *checkToken* which, as its name implies, checks if the token is valid with a given user id. It returns false if either the user does not exists or the token is expired/invalid. Returns true otherwise. Another function is the *getRole* which from a user id gather its role : it can return an empty value which shows that the user does not exists.

Now that this specific class has been identified, we can discuss about each endpoints of the Request Service, keep in mind that any *POST* or *PUT* method will ask for the user id and the token stored in the browser's cache. The first one being a *POST* method with the **/create_request** endpoint : checking first if the user is a volunteer. If it is the case, it will return a Forbidden error. Then it creates the request and sends back the created request :

```
{
  "id": 1,
  "idStatus": 1,
  "idUser": 4,
  "createdAt": "2024-12-24T13:34:47",
  "message": "Hello, World!"
}
```

As we can see, it created the request with the *WAITING* status automatically and the actual date. It will then be moderated by the administrators to change its status.

The **"/change_status"** is a *PUT* endpoint asks for a *Request Id* and a *Status*. Only administrators can modify however they please the status. Once "validated", the volunteers can set the status to "selected". Once finished and selected, the users can update the status. If the status number is too high there will be an error returned, as well as when the user is not allowed to change to a specific status. Once completed, it sends back the modified request.

```
{
  "idStatus": 3
}
```

We send back the entirety of the request in reality, to be sure any other data is coherent.

The **"/get_request"** is a *GET* endpoint which returns every request send in the following format :

```
{
"id": 1,
"idStatus": 3,
"idUser": 4,
"createdAt": "2024-12-24T00:00:00",
"message": "Hello, World!"
},
{
"id": 2,
"idStatus": 1,
"idUser": 3,
...
"message": "Hello, Admin!"
},
{
"id": 3,
...
"message": "Other Message"
}
```

There are no known error unless there are no request that are in the database. Which just sends back an HTTP Status "Bad Request".

3.6 Service : *Feedback*

The Feedback Service manages any feedback that could be made from a given request.

The **"/create_feedback"** *POST* endpoint asks for an *idRequest* and a *message*. The request can have any status to have a feedback, but it can only receive a feedback from a volunteer if it is validated. For users, only the author can give a feedback. If these two conditions are not met, it will send a forbidden HTTP status. Otherwise, it will resend the saved feedback. Another case where the status could be a bad request is when the request Id is not recognized.

```
{
  "id": 1,
  "idUser": 3,
  "idRequest": 1,
  "message": "Hello, response!"
}
```

As we can see the new message has been sent.

The **"/get_feedback"** *GET* endpoint asks for an *idRequest* and will show all the feedback a request has been receiving.

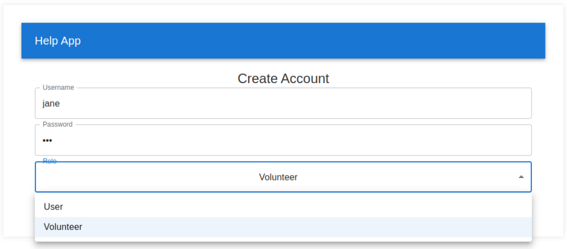
```
{
"id": 1,
"idUser": 3,
"idRequest": 1,
"message": "Hello, response!"
},
{
"id": 2,
"idUser": 4,
"idRequest": 1,
"message": "Can you actually
respond to the request??"
}
```

We would add a timestamp later and maybe not send back the request id which is taking bandwidth for nothing.

4 Software Engineering

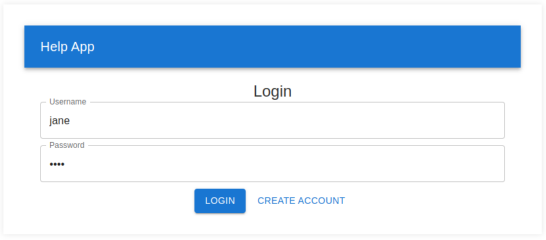
4.1 User Interface

We made a very minimalistic Vite + React user interface to streamline the user experience.



Username: jane
Password: •••
Role: Volunteer
User
Volunteer

FIGURE 5 – Register Screen



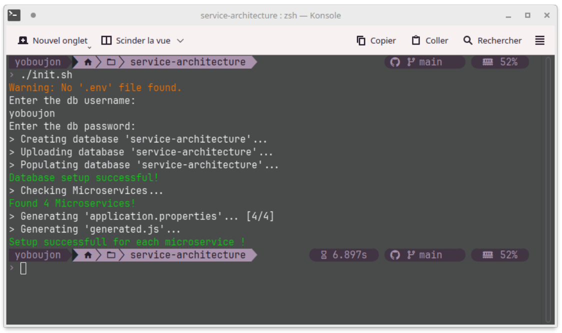
Username: jane
Password: ••••

FIGURE 6 – Login Screen

4.2 Bash scripts

A set of different bash scripts have been made to make development easier and to make our project portable around all the computer we use. When making complex project that connects multiple paradigm and software, it is important to make the simplest configurability possible. The steps to build our project are directly in the Github repository and showed us that it has indeed speed up the development process.

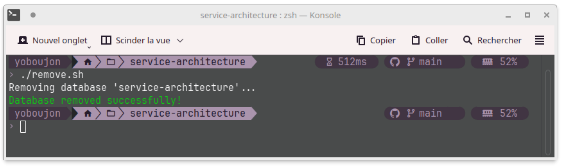
"init.sh" is a bash script that initialize the database by uploading the tables and copying some basic values such as the roles, the status and 3 users to test things out. It asks for the username and the password of the local database, but it can be connected to the INSA database as well. Once the database setup, it analyses how the project's backend is setup. For each *"-service"* folder, it will create a custom *"resource/application.properties"* with a different port for each of them. then for the frontend it will create a *"generated.js"* with all the correct endpoints with human-readable constants.



```
yoboujon  service-architecture                          main   52%
› ./init.sh
Warning: No '.env' file found.
Enter the db username:
yoboujon
Enter the db password:
> Creating database 'service-architecture'...
> Uploading database 'service-architecture'...
> Populating database 'service-architecture'...
Database setup successful!
> Checking Microservices...
Found 4 Microservices!
> Generating 'application.properties'... [4/4]
> Generating 'generated.js'...
Setup successfull for each microservice !
yoboujon  service-architecture             6.897s      main   52%
›
```

Figure 7 – Output from the init script
-

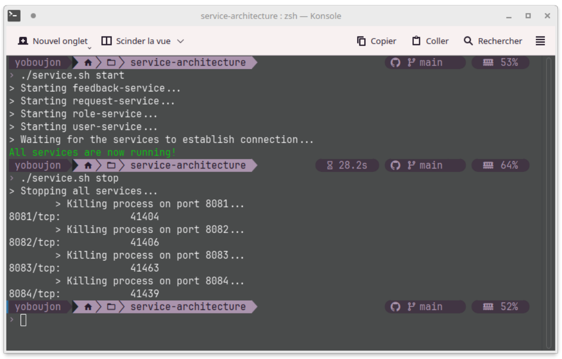
"remove.sh" can be used to clear out the database, the *.env* file is still intact so the initializer can be called right after.



```
yoboujon  service-architecture             512ms      main   52%
› ./remove.sh
Removing database 'service-architecture'...
Database removed successfully!
yoboujon  service-architecture                        main   52%
›
```

Figure 8 – Output from the remove script

"service.sh" is a more complex script that can either compile, start or stop the micro-services with the given keyword. Start creates 4 threads that runs independently the services, checking at the end if every service is running. Stop search for Spring Boot instances and kills them. Before starting anything, it is advised to launch a compilation.



```
yoboujon  service-architecture                          main   53%
› ./service.sh start
> Starting feedback-service...
> Starting request-service...
> Starting role-service...
> Starting user-service...
> Waiting for the services to establish connection...
All services are now running!
yoboujon  service-architecture             28.2s      main   64%
› ./service.sh stop
> Stopping all services...
        > Killing process on port 8081...
8081/tcp:            41404
        > Killing process on port 8082...
8082/tcp:            41406
        > Killing process on port 8083...
8083/tcp:            41463
        > Killing process on port 8084...
8084/tcp:            41439
yoboujon  service-architecture                          main   52%
›
```

Figure 9 – Output from the service script

Because it can take a lot of time to run every services *(28.2 seconds here)*, and because each service are independent, we sometimes launch a simple service to test if it works accordingly. This script can be useful for deployment.

4.3 Continuous Integration

Software development practice that uses a configuration file to do certain step. For example setup, compilation, run tests. The goal is to detect if a given push has any errors and can be used for deployment later. There are some key principles behind *CI* :

— **Frequent Code Integration :** The goal is to check at every frequent push if the project can still be compiled and is still working.

— **Immediate Feedback :** The feedback given by the CI tools can help the developer know fast if changes have to be made. For example in complex multi-compilation programs : developers do not need to have every architecture or machines available in their home, they can simply but the toolchain in distant server.

In our case, we use this tool to simply test if the project builds well. This ensure that even if one collaborator has not tested its code, the CI toolchain tests it. In our case we use GitHub Actions but other tools exists such as Jenkins.

4.4 Continuous Deployment

After testing that the build process work, we can create what we call artifacts, binary files from the project that can be shared in the deployment server. Deployment is usually used for web application but can be a tool for low-level systems, to send these artifacts into a download centre for users to use.

Continuous Deployment is based directly on CI, this is why we talk about a CI/CD Workflow. After doing integration, the pushed code has to go into deployment. In our case we use Microsoft Azure for our server, but we can use other services. On Microsoft Azure, we had to create what we call a *Resource group* : this is the server that will contain multiple containers for every of our projects. On top of that, an *App Service* will be running our artifact and have its own domain name to test if the website is fully working.

5 Conclusion

In this study, we observed that while micro-services and web-based architectures can be beneficial for certain applications, they also come with notable downsides. For simpler applications, the overhead introduced by this paradigm may complicate the development process without offering significant advantages. Specifically, the reliance on HTTP protocols can add latency and make the project network-dependant, making it less suitable for real-time systems or non-web-based solutions.

Furthermore, the large volumes of data exchanged between services can create efficiency challenges. Designing such architectures also requires careful consideration, as it can be difficult to decide what should be a separate micro-service or part of a larger one.

In conclusion, micro-services provide advantages in terms of security, maintenance, scalability, and deployment speed. For a fully web-based solution like ours, these benefits are significant and align perfectly with our needs. However, their adoption should be carefully evaluated when working on projects requiring rapid development or involving complex architectures. In some cases, the added complexity of designing such a system can negatively impact the project.

Références

[1] Yevgeniy Fedkin et al. "Development and evaluation of the effectiveness of the integration gateway for the interaction of the learning management system with external systems and services of state information systems". In : *Eastern-European Journal of Enterprise Technologies* 3 (juin 2022), p. 30-38. doi : 10.15587/1729-4061.2022.258089.

[2] Aki Stankoski Application Engineer. *The SOAP Protocol*. en-US. Juill. 2020. url : https://www.serviceobjects.com/blog/the-soap-protocol/ (visité le 23/12/2024).

[3] *Beautiful Code : Another Level of Indirection*. url : https://www2.dmst.aueb.gr/dds/pubs/inbook/beautiful_code/html/Spi07g.html (visité le 23/12/2024).

[4] *Que signifie REST ? - Definition IT de LeMagIT*. fr. url : https://www.lemagit.fr/definition/REST (visité le 23/12/2024).

[5] Roy Fielding. "Architectural Styles and the Design of Network-based Software Architectures". Thèse de doct. Jan. 2000.

[6] *IBM Documentation*. url : https://www.ibm.com/docs/fr/was/9.0.5?topic=services-wsdl (visité le 23/12/2024).

[7] Suryaveer Chauhan. "A Microservice based Architecture for a Presence Service in the Cloud". en. Mém. de mast. Concordia University, août 2017. url : https://spectrum.library.concordia.ca/id/eprint/982880/ (visité le 23/12/2024).