

Architecture de Service

Kenza Bouzergan
Lamiaa Housni
5 ISS

20/12/2023

Introduction :

Ce rapport technique explore l'évolution de l'architecture de l'application Volunteering, en mettant en lumière les architectures SOAP, REST et Microservices. L'application vise à connecter les personnes nécessitant de l'aide avec des volontaires prêts à offrir leur soutien en implémentant des fonctionnalités comme : valider des demandes, donner des retours, etc...

I. Architecture SOAP

SOAP (Simple Object Access Protocol) est un protocole de communication basé sur XML utilisé pour échanger des informations entre applications. Il offre une structure formelle pour les messages, généralement transportés via des protocoles tels que HTTP, facilitant ainsi l'interopérabilité entre différentes plates-formes.

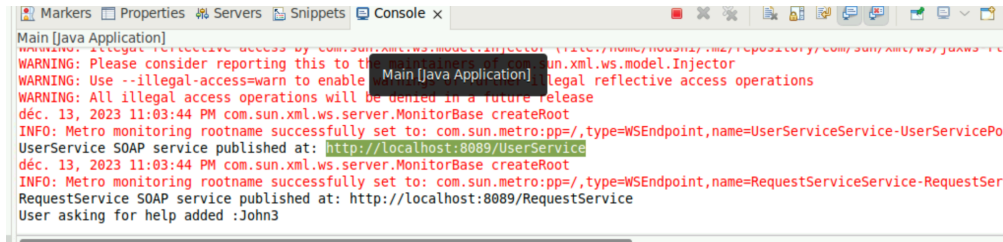
Concernant notre architecture, nous avons créé 5 classes :

- **User** : la classe User fournit une structure de base pour représenter un utilisateur, avec la possibilité de spécifier et de récupérer le nom et l'identifiant associés à cet utilisateur. Cette classe peut être utilisée comme une classe de base à partir de laquelle d'autres classes, comme la classe Admin, peuvent être dérivées pour inclure des fonctionnalités supplémentaires spécifiques.
- **UserService** : la classe UserService offre une interface de service web pour l'ajout d'utilisateurs demandant de l'aide, de bénévoles et d'administrateurs dans le cadre d'une application de bénévolat. Elle utilise des méthodes simples pour afficher des informations dans la console et renvoie les identifiants des utilisateurs ajoutés en tant que confirmation.
- **Admin** : la classe Admin encapsule les caractéristiques spécifiques d'un administrateur, en ajoutant la notion de clé d'administration à celles d'un utilisateur générique. Cette classe peut être utilisée pour créer des instances d'administrateurs et les intégrer dans le système global de l'application de bénévolat.
- **Request** : la classe Request encapsule les informations liées à une demande dans l'application, permettant ainsi de gérer et de suivre l'évolution de chaque requête. Les différents statuts possibles d'une demande sont : "WAITING," "VALIDATED," "REJECTED," "CHOSEN," et "REALIZED."
- **RequestService** : la classe RequestService offre des fonctionnalités pour la gestion des demandes.

Test de notre architecture via le Web Explorer :

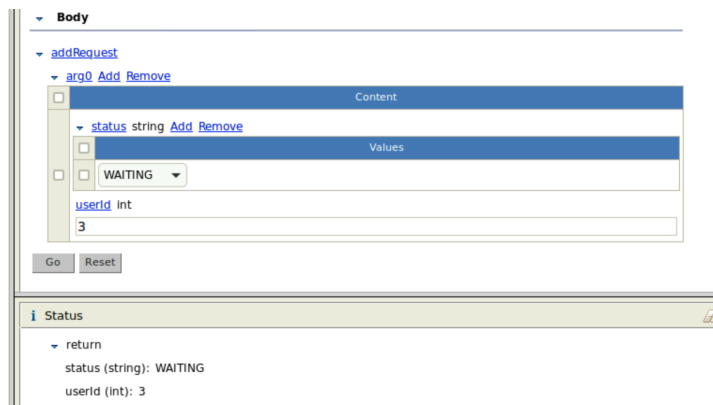
On peut voir que deux services ont été publiés :

- UserService à l'adresse <http://localhost:8809/UserService>
- RequestService à l'adresse <http://localhost:8809/RequestService>



```
Main [Java Application]
WARNING: Please consider reporting this to the vendor, since a future release will be required to support the requested reflective access operation
WARNING: Use --illegal-access=warn to enable warnings of illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
déc. 13, 2023 11:03:44 PM com.sun.xml.ws.server.MonitorBase createRoot
INFO: Metro monitoring rootname successfully set to: com.sun.metro:pp=//, type=WEEndpoint, name=UserService-UserServicePor
UserService SOAP service published at: http://localhost:8809/UserService
déc. 13, 2023 11:03:44 PM com.sun.xml.ws.server.MonitorBase createRoot
INFO: Metro monitoring rootname successfully set to: com.sun.metro:pp=//, type=WEEndpoint, name=RequestService-RequestServ
RequestService SOAP service published at: http://localhost:8809/RequestService
User asking for help added :John3
```

⇒ En ajoutant les arguments depuis le Web Explorer on remarque l'ajout d'un user John avec l'identifiant 3.



Body

addRequest

arg0 Add Remove

Content

status string Add Remove

Values

WAITING

userid int

3

Go Reset

Status

return

status (string): WAITING

userid (int): 3

⇒ ajout de "request" : OK

II. Architecture REST

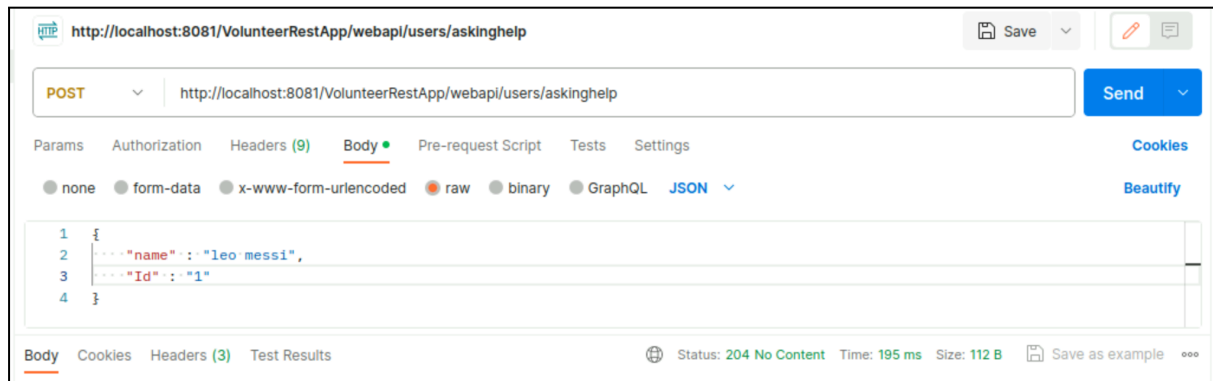
L'architecture REST (Representational State Transfer) est un style architectural pour les systèmes web. Reposant sur des principes simples, REST utilise des URI pour représenter et adresser des ressources, avec des opérations standard comme GET, POST, PUT, et DELETE effectuées via des requêtes HTTP. Son utilisation de formats de représentation standard comme JSON ou XML assure une interopérabilité efficace entre les systèmes, faisant de REST une approche privilégiée pour le développement d'API web robustes.

Nous avons entrepris une migration en passant de notre ancienne architecture basée sur SOAP qui est complexe due à XML à une architecture REST plus flexible. En adoptant cette architecture, nous avons redéfini la façon dont nos services communiquent en utilisant des URIs pour identifier et accéder à des ressources spécifiques. Les opérations CRUD (Create, Read, Update, Delete) sont désormais alignées sur les verbes HTTP appropriés, simplifiant ainsi l'interaction entre les clients et le serveur. De plus, l'utilisation de formats de représentation standard tels que JSON facilite l'échange de données entre les différents composants de notre application. Cette migration vers une architecture REST a non seulement modernisé notre système, mais elle a également renforcé sa scalabilité et son adaptabilité pour répondre aux besoins changeants de notre environnement technologique.

Les classes de notre architecture restent les mêmes mais les annotations diffèrent (on ajoute des POST, GET, PUT ainsi que des Path(/user) par exemple)

Test de notre architecture :

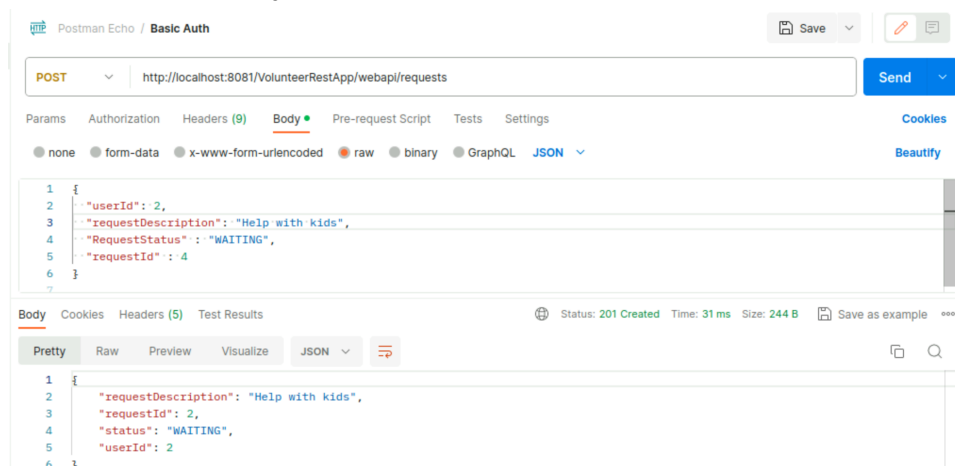
Requête POST qui ajoute un aidSeeker depuis PostMan :



```
INFO: Au moins un fichier JAR a été analysé pour trouver des TLDs mais il n'en contenait pas, le mode "debug" du journal peut  
déc. 13, 2023 10:59:20 PM org.apache.catalina.core.StandardContext reload  
INFO: Le rechargement de ce contexte est terminé  
User asking for help added: leo messi0
```

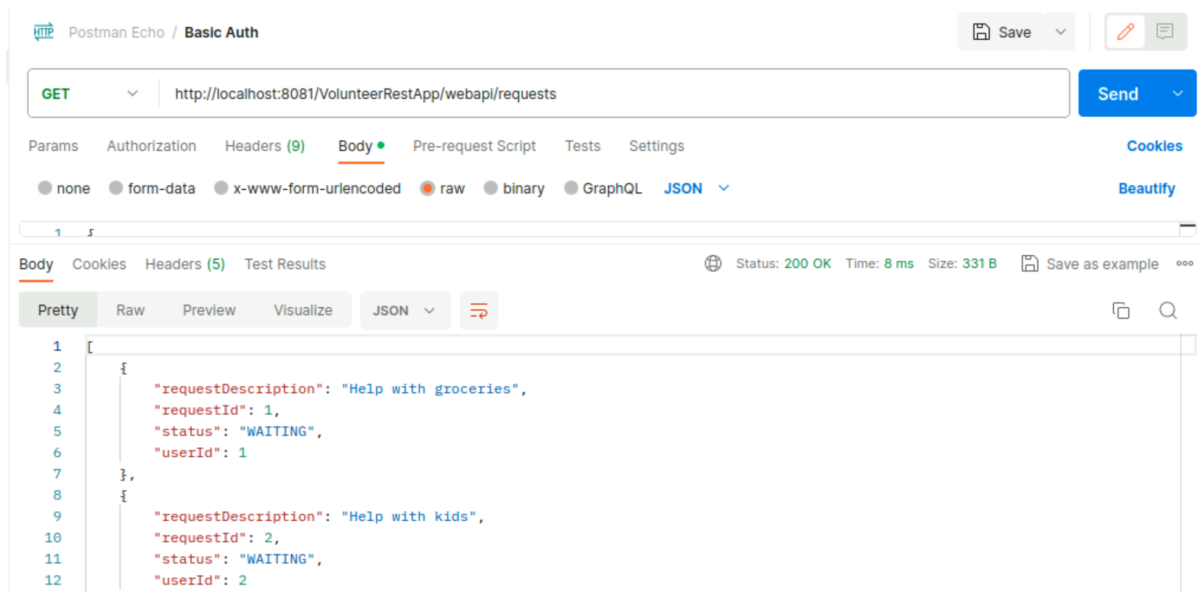
⇒ OK

Requête POST qui ajoute une “request” :



⇒ OK

Requête GET qui retourne les “request” :



⇒ OK

III. Architecture Microservice

Jusqu'ici on a utilisé une architecture REST qui manipule des requêtes HTTP. Or, cette architecture a des limites, notamment une complexité accrue dans la communication entre les ressources. Afin de dépasser ces limites, notre architecture va évoluer vers une architecture de microservices.

Un microservice est une architecture logicielle qui divise une application en plusieurs services autonomes et indépendants, appelés microservices. Chaque microservice est responsable de tâches spécifiques et communique avec les autres microservices via des API bien définies. Cette approche favorise la flexibilité, la scalabilité et la maintenance simplifiée des applications.

- **Caractéristiques des microservices :**

Les caractéristiques des microservices incluent:

- Indépendance et autonomie : Chaque microservice peut être développé, déployé et mis à l'échelle indépendamment.
- Déploiement Facilité : Les microservices peuvent être déployés de manière indépendante, facilitant ainsi les mises à jour et les corrections.
- Technologies variées : Chaque microservice peut être développé en utilisant des technologies différentes adaptées à sa tâche spécifique.
- Résilience et tolérance aux Pannes : Les défaillances dans un microservice n'affectent pas nécessairement l'ensemble du système.

- **Travail effectué :**
 - a. **Collaboration des 3 microservices**

Nous avons créé 3 microservices :

- **UserListService** : Le rôle de ce microservice est d'obtenir la liste complète des utilisateurs selon leur rôle (admin, volunteer, aidseeker). Ce microservice a comme entrée le rôle (e.g., admin) et retourne une liste des utilisateurs(id) des étudiants.
- **UserManagement** : Ce microservice permet de récupérer les informations d'un user donné. Ce microservice a comme entrée un id et retourne le nom.
- **RequestManagement** : Ce microservice a comme entrée un id et retourne la description de la requête faite par un utilisateur.(aidseeker)

La première étape est de générer 3 projets Spring boot pour les différents microservices.

Ce qu'on cherche:

1. Le Microservice *UserListResource* récupère la liste des id des utilisateurs selon le rôle.
2. Pour chaque id, il sollicite le microservice *UserManagement* pour avoir les informations de l'utilisateur.
3. Aussi, il sollicite le microservice *RequestController* pour avoir la description de la demande correspondante à l'utilisateur.
4. Finalement, il retourne toutes les informations récupérées pour chaque utilisateur (ses informations et sa demande)

Test de la collaboration des différents microcontrôleurs:

La BDD simulée est :

- Pour les utilisateurs :

```
List<User> userInfos= Arrays.asList(  
    new User( id: 0, name: "Ahlam"),  
    new User( id: 1, name: "Theo"),  
    new User( id: 2, name: "Laura"),  
    new User( id: 3, name: "Mattia")  
);
```

- Pour les requêtes :

```
List<Request> requestList= Arrays.asList(
    new Request( requestId: 0, userId: 0, requestDescription: "help with groceries",WAITING),
    new Request( requestId: 1, userId: 1, requestDescription: "help with kids",VALIDATED),
    new Request( requestId: 3, userId: 2, requestDescription: "help with studies",REJECTED),
    new Request( requestId: 3, userId: 3, requestDescription: "help moving out",WAITING)
);
```

- Pour simuler les rôles : (Ici, pour le test un utilisateur peut avoir à la fois 2 rôles)

```
private List<Integer> getUserIdsByRole(String role) {

    if ("admin".equalsIgnoreCase(role)) {
        return List.of(0, 1, 2);
    } else if ("aidSeeker".equalsIgnoreCase(role)) {
        return List.of(1, 3);
    } else if ("volunteer".equalsIgnoreCase(role)) {
        return List.of(0, 2);
    } else {
        return List.of();
    }
}
```

Le résultat est :

- Pour les admins :

```
[
  {
    id: 0,
    name: "Ahlam",
    requestDescription: "help with groceries"
  },
  {
    id: 1,
    name: "Theo",
    requestDescription: "help with kids"
  },
  {
    id: 2,
    name: "Laura",
    requestDescription: "help with studies"
  }
]
```

- Pour les volunteers :



```
[
  {
    id: 0,
    name: "Ahlam",
    requestDescription: "help with groceries"
  },
  {
    id: 2,
    name: "Laura",
    requestDescription: "help with studies"
  }
]
```

- Pour aidSeeker :



```
[
  {
    id: 1,
    name: "Theo",
    requestDescription: "help with kids"
  },
  {
    id: 3,
    name: "Mattia",
    requestDescription: "help moving out"
  }
]
```


On conclut que les 3 microservices collaborent correctement.

b. Découverte des microservices :

Dans cette partie, le but est de développer des microservices découvrables en utilisant *Spring Cloud*.

La première étape est de générer un projet Spring Boot en ajoutant les dépendances du serveur Eureka.

Après avoir un serveur prêt à découvrir et enregistrer les microservices il faut publier les 3 microservices qu'on a précédemment développés.


HOME LAST 1000 SINCE STARTUP

System Status

Environment	test	Current time	2023-12-16T20:39:28 +0100
Data center	default	Uptime	00:44
		Lease expiration enabled	true
		Renews threshold	6
		Renews (last min)	8

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
REQUESTMANAGEMENT	n/a (1)	(1)	UP (1) - 10.191.248.137:requestManagement:8082
USERINFOSERVICE	n/a (1)	(1)	UP (1) - 10.191.248.137:userInfoService:5001
USERMANAGEMENT	n/a (1)	(1)	UP (1) - 10.191.248.137:userManagement:8081

Les microservices sont maintenant enregistrés et peuvent être découverts par d'autres services clients.

Dans le cadre de l'intégration du service de découverte Eureka dans le microservice UserListService, plusieurs ajustements ont été apportés pour dissocier les appels aux microservices de leurs adresses de déploiement réelles. Voici les étapes clés de cette mise en œuvre :

1. Création d'une Instance Unique de RestTemplate :

Une méthode annotée `@Bean` a été ajoutée dans la classe principale pour créer une seule instance de `RestTemplate` au lancement de l'application. Cette instance est annotée avec `@LoadBalanced` pour indiquer son utilisation avec Eureka.

2. Utilisation d'`@Autowired` pour Injecter RestTemplate dans le Microservice :

Dans la classe du microservice (`UserListResource`), une variable `RestTemplate` a été déclarée avec l'annotation `@Autowired`. Cela permet à Spring Boot d'injecter automatiquement l'instance correcte de `RestTemplate` au moment opportun.

3. Modification des Appels aux Microservices :

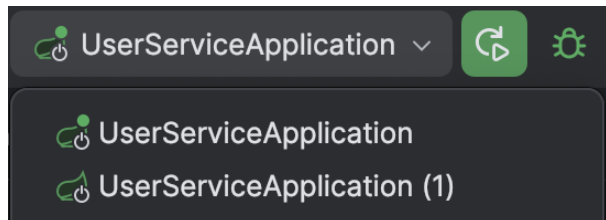
```
User user = restTemplate.getForObject( url: "http://userManagement/user/" + userId, User.class);

// Call RequestResource microservice to get request description
Request request = restTemplate.getForObject( url: "http://requestManagement/request/" + userId, Request.class);
```

c. Instanciation de microservice et répartition de charges (load balancing)

Dans cette section, l'objectif est de créer plusieurs instances d'un microservice, en l'occurrence, le microservice UserMangement, et de mettre en place la répartition de charge à l'aide de Spring Cloud.

On ajoute une deuxième configuration :



On constate deux instances du microservice userManagement : une instance sur le port 8081 et l'autre sur le port 8181.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
REQUESTMANAGEMENT	n/a (1)	(1)	UP (1) - 10.191.248.137:requestManagement:8082
USERINFOSERVICE	n/a (1)	(1)	UP (1) - 10.191.248.137:userInfoService:5001
USERMANAGEMENT	n/a (2)	(2)	UP (2) - 10.191.248.137:userManagement:8081 , 10.191.248.137:userManagement:8181

d. Externalisation des configurations

Pour le moment, notre architecture est constituée d'un service de découverte et de 3 microservices. Cette architecture va évoluer pour y intégrer un service de configuration.

Comme fait précédemment, nous avons créé un projet Spring Boot en ajoutant les dépendances Config Server.

Dans notre fichier client-service.properties qui est dans un dépôt git on a mis la configuration suivante :

Archi_Config / client-service-dev.properties

LamiaaHousni Update client-service-dev.properties

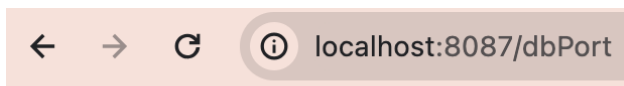
Code Blame 5 lines (5 loc) · 116 Bytes

```
1 db.host=srv-bdens.insa-toulouse.fr
2 db.port=3306
3 db.name=projet_gei_045
4 db.login=projet_gei_045
5 db.pwd=Thui8Lie
```

On a de même changé le fichier applications.properties comme suit :

```
server.port=8888
spring.cloud.config.server.git.uri=https://github.com/LamiaaHousni/Archi_Config.git
```

Une fois que c'est fait, lorsqu'on va sur <http://localhost:8087/dbHost> on trouve bien notre serveur, pareil pour le port et le reste des paramètres.



3306

Maintenant, on va essayer d'externaliser les paramètres de configuration du microservice UserManagement et RequestManagement. Ces services vont être connectés à une BDD relationnelle et on va simuler la récupération des paramètres de connexion de cette BDD. Tout d'abord, il faut ajouter les dépendances du client Eureka dans le fichier pom.xml du client-service pour qu'il soit découvrable et qu'on puisse l'appeler depuis les microservices. Par la suite, on va modifier les controller des deux microservices UserManagement et RequestManagement comme suit :

Remarque : on a créé des classes Volunteer et AidSeeker qu'on aurait pu utiliser au lieu de la classe User.

UserController :

```
@GetMapping("/dbInfo")
public String getDbInfo() {
    String dbHost = restTemplate.getForObject(uri: "http://client-service/dbHost", String.class);
    String dbPort = restTemplate.getForObject(uri: "http://client-service/dbPort", String.class);
    System.out.println("Database Host: " + dbHost + "Database Port: " + dbPort);

    return "Database Host: " + dbHost + "\n Database Port: " + dbPort;
}

@PostMapping(value = "/{idUser}/{name}", produces = MediaType.APPLICATION_JSON_VALUE)
public User addUser(
    @PathVariable int idUser,
    @PathVariable String name,
    @RequestParam String role) {
    String dbHost = restTemplate.getForObject(uri: "http://client-service/dbHost", String.class);
    String dbPort = restTemplate.getForObject(uri: "http://client-service/dbPort", String.class);
    User user = new User(idUser, name);

    String sql = "INSERT INTO users(name, role) VALUES (?, ?)";
    try (Connection conn = DriverManager.getConnection(uri: "jdbc:mysql://" + dbHost + ":" + dbPort +
        PreparedStatement stmt = conn.prepareStatement(sql)) {

        stmt.setString(parameterIndex: 1, user.getName());
        stmt.setString(parameterIndex: 2, role);

        stmt.executeUpdate();

    } catch (SQLException e) {
        e.printStackTrace();
    }

    return user;
}
```

Cette méthode gère la création d'un nouvel utilisateur en lui affectant un role(admin, volunteer ou aidSeeker) , effectue des appels au serveur client-service pour obtenir les paramètres de la BDD, insère les données de l'utilisateur dans une table users(qui a trois colonnes : id, nom et rôle), et renvoie une réponse en conséquence du résultat de l'opération.

Requêtes POST depuis POSTMAN :

POST ▼ http://localhost:8081/user/1/Alice?role=admin

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

Query Params

	Key	Value	Description
<input checked="" type="checkbox"/>	role	admin	

Body Cookies Headers Test Results 200 OK 1485 ms 187 B

POST ▼ http://localhost:8081/user/1/Kenza?role=aidSeeker

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

Query Params

	Key	Value	Description
<input checked="" type="checkbox"/>	role	aidSeeker	

Body Cookies Headers Test Results 200 OK 548 ms 187 B

On peut remarquer que les requêtes renvoient un message de réussite : 200 OK.

```
[mysql> SELECT * FROM users;
+----+-----+-----+
| id | name  | role  |
+----+-----+-----+
| 1  | Alice | admin |
| 2  | Lamiaa | volunteer |
| 3  | Kenza | aidSeeker |
| 4  | Vincent | aidSeeker |
| 5  | Thomas | aidSeeker |
| 6  | Jean | volunteer |
+----+-----+-----+
6 rows in set (0,07 sec)
```

⇒ Les données ont bien été ajoutées à la BDD.

RequestController :

```
@PostMapping(value="@*/addRequest/{seeker}", produces=MediaType.APPLICATION_JSON_VALUE)
public Request addRequest(@PathVariable int seeker, @RequestParam String requestDescription) {
    String dbHost = restTemplate.getForObject( url: "http://client-service/dbHost", String.class);
    String dbPort = restTemplate.getForObject( url: "http://client-service/dbPort", String.class);

    try (Connection conn = DriverManager.getConnection( url: "jdbc:mysql://" + dbHost + ":" + dbPort + "/" + "projet_gei_045", user: "projet_gei_045", password: "Thui8
    Statement stmt1 = conn.createStatement();
    ResultSet rs = stmt1.executeQuery( sql: "SELECT role FROM users WHERE id = "+seeker+"");

    while (rs.next()) {
        String role = rs.getString( columnName: "role");
        if ("aidSeeker".equals(role)) {
            PreparedStatement stmt = conn.prepareStatement( sql: "INSERT INTO request(seeker, request_description, status) VALUES (?, ?, 'WAITING')";

            stmt.setInt( parameterIndex: 1, seeker);
            stmt.setString( parameterIndex: 2, requestDescription);
            stmt.executeUpdate();
            System.out.println("AidSeeker -> creating request");
        } else {
            System.out.println("Not an AidSeeker");
        }
    }
} catch (SQLException e) {
    e.printStackTrace();
}

return new Request();
}
```

Cette méthode permet d'ajouter une demande d'aide, en effet le microservice va interroger d'abord la bdd pour vérifier si l'id passé en paramètre est présent dans la bdd en tant que aidSeeker. Si c'est le cas, la demande sera ajoutée dans la table request(colonnes : request_id | seeker | volunteer | request_description | status) sinon le message "Not an AidSeeker" sera affiché.

```
@PostMapping(value="@*/offerHelp/{volunteer}", produces=MediaType.APPLICATION_JSON_VALUE)
public Request offerHelp(@PathVariable int volunteer, @RequestParam String requestDescription) {
    String dbHost = restTemplate.getForObject( url: "http://client-service/dbHost", String.class);
    String dbPort = restTemplate.getForObject( url: "http://client-service/dbPort", String.class);

    try (Connection conn = DriverManager.getConnection( url: "jdbc:mysql://" + dbHost + ":" + dbPort + "/" + "projet_gei_045", user: "projet_gei_045", password: "Thui8Lie
    Statement stmt1 = conn.createStatement();
    ResultSet rs = stmt1.executeQuery( sql: "SELECT role FROM users WHERE id = "+volunteer+"");

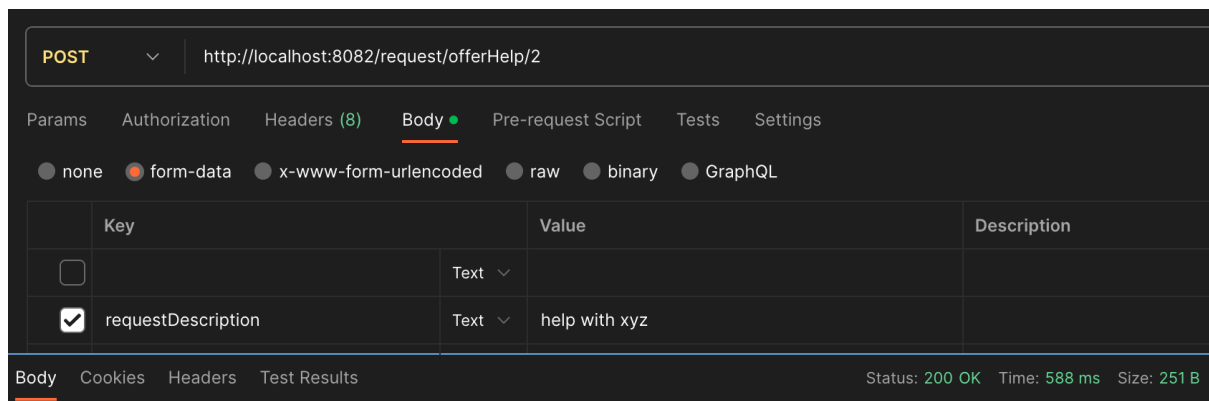
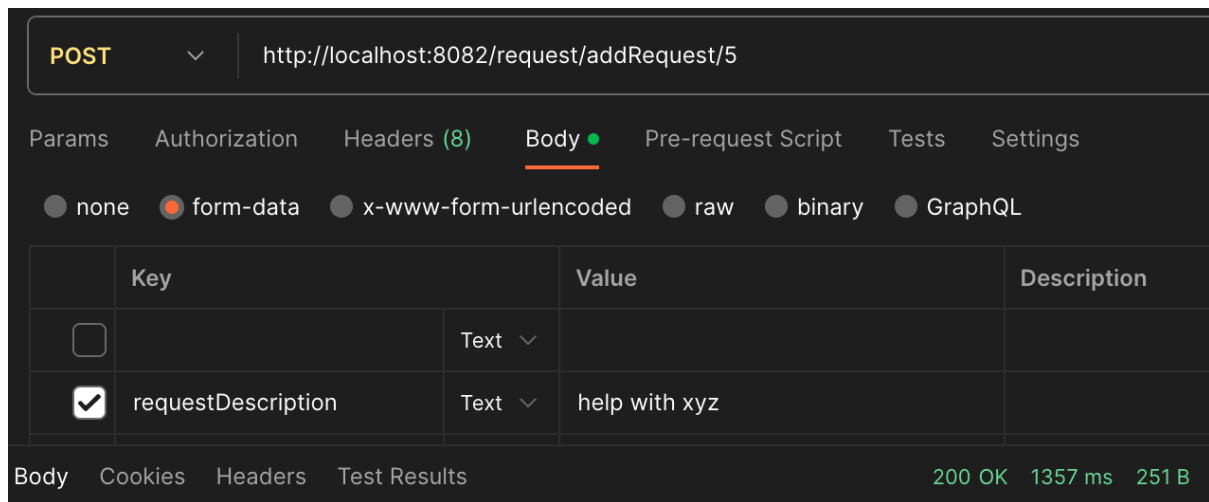
    while (rs.next()) {
        String role = rs.getString( columnName: "role");
        if ("volunteer".equals(role)) {
            PreparedStatement stmt = conn.prepareStatement( sql: "INSERT INTO request(volunteer, request_description, status) VALUES (?, ?, 'WAITING')";

            stmt.setInt( parameterIndex: 1, volunteer);
            stmt.setString( parameterIndex: 2, requestDescription);
            stmt.executeUpdate();
            System.out.println("Volunteer -> offering help");
        } else {
            System.out.println("Not a Volunteer");
        }
    }
} catch (SQLException e) {
    e.printStackTrace();
}

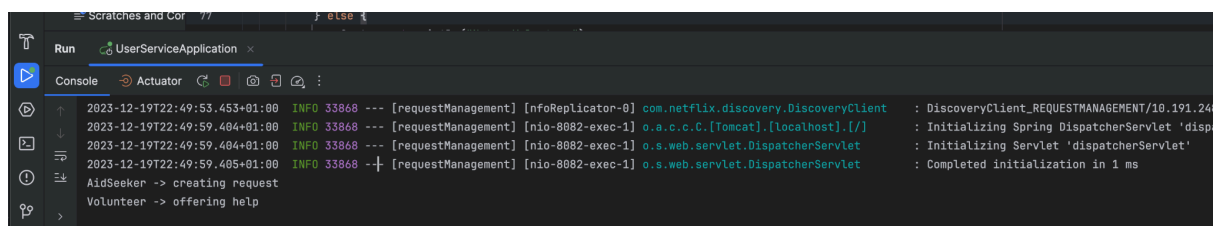
return new Request();
}
```

Cette méthode suit la même logique que celle d'avant et elle permet à un volontaire d'offrir de l'aide.

Requêtes POST depuis POSTMAN :



⇒ Status 200 OK



⇒ Creating request : OK

⇒ Offering help : OK

```
mysql> SELECT * FROM users;
+----+-----+-----+
| id | name  | role  |
+----+-----+-----+
| 1  | Alice | admin |
| 2  | Lamiaa | volunteer |
| 3  | Kenza | aidSeeker |
| 4  | Vincent | aidSeeker |
| 5  | Thomas | aidSeeker |
| 6  | Jean | volunteer |
+----+-----+-----+
6 rows in set (0,05 sec)

mysql> SELECT * FROM request;
+----+-----+-----+-----+-----+
| request_id | seeker | volunteer | request_description | status |
+----+-----+-----+-----+-----+
| 1          | 3      | NULL      | NULL                | 0      |
| 2          | 5      | NULL      | help with xyz       | WAITING |
+----+-----+-----+-----+-----+
2 rows in set (0,07 sec)

mysql> SELECT * FROM request;
+----+-----+-----+-----+-----+
| request_id | seeker | volunteer | request_description | status |
+----+-----+-----+-----+-----+
| 1          | 3      | NULL      | NULL                | 0      |
| 2          | 5      | NULL      | help with xyz       | WAITING |
| 3          | NULL   | 2         | help with xyz       | WAITING |
+----+-----+-----+-----+-----+
```

⇒ Les demandes ont bien été ajoutées à la table requests.

Si l'id ne correspond pas dans la table des users :

```
2023-12-19T22:49:59.404+01:00 INFO 33868 --- [requestManagement] [nio-8082-exec-1] c
2023-12-19T22:49:59.405+01:00 INFO 33868 --- [requestManagement] [nio-8082-exec-1] c
AidSeeker -> creating request
Volunteer -> offering help
Not a Volunteer
Not an AidSeeker
```

Par manque de temps, on a pas pu ajouter plus de méthodes et/ou microservices pour mieux simuler l'application Volunteering. On aurait pu ajouter un microservice qui permet d'ajouter des feedbacks et un deuxième qui gère l'attribution des demandes.

Conclusion :

Les choix d'architecture dépendent des besoins spécifiques du projet. SOAP offre des fonctionnalités avancées, tandis que REST se distingue par sa simplicité et son adéquation avec les technologies modernes. Les microservices, en revanche, visent à résoudre des problèmes de scalabilité et de maintenance en adoptant une approche modulaire. Le choix entre ces architectures dépend des exigences du projet en termes de performances, de flexibilité et de simplicité.