

TD Service Architecture

SOA – REST - Microservice

Cédric Chanfreau

Timothé Bigot

Promotion 58 – 2024 / 2025



Introduction.....	2
1. Architecture SOAP	3
a) Définition de SOA (Architecture orientée service).....	3
b) Choix de SOA pour le projet.....	3
c) Organisation des services	3
2. Architecture REST	5
a) Définition du REST	5
b) Structure du projet	5
3. Développement de l'application sous forme de micro-services	7
a) Création des micro-services avec Spring Boot	7
b) Nos micro-services.....	7
4. Automatisation d'un micro-service de l'application	13
Conclusion	14

Introduction

Le but de ces travaux dirigés est d'appliquer les concepts étudiés en cours concernant les architectures SOAP, REST et le développement de micro-services, à travers un projet concret. Ce projet consiste en la création d'une application de bénévolat, permettant à des bénévoles de répondre à des demandes d'aide formulées par des personnes dans le besoin. Plusieurs fonctionnalités ont été implémentées dans le cadre de ce projet, et ce compte rendu présente notre parcours de développement en détaillant les choix d'architectures, dans le but de mieux comprendre leurs avantages et limitations.

En consacrant un temps significatif à la réalisation des tutoriels, nous avons pu approfondir notre compréhension des concepts sous-jacents aux architectures SOAP et REST, et ainsi les mettre en œuvre efficacement dans notre application. L'ensemble des micro-services a été développé et testé, avec une attention particulière portée à l'automatisation via Microsoft Azure pour l'un des micro-services.



1. Architecture SOAP

a) Définition de SOA (Architecture orientée service)

Une architecture orientée services (SOA) se distingue par sa capacité à décomposer une application en composants autonomes, appelés « services », qui communiquent via des interfaces standardisées, généralement via des protocoles web comme SOAP (Simple Object Access Protocol). Chaque service dans une architecture SOA offre une fonctionnalité spécifique et peut être consommé indépendamment des autres services. Ces services sont souvent accessibles via des APIs, et les messages échangés sont typiquement formatés en XML, garantissant une grande interopérabilité entre différentes plateformes. L'utilisation de WSDL (Web Services Description Language) permet de décrire les services et leurs interfaces, facilitant leur utilisation et leur intégration par d'autres applications.

b) Choix de SOA pour le projet

Le service « User Management » s'occupe de la création, de la suppression et de la mise à jour des utilisateurs dans la base de données. Le service « Volunteer Management » gère les informations relatives aux bénévoles, telles que leurs compétences et disponibilités. Enfin, le service « Request Management » est responsable de la gestion des demandes d'assistance.

Cette organisation permet de rendre chaque service autonome et réutilisable. Par exemple, le service de gestion des bénévoles peut être utilisé par plusieurs autres applications sans dépendre directement de la gestion des utilisateurs. L'approche SOA favorise également la flexibilité : si une fonctionnalité doit être modifiée ou ajoutée, il suffit de mettre à jour le service correspondant.

c) Organisation des services

Dans le cadre d'une architecture SOAP, nous avons créé un service web à partir d'une classe java, puis un client, afin de pouvoir l'invoquer.

L'architecture SOAP adoptée pour le serveur de notre application d'aide aux personnes vulnérables a été testée et validée en utilisant le WSDL (Web Services Description Language). Le WSDL est un langage basé sur XML permettant de décrire l'interface d'un service web.

Nous avons créé une classe AnalyserChaine, ainsi qu'une classe AnalyserChaineApplication afin de pouvoir lancer le service web.

```

package fr.insa.soap;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService(serviceName="analyzer")
public class AnalyserChaineWS {
    @WebMethod(operationName="compare")
    public int analyser(@WebParam(name="chain") String chaine) {
        return chaine.length();
    }
}

package fr.insa.soap;

import java.net.MalformedURLException;
import javax.xml.ws.Endpoint;

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");
    }
}

```

Nous avons ensuite testé le code ci-dessus, à l'aide d'un web service explorer, et du fichier WSDL du service que l'on vient de développer, et qui contient toutes les informations importantes.

Une fois lancé, nous pouvons observer les requêtes SOAP envoyés et reçus et confirmer que le code fonctionne correctement, le bon nombre de caractère étant retourné.

SOAP Request Envelope:

```

<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:q0="http://soap.insa.fr/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <q0:compare>
      <chain>aaaaaa</chain>
    </q0:compare>
  </soapenv:Body>
</soapenv:Envelope>

```

SOAP Response Envelope:

```

<S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:compareResponse xmlns:ns2="http://soap.insa.fr/">
      <return>6</return>
    </ns2:compareResponse>
  </S:Body>
</S:Envelope>

```

2. Architecture REST

a) Définition du REST

L'API REST (Representational State Transfer) a été choisie pour permettre la communication entre les services et pour exposer leurs fonctionnalités aux clients. REST est un style architectural qui s'appuie sur les méthodes HTTP standards (« GET », « POST », « PUT », « DELETE ») pour manipuler des ressources. Dans ce projet, chaque ressource correspond à un élément essentiel : utilisateurs, bénévoles ou requêtes.

b) Structure du projet

Sur la partie REST, nous nous sommes principalement concentrés sur la création d'un service web Rest à partir d'une classe Java et de l'invoquer ensuite avec Postman ou depuis l'interface graphique.

- Pour le service « **UserManagement** », les fonctionnalités suivantes ont été implémentées :

- GET /api/users : récupération de tous les utilisateurs.
- POST /api/users : ajout d'un utilisateur.
- DELETE /api/users/{id} : suppression d'un utilisateur par ID.

- Pour le service « **userRequestService** », les fonctionnalités suivantes ont été implémentées :

- GET /api/requests : Récupération de toutes les requêtes existantes, permettant un suivi centralisé des demandes.
- POST /api/requests : Création d'une nouvelle requête en associant un utilisateur et en décrivant la nature de la demande.
- PUT /api/requests/{id}/status : Mise à jour du statut d'une requête (par exemple : en attente, validée, rejetée, complétée).

- Pour le service « **VolunteerManagement** », les fonctionnalités suivantes ont été implémentées :

- GET /api/volunteers : Récupération de la liste de tous les bénévoles enregistrés, avec leurs informations et disponibilités.
- POST /api/volunteers : Enregistrement d'un nouveau bénévole en saisissant ses compétences, ses disponibilités, ainsi que ses informations personnelles (prénom et nom).
- PUT /api/volunteers/{id} : Mise à jour des informations d'un bénévole existant, notamment ses compétences ou sa disponibilité.

- Pour le service « **feedbackService** », les fonctionnalités suivantes ont été implémentées :

- `POST /api/feedbacks` : Soumission d'un nouvel avis, avec un commentaire et une note d'évaluation.
- `DELETE /api/feedbacks/{id}` : Suppression d'un retour d'expérience spécifique en fonction de son identifiant.

Pour chaque service, l'implémentation REST garantit une simplicité d'intégration. Les clients (comme le frontend en JavaScript) peuvent interagir avec le backend, en envoyant des requêtes HTTP standard et en recevant des réponses formatées en JSON.

A titre de comparaison, l'architecture SOAP possède une structure assez rigide, basée sur XML mais permet d'être plus formel. Cependant elle reste tout de même complexe et est peu flexible et ainsi remettre en cause sa facilité d'utilisation.

REST, en se basant sur des standards tel que JSON et HTTP permet d'être plus flexible avec une interaction des ressources et des requêtes plus intuitives. C'est un choix d'architecture plus adapté pour des systèmes distribués avec une rapidité et une scalabilité plus importante qu'avec SOAP.

Nous allons maintenant nous concentrer sur le développement de cette application d'offre et de demande de bénévolat mais d'un point de vue micro-services, qui est une forme hybride des deux architectures précédentes, mais basé principalement sur la vision ressources de REST.

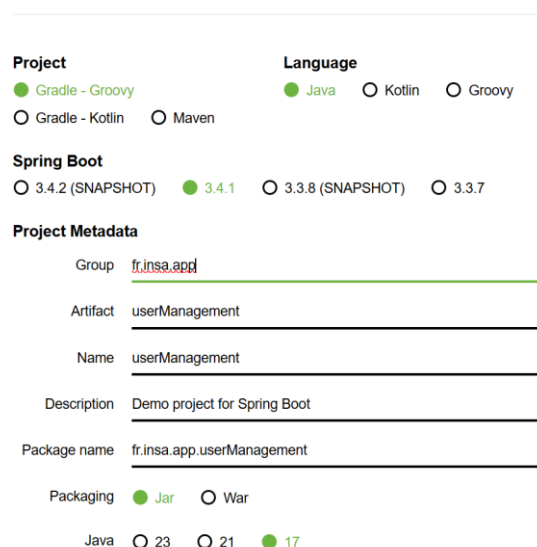
3. Développement de l'application sous forme de micro-services

Les micro-services constituent une évolution de notre projet. Ils consistent à décomposer une application en services autonomes et déployables indépendamment. Dans ce projet, chaque service (« User Management », « Volunteer Management », « Request Management », « Feedback Management ») est implémenté en tant que micro-service isolé.

Chaque micro-service dispose de sa propre base, de son propre cycle de développement et de déploiement. Par exemple, le service « User Management » est déployé sur le port 8090, tandis que « Volunteer Management » est accessible sur le port 8092. Les micro-services communiquent entre eux via des requêtes HTTP RESTful. Ainsi, le service « Request Management » peut récupérer des utilisateurs en interrogeant l'API du service « User Management ».

a) Création des micro-services avec Spring Boot

Tout commence par une première version utilisant Spring Boot avec une unique dépendance liée au service web. Un aspect important de cette architecture est l'utilisation de Spring Boot, qui facilite la création de micro-services.



The image shows a screenshot of the Spring Boot project configuration form. It includes sections for Project, Language, Spring Boot, Project Metadata, and Packaging. The Project section has radio buttons for Gradle - Groovy, Gradle - Kotlin, and Maven. The Language section has radio buttons for Java, Kotlin, and Groovy. The Spring Boot section has radio buttons for 3.4.2 (SNAPSHOT), 3.4.1, 3.3.8 (SNAPSHOT), and 3.3.7. The Project Metadata section includes fields for Group (fr.insa.app), Artifact (userManagement), Name (userManagement), Description (Demo project for Spring Boot), and Package name (fr.insa.app.userManagement). The Packaging section has radio buttons for Jar and War. The Java section has radio buttons for 23, 21, and 17.

Project	Language
☒ Gradle - Groovy	☒ Java ☐ Kotlin ☐ Groovy
☐ Gradle - Kotlin ☐ Maven	
Spring Boot	
☐ 3.4.2 (SNAPSHOT) ☒ 3.4.1 ☐ 3.3.8 (SNAPSHOT) ☐ 3.3.7	
Project Metadata	
Group	<u>fr.insa.app</u>
Artifact	<u>userManagement</u>
Name	<u>userManagement</u>
Description	<u>Demo project for Spring Boot</u>
Package name	<u>fr.insa.app.userManagement</u>
Packaging	☒ Jar ☐ War
Java	☐ 23 ☐ 21 ☒ 17

b) Nos micro-services

Chaque micro-service est configuré de manière indépendante dans un fichier application.properties :

- userManagement:

Ce premier service a pour but d'ajouter un utilisateur et de lui attribuer un statut USER, VOLUNTEER ou ADMIN à l'adresse localhost:8090.

The top part of the image shows an IDE view of the `userRequestService` package structure:

- `userRequestService` [APP master]
 - `src/main/java`
 - `fr.insa.app.userRequestService`
 - `Request.java`
 - `RequestController.java`
 - `RequestDto.java`
 - `RequestService.java`
 - `RequestStatus.java`
 - `UserRequestServiceApplication.java`

The bottom part shows a Postman client interface for a POST request to `http://localhost:8090/users`. The request body is a JSON object:

```

{
  "firstName": "Cedric",
  "lastName": "Chanfreau",
  "role": "admin"
}

```

The response body is also shown as JSON:

```

{
  "id": 1,
  "firstName": "Cedric",
  "lastName": "Chanfreau",
  "role": "admin"
}

```

Dans un premier temps la base de données, n'étant pas fonctionnelle nous avons lancé nos services sans qu'il y ait d'enregistrement ou lecture de table. Par la suite nous avons ajouté le lien avec la base.

The top part of the image shows a Postman client interface for a POST request to `http://localhost:8090/api/users`. The request body is a JSON object:

```

{
  "firstName": "Cédric",
  "lastName": "Chanfreau",
  "role": "admin"
}

```

The bottom part shows a MySQL terminal window with the following queries and results:

```

mysql> SELECT * FROM User;
+----+-----+-----+-----+
| id | first_name | last_name | role |
+----+-----+-----+-----+
| 1 | John | Doe | volunteer |
| 2 | Cedric | Chanfreau | admin |
+----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> SELECT * FROM User;
+----+-----+-----+-----+
| id | first_name | last_name | role |
+----+-----+-----+-----+
| 1 | John | Doe | volunteer |
| 2 | Cedric | Chanfreau | admin |
+----+-----+-----+-----+
2 rows in set (0.00 sec)

```

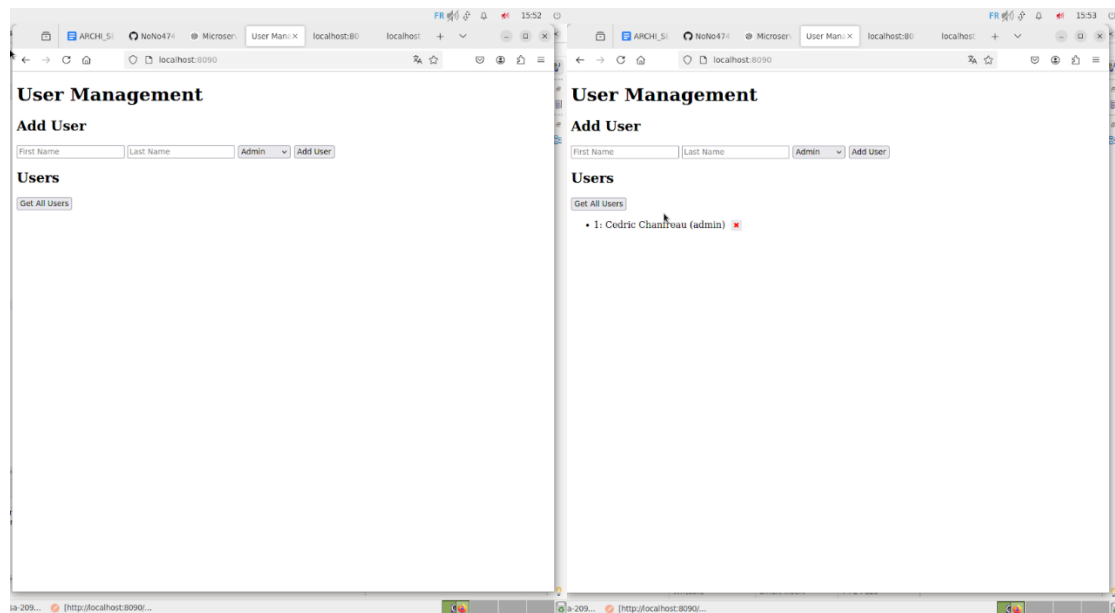
The bottom part of the image shows a terminal window with the following commands:

```

hibernate: insert into user (first_name,last name,role) values (?,?,?)
hibernate: insert into user (first_name,last name,role) values (?,?,?)
hibernate: insert into user (first_name,last name,role) values (?,?,?)
hibernate: select u1_0.id,u1_0.first_name,u1_0.last_name,u1_0.role from user u1_0

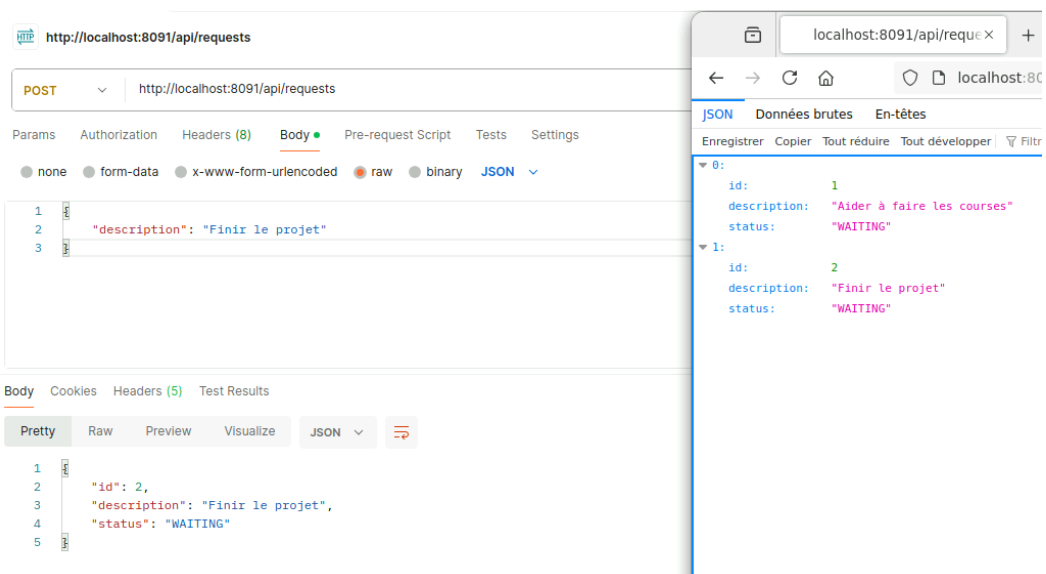
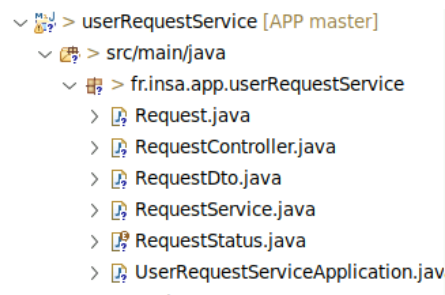
```

Pour finir, nous avons ajouté de l'HTML, CSS et JavaScript pour faire en sorte que l'application soit davantage interactive.

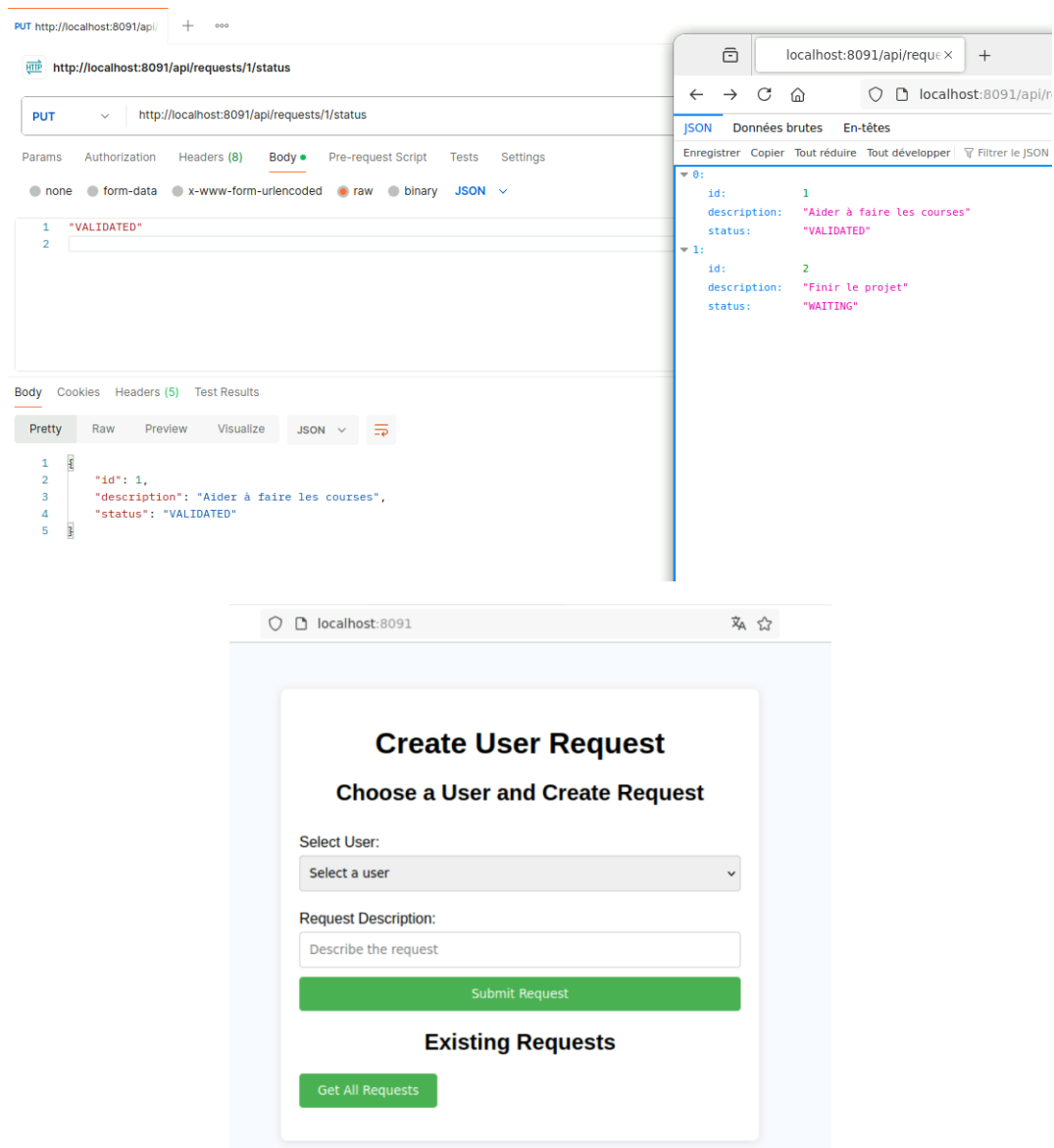


- userRequestService:

Ce micro-service permet d'associer à un utilisateur une requête. Une fois la demande effectuée il y a le statut qui peut être modifier pour faire en sorte de connaitre l'avancer.



Changement du status de la requête numéro 1 :



Depuis cette interface, il est possible de sélectionner un utilisateur présent dans la base de données et de lui envoyer une requête. Le bouton submit est associé à la requête POST et get all user à GET.

- volunteerManagement :

Pour ce micro-service, il est possible pour un volontaire d'entrer ses compétences et ses disponibilités. Le but étant qu'un user ait accès aux volontaires disponible pour le contacter.

- > volunteerManagement [APP master]
 - > src/main/java
 - > fr.insa.app.volunteerManagement
 - > Feedback.java
 - > Volunteer.java
 - > VolunteerDto.java
 - > VolunteerManagementApplication.j
 - > VolunteerRessource.java

POST http://localhost:8092/vol

http://localhost:8092/volunteers

POST http://localhost:8092/volunteers

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary JSON

```

1 {
2   "firstName": "Paul",
3   "lastName": "Louise",
4   "skills": "Maintenance PK INSA",
5   "availability": "Lundi, Mardi, Mercredi"
6 }
7

```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```

1 {
2   "id": 1,
3   "firstName": "Paul",
4   "lastName": "Louise",
5   "skills": "Maintenance PK INSA",
6   "availability": "Lundi, Mardi, Mercredi",
7   "active": true
8 }

```

localhost:8090/users localhost:8091/api/req localhost:8092/volunte

localhost:8092/volunteers

JSON Données brutes En-têtes

Enregistrer Copier Tout réduire Tout développer Filtre le JSON

```

0:
  id: 1
  firstName: "Paul"
  lastName: "Louise"
  skills: "Maintenance PK INSA"
  availability: "Lundi, Mardi, Mercredi"
  active: true

```

PUT http://localhost:8092/vol

http://localhost:8092/volunteers/2/deactivate

PUT http://localhost:8092/volunteers/2/deactivate

Params Authorization Headers (7) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary JSON

```

1

```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```

1 {
2   "id": 2,
3   "firstName": "Jena",
4   "lastName": "Duj",
5   "skills": "Appeler",
6   "availability": "Samedi",
7   "active": false
8 }

```

localhost:8090 localhost:8091 localhost:8092

localhost:8092/volunteers

JSON Données brutes En-têtes

Enregistrer Copier Tout réduire Tout développer Filtre le JSON

```

0:
  id: 1
  firstName: "Paul"
  lastName: "Louise"
  skills: "Maintenance PK INSA"
  availability: "Lundi, Mardi, Mercredi"
  active: false
1:
  id: 2
  firstName: "Jena"
  lastName: "Duj"
  skills: "Appeler"
  availability: "Samedi"
  active: true

```

feedbackService:

The image shows two screenshots. The top screenshot is from an IDE showing the project structure for 'feedbackService [APP master]'. The structure is as follows:

- feedbackService [APP master]
 - src/main/java
 - fr.insa.app.feedbackService
 - Feedback.java
 - FeedbackDto.java
 - FeedbackRessource.java
 - FeedbackServiceApplication.java

The bottom screenshot shows a web browser and a REST client. The browser address bar shows 'localhost:8094/feedbacks'. The REST client shows a POST request to 'http://localhost:8094/feedbacks?volunteerId=1&userId=2' with a JSON body:

```
{
  "comments": "Très bonne expérience, le bénévolat a été enrichissant.",
  "rating": 5
}
```

The REST client also shows the response in JSON format:

```
{
  "volunteerId": 1,
  "userId": 2,
  "comments": "Très bonne expérience, le bénévolat a été enrichissant.",
  "rating": 5
}
```

L'implémentation de ce service permet de récupérer les requêtes validées et donner la possibilité à l'utilisateur de faire un feedback sur le travail effectué.

4. Automatisation d'un micro-service de l'application

Une fois l'architecture logicielle distribuée, basée sur l'interaction de plusieurs microservices, mise en place, nous avons implémenté l'intégration continue (automatisation de la construction et des empaquetages) ainsi que le déploiement continu. Pour cela, nous avons utilisé Microsoft Azure comme plateforme de développement et Github Actions pour gérer l'intégration et le déploiement. Nous avons choisi de travailler sur le microservice **userManagement**.

Après avoir créé un compte Microsoft Azure, nous avons créé un groupe de ressources nommé **MSDeployment-group**, ainsi qu'une application web avec l'instance **userManagementMS-Instance**.

Créer une application web

Déployez et générez du code à partir de votre fournisseur de build et source par défaut.

Détails du projet Sélectionnez un abonnement pour gérer les coûts et les ressources déployées. Utilisez les groupes de ressources comme des dossiers pour organiser et gérer toutes vos ressources.	Source GitHub Se déconnecter
Abonnement * Azure for Students	Connecté comme CedricChnfr
Groupe de ressources * MsDeployment-group Créer nouveau	Organisation CedricChnfr
Détails de l'instance Nom userManagementMS .azurewebsites.net	Dépôt service-architecture-insa
Publier * Code ☒ Conteneur ☐	Branche master
Pile d'exécution * Java 11	Build
Pile serveur web Java * Java SE (Embedded Web Server)	Fournisseur de build GitHub Actions
Système d'exploitation * Linux ☒ Windows ☐	Pile d'exécution Java
Région * East US	Version Java 11

Plans de tarification
Le niveau tarifaire du plan App Service détermine l'emplacement, les fonctionnalités, le coût et les ressources de calcul associés à votre application. [En savoir plus](#)

Plan Linux (East US) *
ASP-MSDeploymentGroup-9044 (B1)
[Créer](#)

Plan de tarification
De base B1 (100 ACU au total, 1,75 Go de mémoire, 1 processeur virtuel)

Redondance de zone
Vous pouvez déployer un plan App Service en tant que service redondant interzone dans des régions le prenant en charge. Vous devez décider de cela au moment du déploiement car vous ne pouvez plus rendre une zone plan App Service redondante après son déploiement. [En savoir plus](#)

Ensuite, nous avons généré un fichier de script YAML dans le dépôt GitHub pour automatiser le processus. Une dernière configuration du fichier était nécessaire pour associer le nom du micro-service dans le script du workflow. Une fois cette étape terminée et le workflow compilé, le déploiement était complet. En

effectuant une requête GET sur ce micro-service, le workflow se déclenchait comme prévu.

Conclusion

En somme, ces travaux dirigés nous ont permis de découvrir et de confronter différentes architectures de services, en mettant en lumière leurs avantages et leurs limitations. Cette première implémentation pratique constitue une base qui nous sera particulièrement utile pour les prochains travaux pratiques.

Nous espérons pouvoir étendre les fonctionnalités, notamment en explorant des aspects tels que la découvrabilité des micro-services, bien que ce sujet ait été abordé en tutoriel mais non intégré au projet. Nous envisageons également d'ajouter des services de configuration ou de répartition de charge, en complément des micro-services et de l'automatisation via Azure.

De plus, ces travaux ont renforcé notre compréhension de la vision décentralisée des micro-services et de leur capacité à offrir une évolutivité et une maintenabilité supérieures par rapport à une approche centralisée, comme celle de oneM2M abordée lors du TP de "Middleware for IoT". Cette expérience nous a permis de mesurer l'impact de l'architecture décentralisée sur la performance et la flexibilité du système, des points essentiels pour les projets futurs.

<https://github.com/CedricChnfr/service-architecture-insa>