

Service Of Architecture

**Brunetto Marie
Beghin Léa
Boukouiss Samia**

5 ISS

22/12/2024

Introduction

This technical report examines the evolution of the architecture of the Volunteering application, highlighting the SOAP, REST, and Microservices architectures. The application aims to connect individuals in need of assistance with volunteers ready to offer their support, while implementing features such as request validation and feedback submission.

Structure of the application

Before starting the application, we first needed to define how our application would be structured. Some requirements were given to develop the application: *Create an application putting in contact people seeking help for everyday tasks with volunteers using microservices.*

To proceed, we first had to define the nature and content of our microservices. After brainstorming, we came to 5 final services: User Manager, Publication Manager, Assignment Manager, Authentication Manager and Feedback Manager. However, being a test application that we have to develop in a short span, we decided to focus on the three first aspects: Users, Publications and Assignments. The evolution of our service structure is explained through this report.

Each one of these services manages a class that would be stored in a database. The classes can be described as follow:

- **User:** a class gathering all of the users of our application. Even though they all have different roles, they are sharing some common characteristics such as name, surname, age and mail address to contact them. The different classes of our application are:
 - **Help-seekers:** Person that signs up on the application to get help on some specific tasks, such as grocery shopping, clothes washing, transportation or having company to discuss with.
 - **Volunteers:** Person who gets in contact with and completes the tasks from the help seekers. They can also spontaneously propose their help for specific tasks.
 - **Checkers:** Some of the help-seekers might have some restrictions, for instance medical. The checker role is to ensure that the publications that are posted/accepted do not have a risk of harming them.
 - **Administrators:** The administrator has a global view of every publication and every user on the application, and is able to manage problematic publications by deleting them and warning the users.

- **Publication:** a class that will represent publications that will be published, consulted and interacted with. These will be used to put volunteers and help-seekers in contact. They can be created and seen by both volunteers and help-seekers, and could be removed by their creator, related checker (if any) or an administrator.
- **Assignment:** A link created when a user decides to apply to a task in a publication.

Each microservice would handle functions to create, delete, update and get one, many or all of the components it manages, be it users, publications or assignments.

Finally, we would have a final service to communicate with the front end of the project, and other services that will be developed later for other functionalities, such as discovery or configuration.

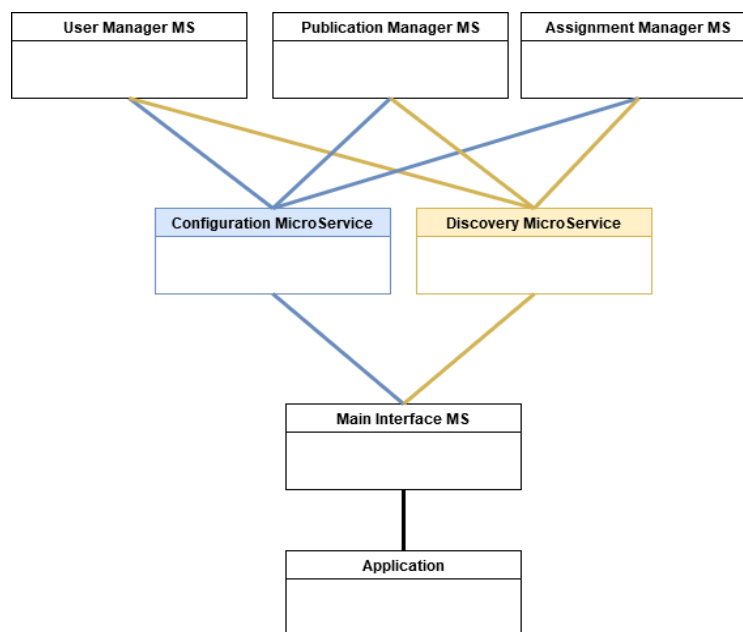


Figure 1 - Architecture of our microservices

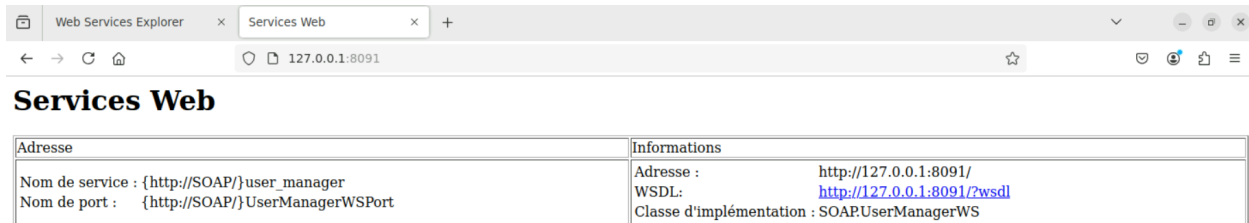
Architectures handled

I. SOAP Architecture

SOAP (Simple Object Access Protocol) is an XML-based communication protocol designed to exchange information between applications. It provides a strict structure for messages, which are typically transmitted via protocols like HTTP, enabling better interoperability between different platforms.

Testing our architecture through the Web Explorer

For this implementation, we decided to only develop the UserManager Web Service to try the architecture SOAP. We added the functions 'AddUser' and 'DeleteUser' to call and test out our service. Once the application runs as a service, we could preview it using localhost and the port it is set on. We were also able to test the function call through the provided Web Services Explorer.



The screenshot shows the 'Web Services Explorer' window with a tab for 'Services Web'. The address bar displays '127.0.0.1:8091'. The main content area is titled 'Services Web' and contains a table with two columns: 'Adresse' and 'Informations'.

Adresse	Informations
Nom de service : {http://SOAP/}user_manager Nom de port : {http://SOAP/}UserManagerWSPort	Adresse : http://127.0.0.1:8091/ WSDL: http://127.0.0.1:8091/?wsdl Classe d'implémentation : SOAP.UserManagerWS

Figure 2 - Preview of our Web Service UserManagerWS

II. REST Architecture

REST (Representational State Transfer) is an architectural style for web systems. Based on simple principles, REST uses URI to represent and address resources, relying on standard operations such as GET, POST, PUT, and DELETE, performed via HTTP requests. By utilizing standard representation formats like JSON or XML, REST ensures effective interoperability between systems, making it a preferred approach for developing robust web APIs.

We transitioned from our previous SOAP-based architecture, considered complex due to the use of XML, to a more flexible REST architecture. This shift redefined the way our services communicate, using URI to identify and access specific resources.

This migration to REST has modernized our system while enhancing its scalability and adaptability to meet the evolving needs of our technological environment. The classes in our architecture remain unchanged, but the annotations differ: we've added POST, GET, PUT, and Path (such as /user) to specify the access points to resources.

Unlike the SOAP application, we were not able to run the service and access it from Web Services Explorer. To test out the communication, we had to use:

- A server to host our services. We used **Apache Tomcat** as an HTTP web server
- An API platform, **Postman**, to communicate with the service. Even though we could use urls as we did for SOAP, we could not test some functionalities such as POST, because we are not able to embed a body with data in an url only.

Testing our architecture

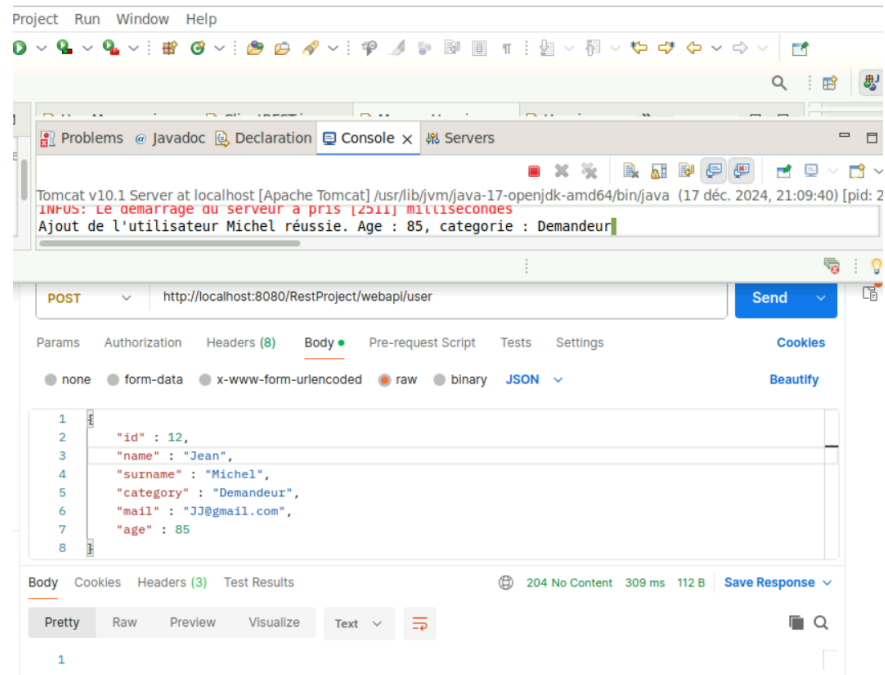


Figure 3 - Creation of a user in REST using Postman

III. Microservice Architecture

The REST architecture has some limitations, particularly an increased complexity in communication between resources. To overcome these challenges, we have decided to evolve our architecture towards a microservices model.

A microservice is an architectural approach that breaks down an application into several autonomous and independent services (called microservices). Each microservice is dedicated to a specific task and communicates with others through well-defined APIs. This method enhances flexibility, optimizes scalability, and simplifies maintenance of applications.

Consequently, we redesigned our services and transformed them into microservices. After careful consideration, the 'AddUser' and 'DeleteUser' services developed in SOAP at the start of the project concerned the same aspect of the application: user management. We grouped them together and were able to distinguish the two functions (add/delete) via the queries (CREATE/DELETE). These queries enabled us to bring together similar functions within the same microservice. This considerably reduced the number of services initially planned.

1. Collaboration of microservices

The collaboration between services is the ability for a microservice to call the API for another service. This can be useful if information is required, even though it is not managed by the current functions.

In our project, our three main microservices did not require communication between them. Even though they might have been linked in some way (a publication having a user as author, and an assignment being an association of a user and a publication), only the id is returned to be able to get the element from the right service.

The only service that uses collaboration is our main interface to the application. To be able to communicate with the APIs, we needed to recreate the classes used in the called API, as well as the url exposing the called service. For instance, to get the user of id 12, we used the following function:

```
User user = restTemplate.getForObject("http://localhost:8083/user/12", User.class);
```

2. Discovery of microservices

When an application decides to use services to divide the tasks, a problem can quickly be met: every service uses a port to expose itself, meaning that changing the port might require fixing every address in every call of the service.

One solution that is applied in microservices is the use of a discovery service. A discovery service is an intermediate service that translates the url to make it clearer and opaque to the service calling them. Every microservice has to set a name, a port and subscribe to a discovery service, using a dependency for instance. This will make the instance visible to any other service that wishes to call their API.

For our implementation, we use the Eureka server from the Spring library. This ensured the compatibility as we used spring dependencies for the creation of our microservices. The discovery service can for instance expose by changing the url from :

```
http://localhost:8080/user/12
```

to :

```
http://user-management-ms/user/12
```

a. Creation of a single instance of RestTemplate

A method annotated with `@Bean` has been added in the main class to create a single instance of `RestTemplate` when the application starts. This instance is also annotated with `@LoadBalanced` to indicate that it is used with Eureka for load balancing requests between services.

b. Using @Autowired to inject RestTemplate into the microservice

In the microservice class, a RestTemplate variable has been declared with the @Autowired annotation. This allows Spring Boot to automatically inject the appropriate RestTemplate instance when needed.

3. Creation of multiple instances

One of the main advantages of microservices architecture is its ability to respond effectively to scalability requirements. This is because it allows multiple instances of microservices to be deployed at the same time, in order to absorb the load and optimise the overall performance of the system. In the next section, we will set up the process for creating multiple instances of microservices.

Multiple instances can be created by configuring the application concerned. All that is required is to add a configuration and specify the port on which the second instance will be deployed. If the first instance (base instance) does not have a specific port, this must also be indicated to avoid any confusion. A second instance is then created and run at the same time as the first one when the application is launched. They are both visible on Eureka, as shown in the figure below. Each of them has its own port, 8183 and 8083.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
APPLICATIONBENEVOLE	n/a (1)	(1)	UP (1) - insa-20928.insa-toulouse.fr:applicationBenevole:8080
USERMANAGEMENTMS	n/a (2)	(2)	UP (2) - insa-20928.insa-toulouse.fr:userManagementMS:8183 , insa-20928.insa-toulouse.fr:userManagementMS:8083
General Info			

Figure 4 - Two instances of userManagementMS

This feature is particularly useful when the application is very busy. Several instances of the microservices will therefore be launched to distribute the load efficiently and minimise latency. Here, our two instances can share the task.

4. Service Configuration and its client

In a microservices architecture, configuration parameters play an essential role, from port numbers to database connection information. Until now, these parameters were defined in the application.properties file. However, this approach has significant limitations: each modification requires the microservice to be stopped, the parameters to be updated manually, recompiled and redeployed. At the scale of a complete application, this method quickly becomes inefficient and a source of major constraints.

To overcome these challenges, one solution is to externalise the configuration parameters via a dedicated microservice. Having set up such a configuration service, we will examine its use by

developing a client microservice capable of integrating with this configuration service in an efficient and flexible way.

a. Server deployment

The first step is to set up the configuration microservice using Spring Cloud dependencies. In order for the microservice to play the role of server, the `@EnableConfigServer` annotation must be present in its main application. In addition, with Spring Cloud, it is possible to create configuration files adapted to different development environments, such as development, test or production. This functionality, called profile in Spring Cloud, allows configuration parameters to be adapted to the execution environment.

In our case, we are going to focus on using the default and dev profiles, which correspond to the default configuration and the development environment respectively. To do this, we have created two files, `client-service.properties` and `client-service-dev.properties`, corresponding to the default and dev configurations respectively. These files have been placed at the root of the Git project so that they can be accessed by the configuration microservice.

By specifying the URI of the Git repository, we were able to access the file configurations by deploying the microservice.

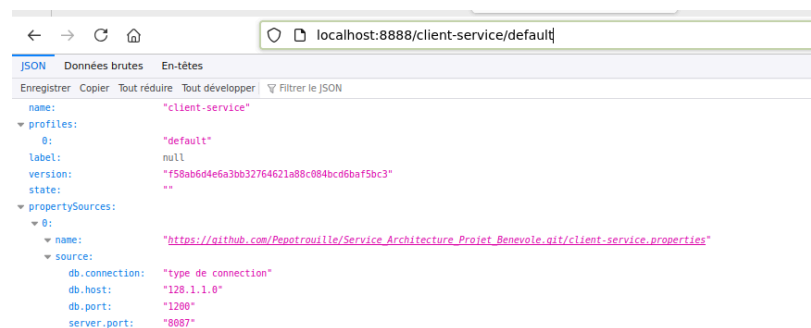


Figure 5 - Test of the default profile

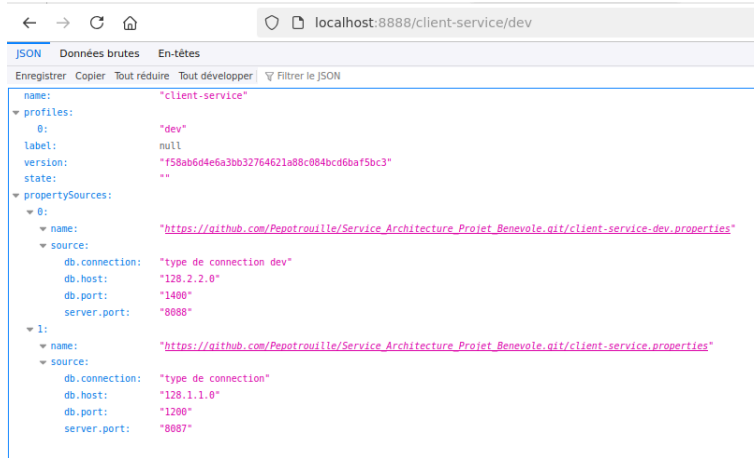


Figure 6 - Test of the dev profile

b. Client deployment

In this section, we set up a client microservice that will use the server microservice deployed earlier. The aim is to retrieve the configuration parameters from the two files in the git folder.

To set up this service, we need to specify the URL where the configuration server can be found. The parameters defined in the Java class must then be linked to those in the configuration file stored in the Git repository. For example, the `typeConnectiondeDB` parameter in the class must be associated with the `db.connection` parameter defined in the configuration file. To establish this correspondence, we used the `@Value("${parameter_name}")` annotation. By then writing GET requests, we were able to test the retrieval of the `dbHost` parameter. We were thus able to retrieve this parameter in both of the configurations.

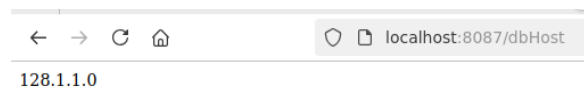


Figure 7 - dbHost of the default configuration



Figure 8 - dbHost of the dev configuration

5. Deployment via Microsoft Azure

This last part explores the implementation of continuous deployment (CD) of one of the microservices implemented in the previous sections. Among the various Cloud solutions available, the Microsoft Azure platform was chosen for this project. The objective is to deploy a

Spring Boot service as a Web application, a method that can be applied to other types of Web projects.

This implementation uses GitHub for collaborative management and GitHub Actions to automate the continuous integration and deployment processes. Microsoft Azure is used as the main platform for hosting and managing the deployed service.

To do this, we first needed to create the resource groups required to deploy the service. A resource group is a logical way of grouping different resources, such as applications, so that they can be organised and managed efficiently. Using 'App Services', we were able to create the web application corresponding to the resource groups. App Services can be used to manage different types of web applications. A summary of information relating to the service was displayed, including details such as the subscription ID, the operating system used and the size of memory available (for example, 1.75 GB).

Then, using the 'Deployment Centre', we configured the deployment of the application by specifying the source of the microservice code to be deployed. The aim was to automate deployment by connecting this source to a GitHub repository. Once configured, this step generated an automation script in the form of a YAML file (.yaml).

```
- name: Deploy to Azure Web App
  id: deploy-to-webapp
  uses: azure/webapps-deploy@v3
  with:
    app-name: 'user1ManagementMS'
    slot-name: 'Production'
    package: '*.jar'
```

Figure 9 - Deployment part of the YAML file

The YAML script was automatically added to the GitHub repository. Each time a push is made, the workflow starts automatically. Once execution is complete, you can view the results. In the example shown in the figure below, the workflow is currently running. If it is completed successfully, a green check symbol will appear.

Conclusion

The architectural choices depend on the specific needs of the project. SOAP offers advanced features, while REST stands out for its simplicity and compatibility with modern technologies. On the other hand, microservices aim to address scalability and maintenance challenges through a modular approach. The choice between these architectures depends on the project's requirements in terms of performance, flexibility, and simplicity.

However, throughout this project, we were able to see the advantages of a microservice architecture. Each microservice is used for a very specific function. They are independent of each other, so it is possible to modify one of them without having to stop the whole application. With the evolution of technologies and the growth in the development of web applications, these benefits are not negligible and make this architecture a considerable asset for service management.