

Service Oriented Architecture

Directed Labs Report

Romain Moulin & Aude Jean-Baptiste, 5ISS 2023

Table of Contents

Table of Contents.....	0
Context.....	1
SOA Architecture.....	1
1. Define the SOAP architecture of the application.....	1
2. Present the designed architecture to your teacher for validation.....	2
3. Implement a subset of SOAP services (1 or 2 services only).....	2
The code of our services.....	2
The WSDL of our services.....	5
Development of Restfull Services and REST API.....	7
1. Use the previously defined architecture.....	7
2. Migrate the developed SOAP services into RESTFull services.....	7
3. Analyze and compare the two models (SOAP and REST).....	12
Development of Microservices.....	12
1. Analyze the architecture you have designed. Are there limits?.....	12
2. Does this architecture conform to a microservice architecture norm?.....	12
3. Evolve you architecture toward a microservice one.....	12
4. Implement this architecture.....	13
Conclusion.....	15

Context

You can find the context of these directed labs [here](#) ¹.

Functionalities to implement :

- Add new users asking help
- Add new volunteers
- Add administrators
- Add requests
- Requests are validated
- Requests which are not validated, the reason is given
- A request has several status: waiting, validated/rejected, chosen, realized
- The users asking help and volunteers can give feedbacks

The code for the last part (microservices architecture) and much of the figures presented on this report can be found on [this github repository](#). ²

SOA Architecture

1. Define the SOAP architecture of the application

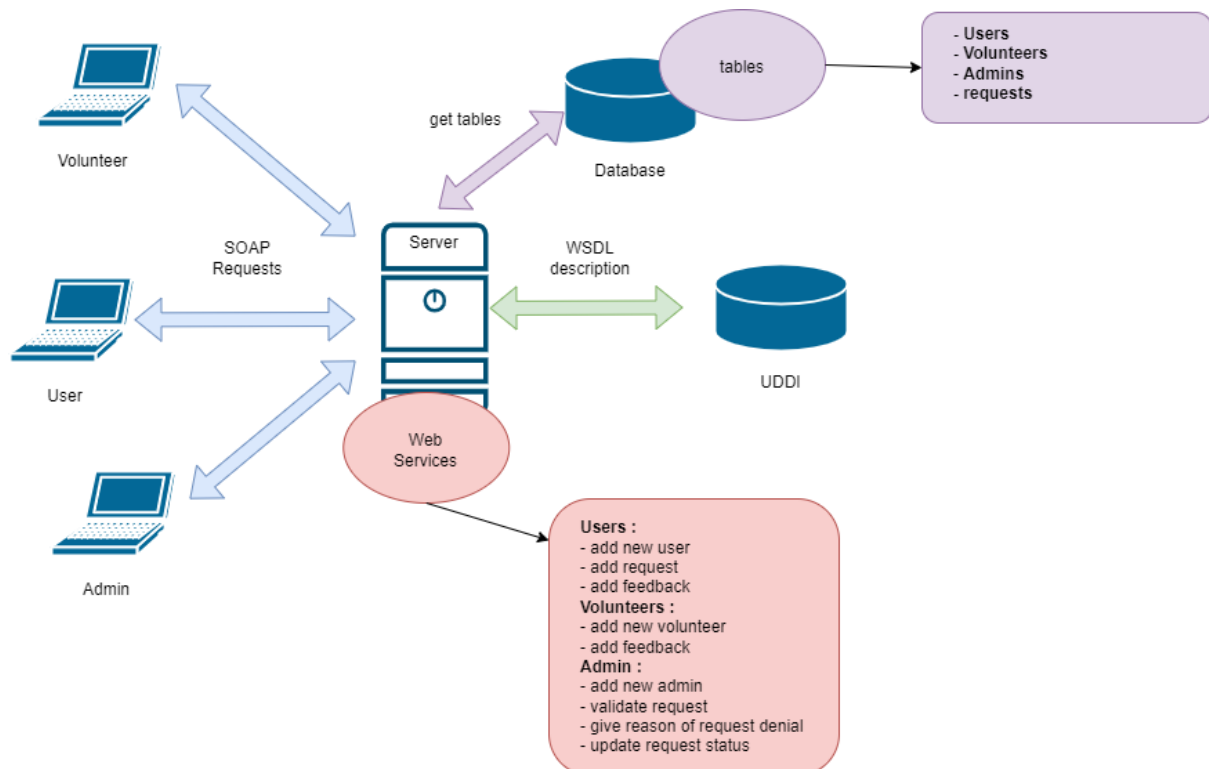


Figure 1: Our proposition of a SOAP architecture for this application³

¹ <https://moodle.insa-toulouse.fr/mod/resource/view.php?id=81554>

² <https://github.com/RomainMIn/ServiceArchitecture/>

³ made with draw.io. We could have used UML.

2. Present the designed architecture to your teacher for validation



3. Implement a subset of SOAP services (1 or 2 services only)

We implemented the services *add user* and *add request*. In this implementation the database's tables are modeled by two java ArrayList containing Strings representing the users and requests.

As we implemented two services, we used two different ports : 8089 for the add request and 8098 for the user.

NB : having one port per service is not particularly intuitive nor scalable.

The code of our services



```
package fr.insa.soap;
import java.util.ArrayList;
import java.util.List;
import javax.xml.ws.WebMethod;
import javax.xml.ws.WebParam;
import javax.xml.ws.WebService;

@WebService(serviceName="request")
public class Request {

    private List<String> requestList=new ArrayList<String>();

    @WebMethod(operationName="add")
    public boolean addRequest(@WebParam(name="request")String request) {
        return requestList.add(request);
    }

}
```

Figure 2: The code for our AddRequest service⁴

⁴ those beautiful PNGs of our code are made with carbon.now.sh

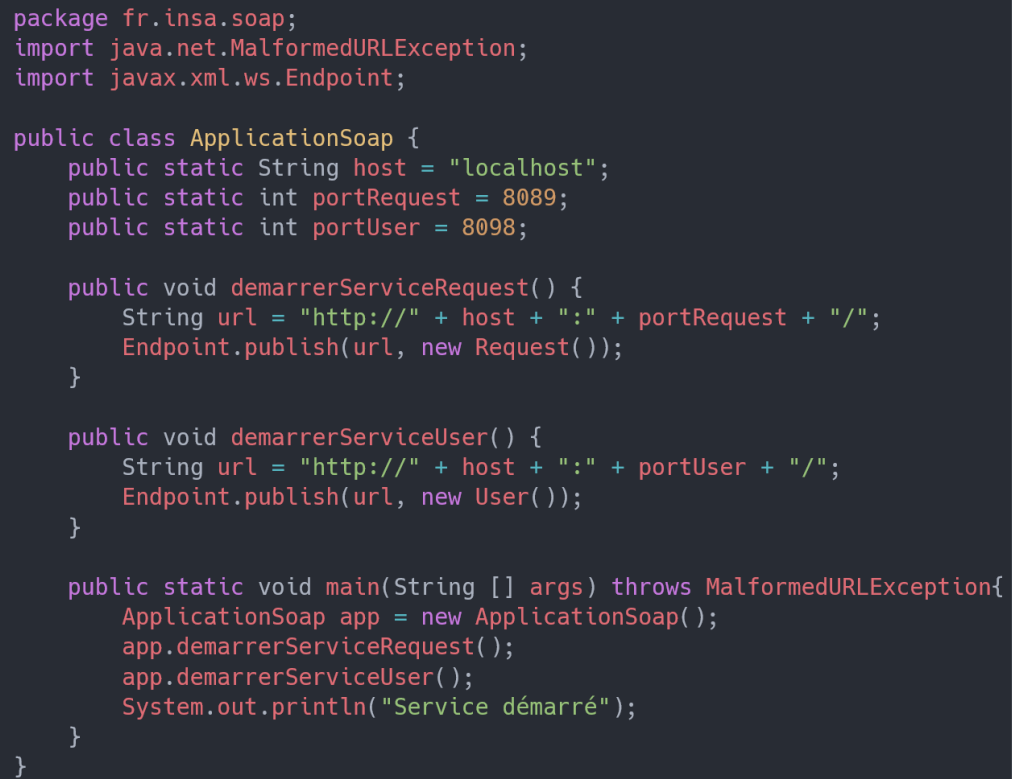


```
package fr.insa.soap;
import java.util.ArrayList;
import java.util.List;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService(serviceName="user")
public class User {
    List<String> listUser = new ArrayList<String>();

    @WebMethod(operationName="add")
    public boolean addUserToList(@WebParam(name="user")String user) {
        return listUser.add(user);
    }
}
```

Figure 3: The code for our AddUser service



```
package fr.insa.soap;
import java.net.MalformedURLException;
import javax.xml.ws.Endpoint;

public class ApplicationSoap {
    public static String host = "localhost";
    public static int portRequest = 8089;
    public static int portUser = 8098;

    public void demarrerServiceRequest() {
        String url = "http://" + host + ":" + portRequest + "/";
        Endpoint.publish(url, new Request());
    }

    public void demarrerServiceUser() {
        String url = "http://" + host + ":" + portUser + "/";
        Endpoint.publish(url, new User());
    }

    public static void main(String [] args) throws MalformedURLException{
        ApplicationSoap app = new ApplicationSoap();
        app.demarrerServiceRequest();
        app.demarrerServiceUser();
        System.out.println("Service démarré");
    }
}
```

Figure 4: The code of our application

The WSDL of our services

```
<!--
  Published by JAX-WS RI (https://github.com/eclipse-ee4j/metro-jax-ws). RI's version is JAX-WS RI 2.3.3 git-revision#b4c5bb6.
-->
<!--
  Generated by JAX-WS RI (https://github.com/eclipse-ee4j/metro-jax-ws). RI's version is JAX-WS RI 2.3.3 git-revision#b4c5bb6.
-->
<definitions targetNamespace="http://soap.insa.fr/" name="request">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://soap.insa.fr/" schemaLocation="http://localhost:8089/?xsd=1"/>
    </xsd:schema>
  </types>
  <message name="add">
    <part name="parameters" element="tns:add"/>
  </message>
  <message name="addResponse">
    <part name="parameters" element="tns:addResponse"/>
  </message>
  <portType name="Request">
    <operation name="add">
      <input wsam:Action="http://soap.insa.fr/Request/addRequest" message="tns:add"/>
      <output wsam:Action="http://soap.insa.fr/Request/addResponse" message="tns:addResponse"/>
    </operation>
  </portType>
  <binding name="RequestPortBinding" type="tns:Request">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="add">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="request">
    <port name="RequestPort" binding="tns:RequestPortBinding">
      <soap:address location="http://localhost:8089/">
    </port>
  </service>
</definitions>
```

Figure 5: The WSDL of our add request service⁵

⁵ I'm sorry, this screenshot is not very pretty

```

<!--
  Published by JAX-WS RI (https://github.com/eclipse-ee4j/metro-jax-ws). RI's version is JAX-WS RI 2.3.3 git-revision#b4c5bb6.
-->
<!--
  Generated by JAX-WS RI (https://github.com/eclipse-ee4j/metro-jax-ws). RI's version is JAX-WS RI 2.3.3 git-revision#b4c5bb6.
-->
<definitions targetNamespace="http://soap.insa.fr/" name="user">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://soap.insa.fr/" schemaLocation="http://localhost:8098/?xsd=1"/>
    </xsd:schema>
  </types>
  <message name="add">
    <part name="parameters" element="tns:add"/>
  </message>
  <message name="addResponse">
    <part name="parameters" element="tns:addResponse"/>
  </message>
  <portType name="User">
    <operation name="add">
      <input wsam:Action="http://soap.insa.fr/User/addRequest" message="tns:add"/>
      <output wsam:Action="http://soap.insa.fr/User/addResponse" message="tns:addResponse"/>
    </operation>
  </portType>
  <binding name="UserPortBinding" type="tns:User">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="add">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="user">
    <port name="UserPort" binding="tns:UserPortBinding">
      <soap:address location="http://localhost:8098/">
    </port>
  </service>
</definitions>

```

Figure 6: The WSDL of our add user service

Development of Restfull Services and REST API

1. Use the previously defined architecture

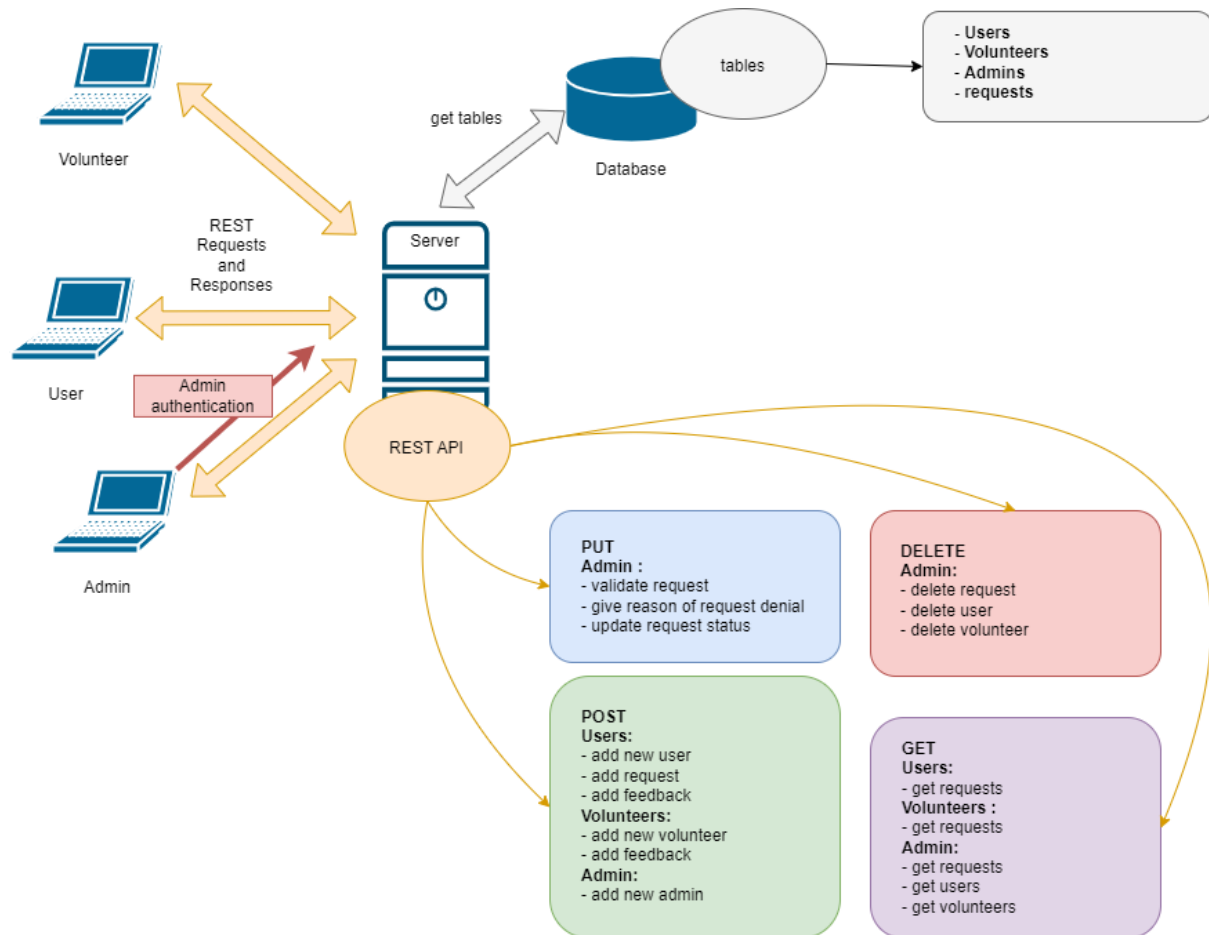


Figure 7: Our previous architecture modified according to REST ⁶

2. Migrate the developed SOAP services into RESTFull services

As previously, we implemented the *Add User* and *Add Request* services. As for the implementation of the SOAP services, we do not have a real database. However, REST services are stateless. We then cannot guard the information in Java Objects as we did for SOAP services. This avoids us getting the resources we just posted.

⁶ also made with draw.io, isn't it pretty?


```

1 package fr.insa.soa.TT;
2
3 public class User {
4     private String name;
5     private int age;
6     private String city;
7
8     public User(String name, int age, String city) {
9         this.name = name;
10        this.age = age;
11        this.city = city;
12    }
13    public User() {}
14
15    public void setName(String name) {
16        this.name = name;
17    }
18
19    public String getName() {
20        return this.name;
21    }
22
23    public void setAge(int age) {
24        this.age = age;
25    }
26
27    public int getAge() {
28        return this.age;
29    }
30
31    public void setCity(String city) {
32        this.city = city;
33    }
34
35    public String getCity() {
36        return this.city;
37    }
38
39    public String toString() {
40        return ("Nom: " + this.name + " Age: " + this.age + " City: " + this.city);
41    }

```

Figure 8: The code of a User⁷

⁷ sorry, we had no time to make these pretty with carbon.now.sh

```

1 package fr.insa.soa.TT;
2
3 public class Request {
4
5     private String name;
6     private String description;
7     private User creator;
8
9     public Request() {}
10
11     public void setName(String name) {
12         this.name = name;
13     }
14
15     public String getName() {
16         return this.name;
17     }
18
19     public void setDescription(String description) {
20         this.description = description;
21     }
22
23     public String getDescription() {
24         return this.description;
25     }
26
27     public void setCreator(User creator) {
28         this.creator = creator;
29     }
30
31     public String getCreator() {
32         return this.creator.toString();
33     }
34
35     public String toString() {
36         return ("Name : " + this.name + " Description : " + this.description + " Creator : (" + this.creator.toString() + ")");
37     }
38 }

```

Figure 9: Content of a Request

```

1 package fr.insa.soa.TT;
2
3 import jakarta.ws.rs.GET;
4 import jakarta.ws.rs.PUT;
5 import jakarta.ws.rs.POST;
6 import jakarta.ws.rs.Path;
7 import jakarta.ws.rs.PathParam;
8 import jakarta.ws.rs.Produces;
9 import jakarta.ws.rs.Consumes;
10 import jakarta.ws.rs.core.MediaType;
11
12 @Path("user")
13 public class Users {
14
15     @POST
16     @Path("/{idUser}")
17     @Consumes(MediaType.APPLICATION_JSON)
18     public String addUser(User user) {
19         System.out.println("User crée");
20         return("L'user " + user.toString() + " a bien été crée");
21     }
22 }

```

Figure 10: Java code of the POST User request

```

1 package fr.insa.soa.TT;
2
3 import jakarta.ws.rs.GET;
4 import jakarta.ws.rs.PUT;
5 import jakarta.ws.rs.POST;
6 import jakarta.ws.rs.Path;
7 import jakarta.ws.rs.PathParam;
8 import jakarta.ws.rs.Produces;
9 import jakarta.ws.rs.Consumes;
10 import jakarta.ws.rs.core.MediaType;
11
12 @Path("request")
13 public class Requests {
14
15     @POST
16     @Path("/{idRequest}")
17     @Consumes(MediaType.APPLICATION_JSON)
18     public String addRequest(Request request) {
19         System.out.println("Request crée");
20         return ("La requête d'id " + request.toString() + " a bien été créée");
21     }
22
23 }

```

Figure 11: Java code of the POST Request request

We used a postman client to access our API.

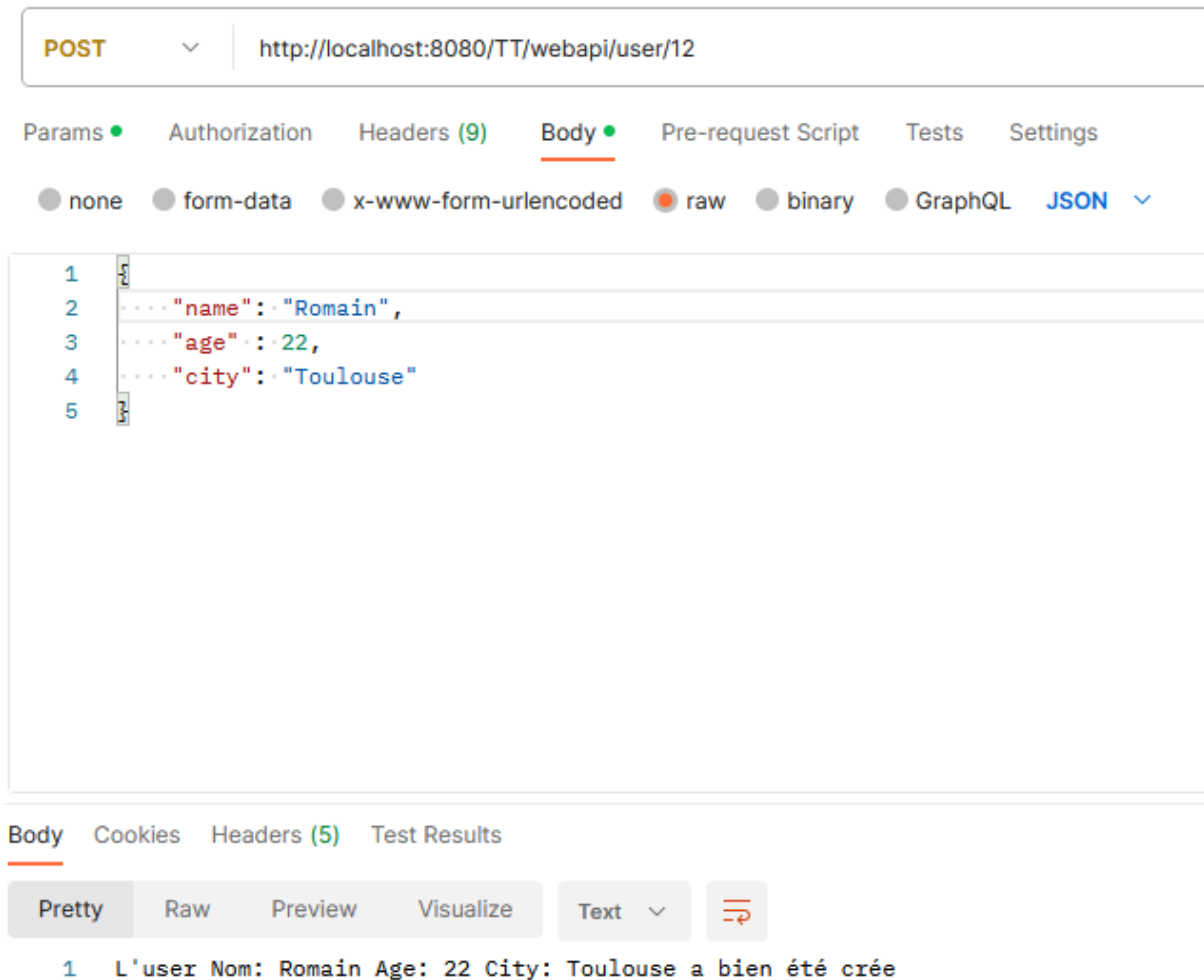


Figure 12: Result of a POST User request made with Postman

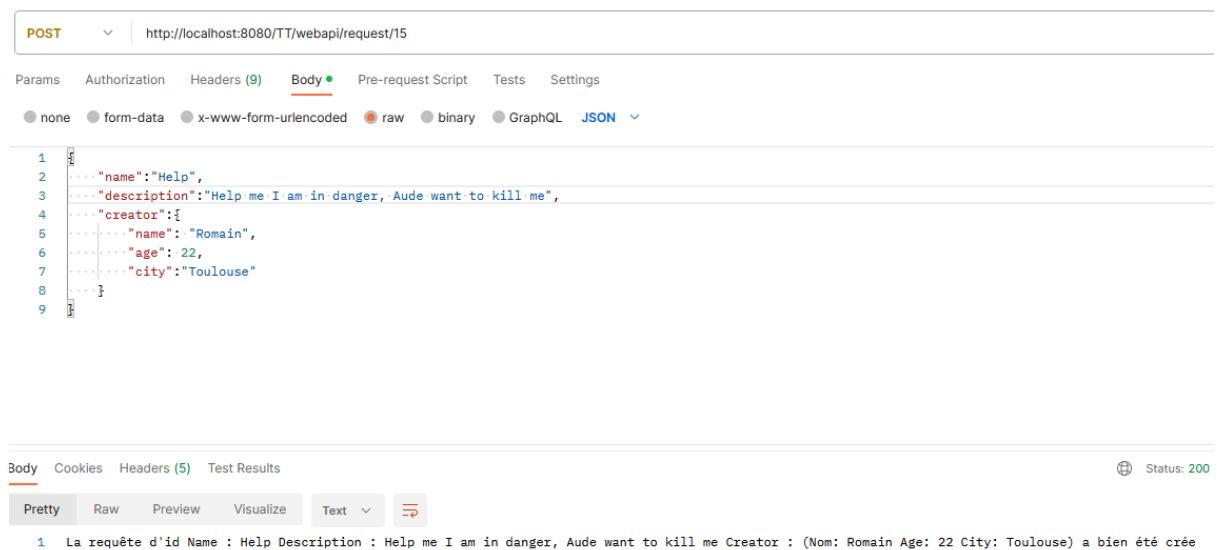


Figure 13: Result of a POST Request request made with Postman

3. Analyze and compare the two models (SOAP and REST)

The SOAP model is quite rigid. We need to implement each service one by one and execute each of them on its own port, which is not very practical. Moreover, on a machine, the number of ports is limited, and more open ports can mean more entries for attackers, and more work for the network administrator.

The REST model allows us to expose an API on a unique port for all the services. We noticed that the paradigm of the HTTP verbs (post, put, get, delete) is quite helpful to define the services. It is more intuitive than thinking by type of users like we did with SOAP, and there is less chance of forgetting something.

On the REST model, like on the SOAP model, we need to implement the services one by one.

Development of Microservices

1. Analyze the architecture you have designed. Are there limits?

The architecture we designed is still quite rigid. Updating the architecture would mean modifying lots of services to ensure smooth operation, and reusing the services implemented in another application would force long hours of debug and version compatibility checking.

In addition to that, we did not implement any mechanism to support a sudden rise of the number of users: if it was to exceed the capacities of the host server, there would be nothing to do to avoid a DoS (Deny of Service).

2. Does this architecture conform to a microservice architecture norm?

The previously (cf. Figure 4) defined architecture is not a microservice architecture. It lacks the granularity and decentralization of the microservices. It has not been designed with the idea of implementing reusable services which all have their own execution environment and responsibility.

3. Evolve your architecture toward a microservice one

We modified our previous architecture for a microservice-oriented one, with each microservice having its own responsibility.

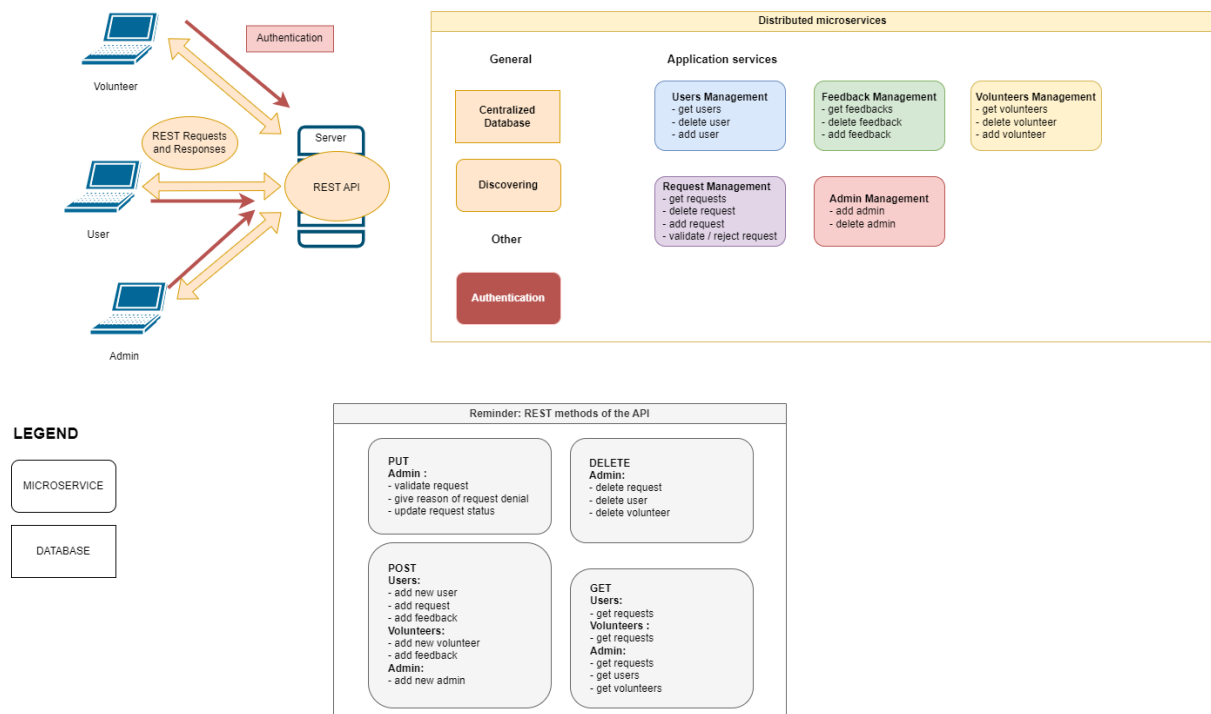


Figure 14: Our microservice architecture⁸

4. Implement this architecture

This time, we use a real database located on INSA servers.

You will find in the following figure the microservice architecture that we implemented. It is slightly different from the previous one for simplicity reasons.

⁸ still made with draw.io

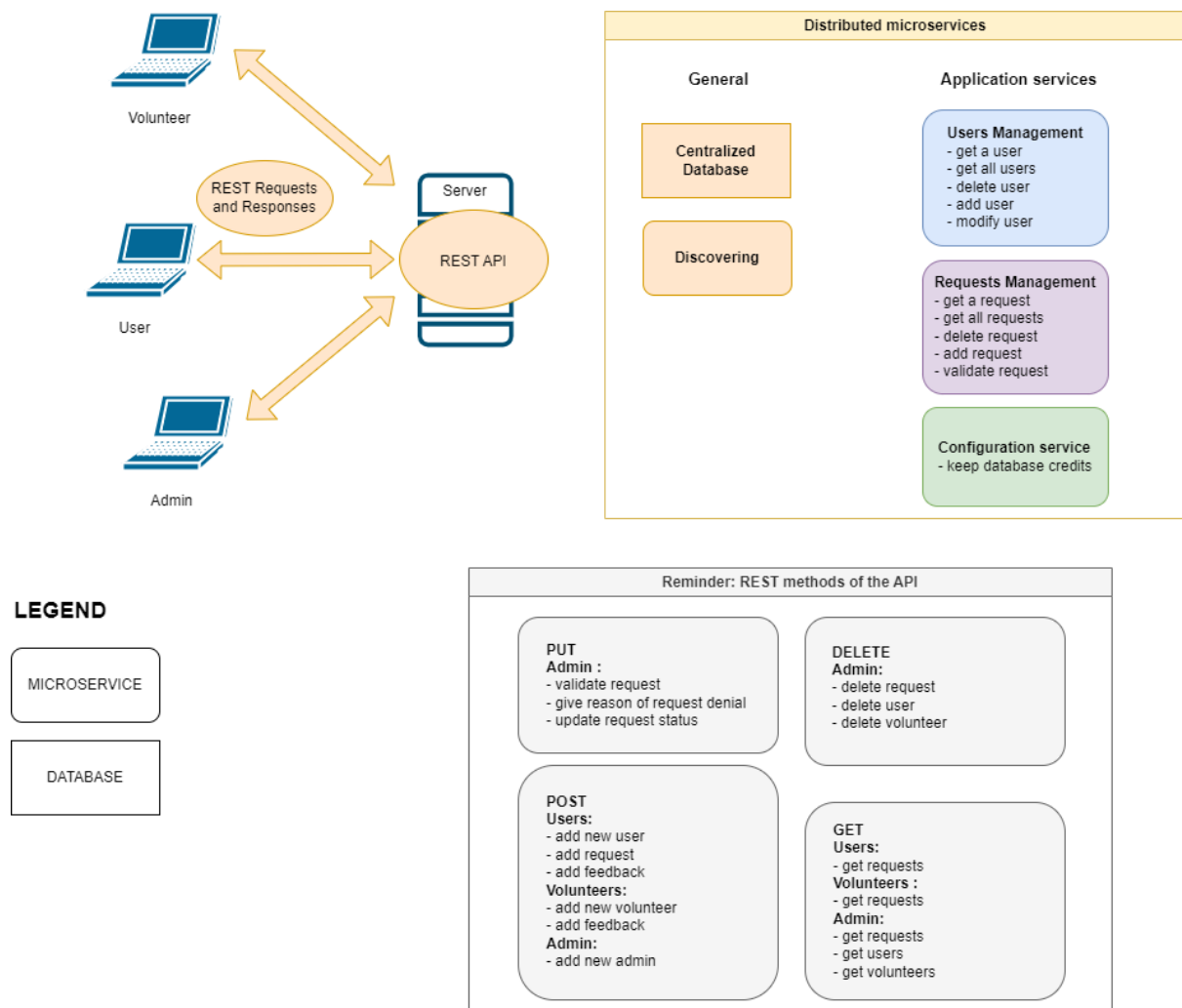


Figure 15: Our final microservice architecture (implemented)

We did not provide an authentication service for simplicity reasons. However, we did implemented some security with the time we had:

- the admin and the user are differentiated in the Database with an admin field at 1 for admin and 0 for user
- simple users can only delete their own requests
- simple user can delete and/or modify their profile, but not other users' profiles
- admin can modify and delete everyone's requests or user profile

When the Users Management or Requests Management need to access the database, they contact the Configuration Service through the Discovering Service. Then the Configuration Service provides them the credits of the Database so they can interact with it.

The Java code of this architecture is located on [this github repository](https://github.com/RomainMIn/ServiceArchitecture/).⁹

Once again, we used a postman client to interact with the Rest API exposed by our architecture.

⁹ <https://github.com/RomainMIn/ServiceArchitecture/>

POST Create User
GET Get all users
GET Get user 1
DEL Delete User
PUT Update User
GET Get request 1
GET Get all request
POST Create Request
DEL Delete Request
PUT Validate Request

Figure 16: Screenshot of the requests history: Example of possible requests

spring Eureka

HOME LAST 1000 SINCE STARTUP

System Status

Environment	N/A	Current time	2023-12-05T10:13:41 +0100
Data center	N/A	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	6
		Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONFIGURATIONSERVICE	n/a (1)	(1)	UP (1) - localhost:ConfigurationService:8082
REQUESTSERVICE	n/a (1)	(1)	UP (1) - localhost:RequestService:8081
USERSERVICE	n/a (1)	(1)	UP (1) - localhost:UserService:8083

Figure 17: Screenshot of the discovery service: microservices registered

GET

http://localhost:8083/user/5

Params

Authorization

Headers (6)

Body

Body

Cookies

Headers (5)

Test Results

Pretty

Raw

Preview

Visualize

1

2

3

4

5

"id": 5,

"name": "Test",

"city": "Lyon"

Figure 18: Screenshot of a request made with postman

Conclusion

Those directed labs allowed us to create various service oriented architecture, understanding the advantages and flaws of each one.