

Conception et Transition Architecturale : De SOAP à REST et Vers une Architecture de Microservices pour une Application d'Assistance

Url Git : <https://github.com/Abdel211/Serv-archi.git>

HATIBI Mohammed Amine
Anka Soubaai Abdelmajid

5ISS

Table de matieres :

Introduction.....	3
--------------------------	----------

I-Architecture SOAP

1) Serveur SOAP.....	4
2) Client SOAP.....	9
3) Bilan sur l'utilisation de SOAP.....	10

II-Architecture REST

1) Serveur REST.....	11
2) Client REST.....	12
3) Bilan Rest Vs SOAP.....	14

III-Microservice

1) Première Version de nos microservice en utilisant Spring Boot.....	16
2) Ajout du microservice de découverte.....	23
3) Ajout du microservice de configuration.....	28
4) Bilan SOAP vs REST vs Microservices.....	31

IV-Conclusion.....	32
---------------------------	-----------

Introduction

L'évolution rapide des besoins et des exigences technologiques dans le développement d'applications modernes a conduit à une exploration continue des architectures logicielles. Notre projet, axé sur la création d'une application visant à aider les personnes vulnérables, a été le terrain d'une réflexion approfondie sur les choix architecturaux. Cette initiative découle d'une constatation simple mais cruciale : un grand nombre d'individus se trouvent isolés au sein de notre société pour diverses raisons, qu'il s'agisse de distances géographiques, de problèmes de santé incapacitants, ou de séjours prolongés à l'hôpital.

Face à cette réalité, notre objectif était de concevoir une plateforme numérique permettant de connecter ceux qui ont besoin d'aide avec des bénévoles prêts à offrir leur soutien. Au cours de ce projet, nous avons adopté et étudié trois architectures logicielles majeures : SOAP, REST et les microservices, chacune offrant ses propres avantages et défis.

Ce compte rendu documente notre parcours à travers ces architectures, depuis les premières étapes de conception en utilisant SOAP, jusqu'à la transition vers REST, puis vers une architecture de microservices plus flexible et évolutive. Nous explorerons les motivations derrière chaque choix architectural, les ajustements opérés à chaque phase, et nous évaluerons l'impact de ces décisions sur l'application d'aide aux personnes vulnérables.

Cette analyse approfondie des trois architectures permettra de mieux comprendre leurs spécificités, leurs implications pratiques et leur adéquation avec les exigences changeantes de notre application.



I-Architecture SOAP

1) Serveur Soap

L'architecture SOAP, un protocole de communication basé sur XML a été sélectionnée comme base fondamentale pour le développement de l'application d'aide aux personnes vulnérables. Cette architecture repose sur la création de services web interopérables, permettant l'échange de messages structurés entre différents systèmes.

Au cœur de cette architecture se trouvent plusieurs classes cruciales au sein du serveur de l'application. La première classe, "**AddRequest**", expose un service web via l'annotation "**@WebService**" et met à disposition une méthode nommée "**addNewRequest**". Cette méthode accepte en paramètres des informations telles que le nom du demandeur et les détails spécifiques de la demande. Une fois ces données reçues, une nouvelle instance de la classe "**Request**" est créée pour représenter la demande, puis ajoutée à une liste interne de demandes en attente.

Parallèlement, la classe "AddUser" offre un service web similaire nommé "**addUserRequestHelp**". Ce service permet à de nouveaux utilisateurs de s'enregistrer en fournissant des détails tels que leur nom d'utilisateur, leur adresse e-mail et les détails de la demande d'aide qu'ils souhaitent offrir. Ces données sont utilisées pour créer une nouvelle instance de la classe "**User**" et sont ensuite intégrées à une liste dédiée aux utilisateurs demandant de l'aide.

Pour permettre l'accessibilité à ces services, la classe "**lancerservice**" joue un rôle central. Elle est responsable du démarrage et de la publication des services web. En utilisant la méthode `Endpoint.publish()`, cette classe configure des points de terminaison spécifiques, tels que "**AddUser**" et "**AddRequest**", à des URL définies, rendant ainsi ces services disponibles pour les utilisateurs de l'application.

```
déc. 13, 2023 8:29:34 PM com.sun.xml.ws.server.MonitorBase createRoot
INFO: Metro monitoring rootname successfully set to: com.sun.metro:pp=/,type=WSEndpoint,name=AddUserService-AddUserPort
Service web AddUser démarré avec succès : http://localhost:8089/AddUser
```

Figure 1 : Démarrage du service "Add User"

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, spécifiant les opérations disponibles, les formats de messages, les protocoles de communication et les adresses de points de terminaison. Ce fichier WSDL agit comme un contrat entre le client et le serveur, détaillant comment accéder et interagir avec les services exposés.

Chaque service web que nous avons développé est associé à un point de terminaison unique identifié par un port spécifique. Cependant, une optimisation a été apportée dans la configuration des services. Dans la classe regroupant les deux fonctions de service, "lancerservice", initialement conçue pour placer chaque service sur un port distinct, une observation a été faite sur l'avantage de regrouper ces services sur le même port. Cette consolidation des services sur un port unique a été considérée comme une amélioration, simplifiant ainsi la configuration et la gestion des points de terminaison. Cette modification a permis de regrouper efficacement les services, offrant une meilleure gestion et facilitant les interactions entre les différents services au sein de l'application.

Cette approche a été testée et validée en utilisant le WSDL pour accéder et interagir avec ces services depuis le côté serveur. En utilisant le WSDL, nous avons pu vérifier la conformité des services exposés par rapport à leur description et nous assurer que les clients potentiels pouvaient aisément comprendre et interagir avec ces services web.

L'association de ces services sur un même port, tout en étant conforme aux standards définis par le WSDL, a démontré une amélioration de l'efficacité et de la gestion des services au sein de l'architecture SOAP de notre application.

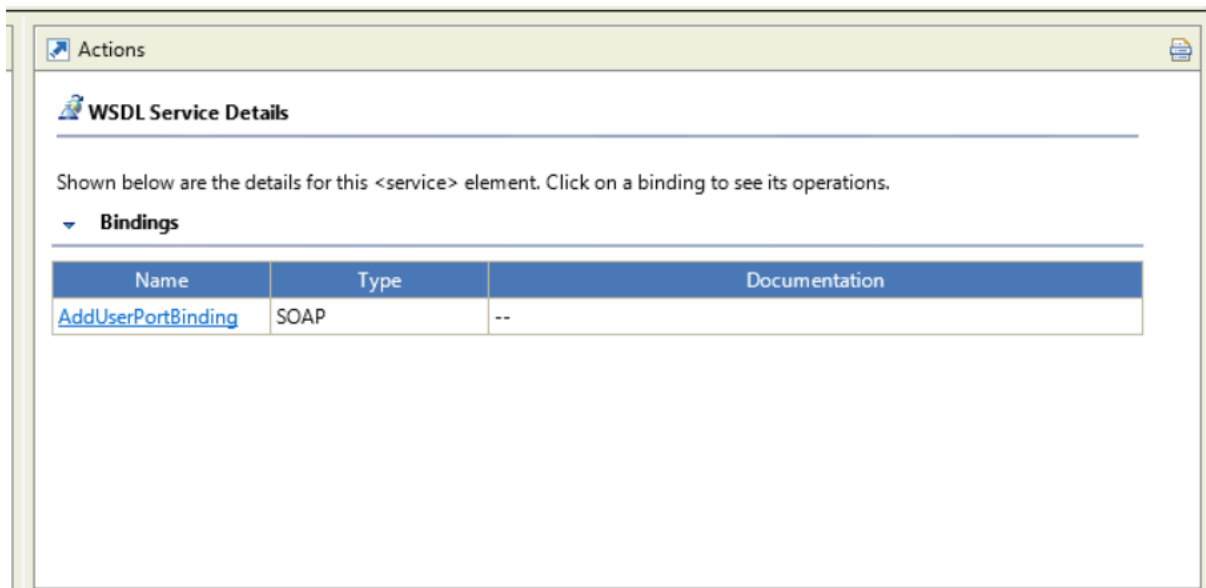


Figure 2 : Utilisation du WSDL pour le service AddUser

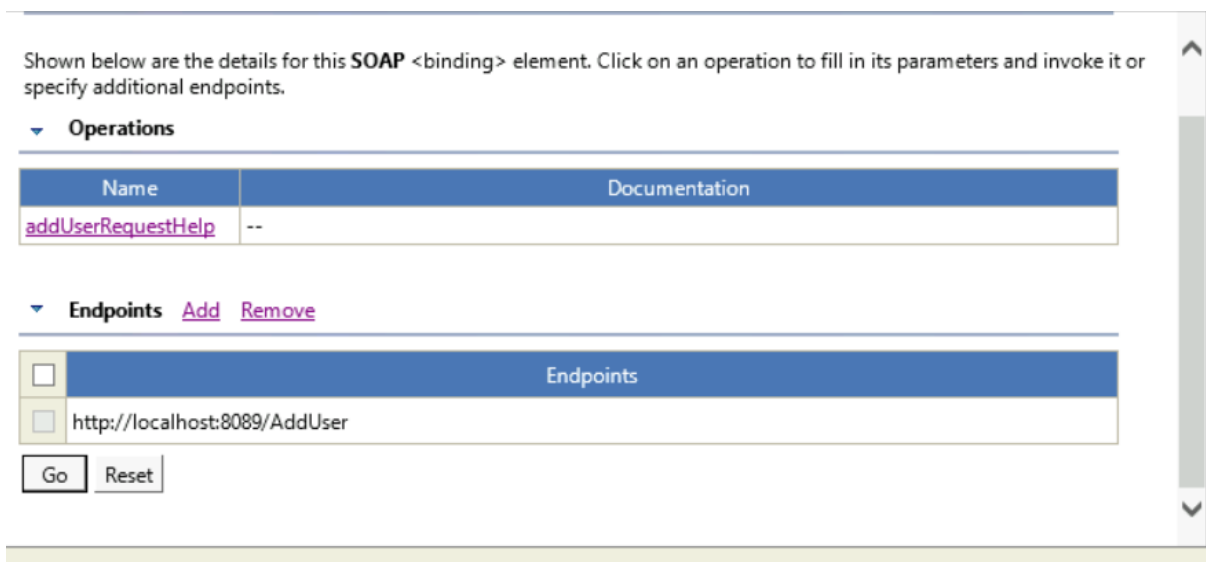


Figure 3 : Démarrage du service AddUser

Body

addUserRequestHelp

username string [Add](#) [Remove](#)

☐ Values

email string [Add](#) [Remove](#)

☐ Values

requestDetails string [Add](#) [Remove](#)

☐ Values

Figure 4 : Simulation du Service "Add User"

Source

Body

addUserRequestHelpResponse

return (string): Utilisateur Abdel ajouté avec succès pour demander de l'aide.

déc. 13, 2023 8:45:35 PM com.sun.xml.ws.server.MonitorBase createRoot

INFO: Metro monitoring rootname successfully set to: com.sun.metro:pp=/,type=WSEndpoint,name=AddRequestService-AddRequestPort

Service web AddRequest démarré avec succès : http://localhost:8090/AddRequest

Figure 5 : Résultat de la simulation du service AddUser

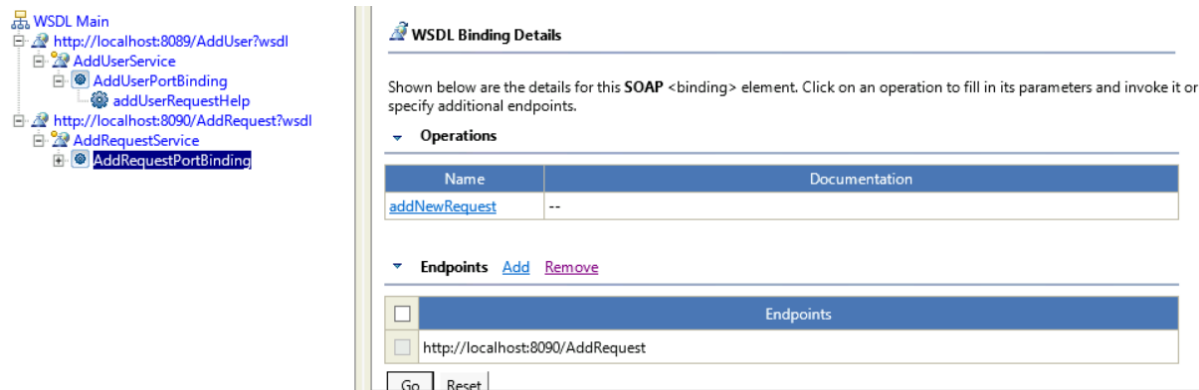


Figure 6 : Utilisation du WSDL pour le service AddRequest

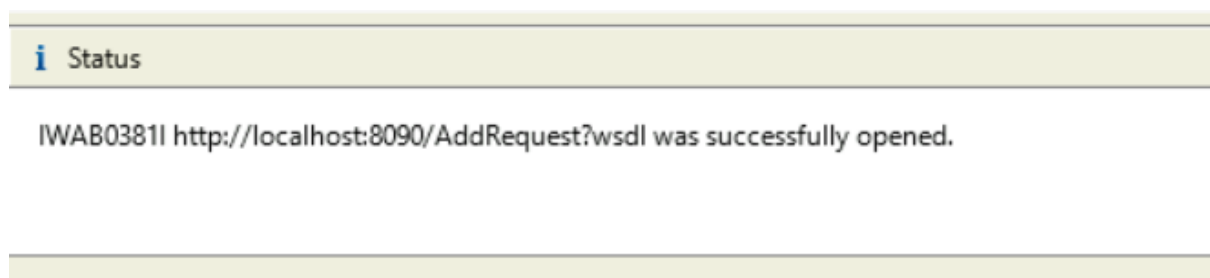


Figure 7 : Démarrage du service AddRequest



Figure 8 : Simulation du service AddRequest

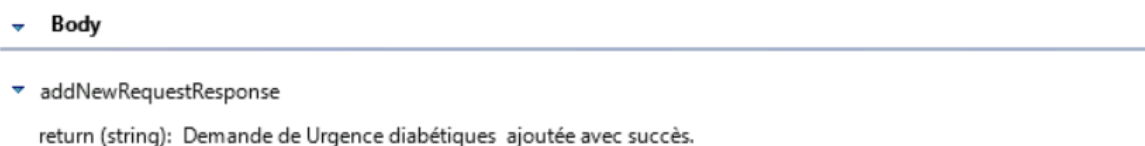


Figure 9 : Résultat du service "Add Request"

2) Client SOAP :

La classe `Client` constitue l'interface entre le client et le service web SOAP exposé côté serveur. Son rôle essentiel est de permettre la communication et l'utilisation des fonctionnalités offertes par le service. À travers l'utilisation de l'API JAX-WS, cette classe établit une connexion avec le service web en se basant sur le fichier WSDL (`client.wsdl`). Elle utilise les informations détaillées dans ce fichier pour créer une instance du service web (`AddRequestService`) en spécifiant l'URL du WSDL. Une fois cette connexion établie, le client peut invoquer les méthodes exposées par le service, telles que `addNewRequest`, en fournissant les données nécessaires à travers lesquelles il peut interagir avec le service et traiter les réponses obtenues.

Le fichier `client.wsdl`, quant à lui, fournit une description complète du service web côté serveur. Il spécifie les opérations disponibles, les types de données associés, ainsi que les détails de configuration essentiels pour que le client puisse accéder et utiliser correctement le service. Grâce à ces informations détaillées dans le WSDL, le client peut comprendre la structure et les modalités d'utilisation du service, lui permettant ainsi d'interagir de manière adéquate avec celui-ci.

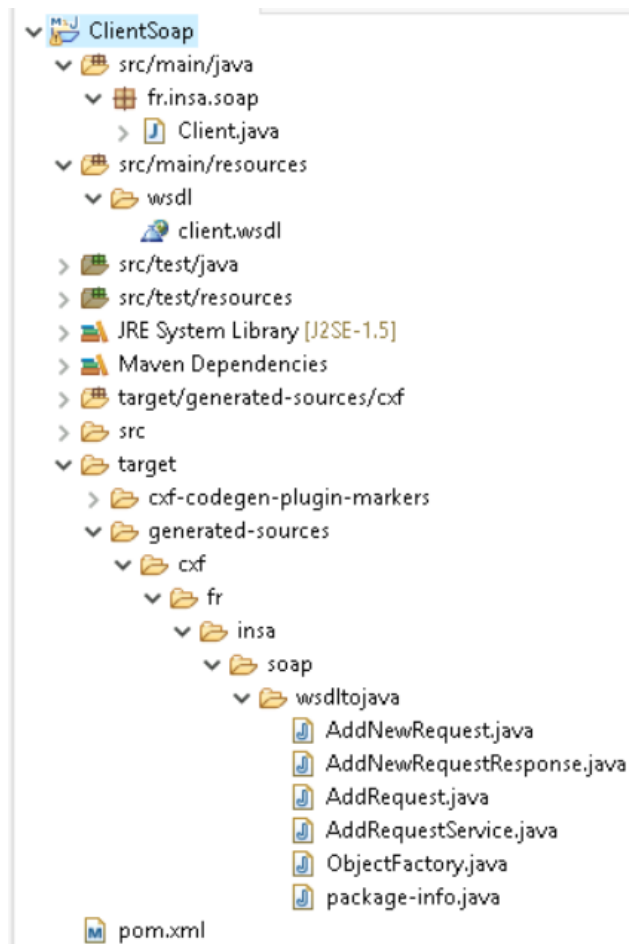


Figure 10 : Architecture côté serveur

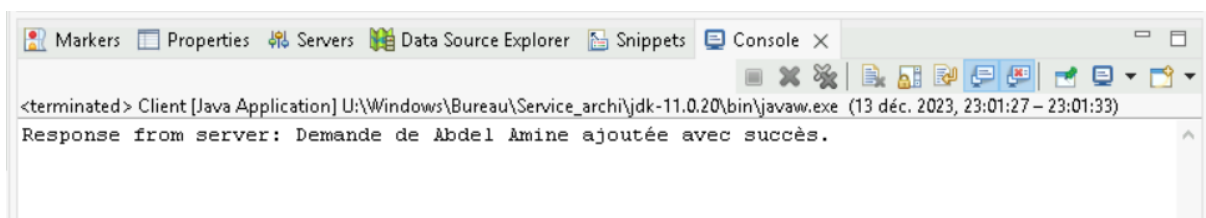


Figure 11 : Simulation de mon client

3) Bilan sur l'utilisation de SOAP :

L'adoption de l'architecture SOAP a offert une communication sécurisée et structurée entre le client et le serveur grâce à l'interopérabilité assurée par les messages XML. Les avantages incluent une description claire des services via le fichier WSDL et une sécurité renforcée via WS-Security. Cependant, cette approche peut être complexe et rigide, avec un potentiel surcoût en bande passante dû à la structure XML des messages. Malgré ses avantages, la complexité et la rigidité de SOAP peuvent être limitantes pour des scénarios nécessitant plus de flexibilité.

II-Architecture Rest :

1) Serveur Rest

Initialement configuré avec une architecture basée sur les principes de SOAP, notre projet a évolué vers l'utilisation de l'approche REST, notamment avec l'intégration du framework Jersey. Bien que les configurations initiales soient restées inchangées dans un premier temps, cette transition vers REST avec Jersey a marqué une évolution significative dans la manière dont nos services web sont conçus et déployés .

Notre serveur REST est désormais accessible via localhost sur le port 8080. En cliquant sur Jersey ressource nous sommes automatiquement redirigé .



Pour confirmer le bon fonctionnement de notre service d'ajout de demandes d'aide, nous avons utilisé Postman, un outil de test d'API. En sélectionnant la méthode POST et en configurant le corps de la requête avec les clés '**requestername**' et '**requestdetails**', nous avons pu soumettre des demandes d'aide simulées au serveur. Cette approche nous a permis de valider l'intégrité et la précision du service, confirmant ainsi la possibilité d'ajouter des demandes d'aide au sein de notre système grâce à cette interface REST nouvellement adoptée.

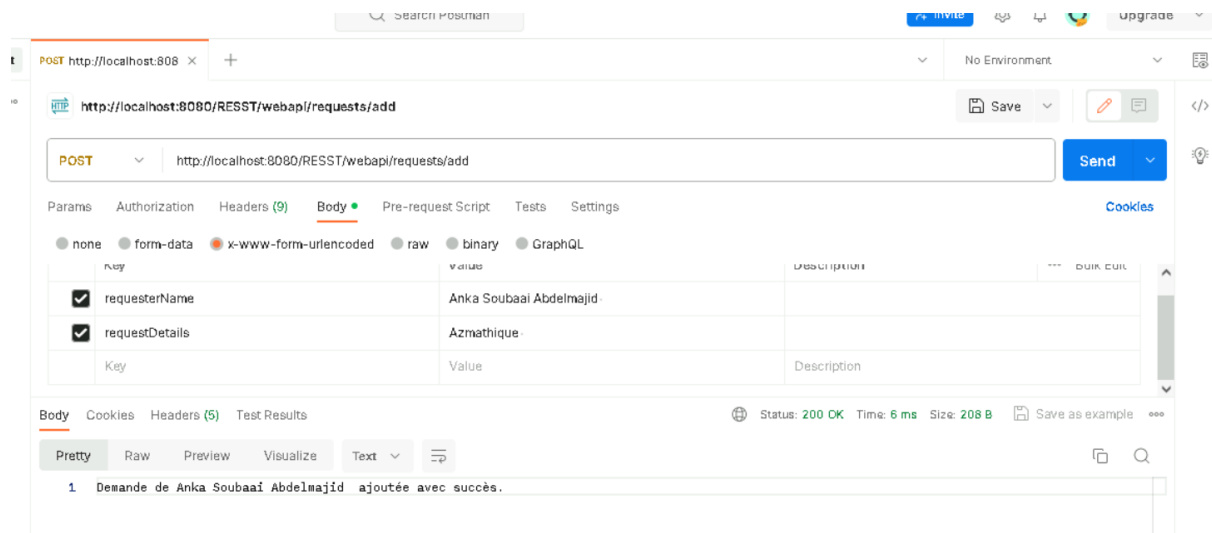


Figure 12 : Simulation de notre serveur via Postman

2) Client Rest :

Nous avons développé un client REST pour interagir avec notre service web. Ce client, implémenté dans la classe '**RestClient**', permet à l'utilisateur de saisir les détails d'une demande d'aide et les détails d'un volontaire, en utilisant les fonctionnalités offertes par notre service REST.

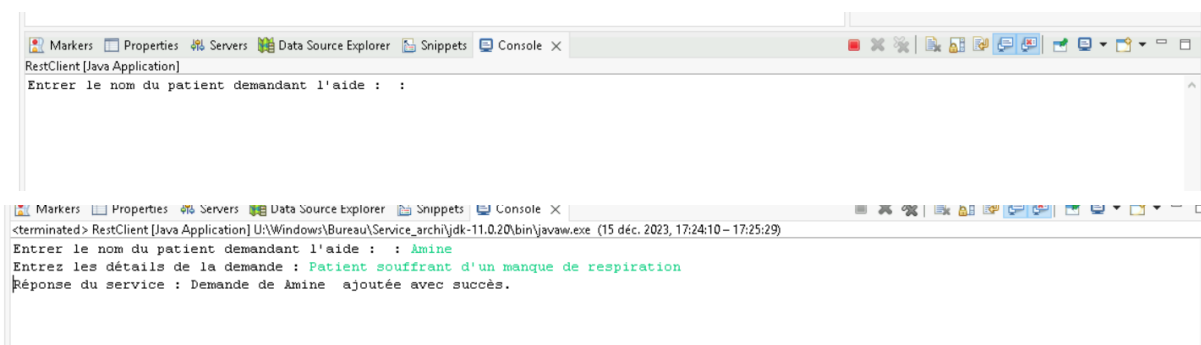


Figure 13 : Version initiale du client sans capacité d'acceptation ou de rejet des demandes

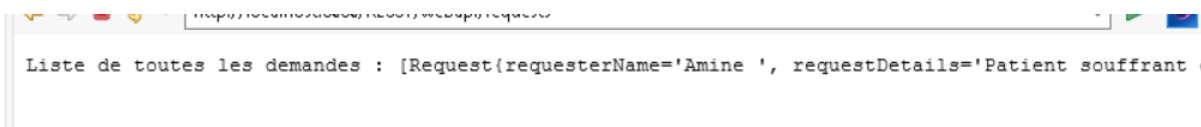


Figure 14 : Résultat de la simulation du serveur REST via Postman

```

RestClient [Java Application] U:\Windows\Bureau\Service_archi\jdk-11.0.20\bin\javaw.exe (16 déc. 2023, 15:17:29)
Nom du demandeur d'aide : Amine Hatibi
Détails de la demande : Patient qui a besoin d'aide pour son opération de foie
Réponse du service pour la demande : Demande de Amine Hatibi ajoutée avec succès.
Voulez-vous accepter cette demande ? (Oui/Non) :

RestClient [Java Application] U:\Windows\Bureau\Service_archi\jdk-11.0.20\bin\javaw.exe (16 déc. 2023, 15:17:29)
Nom du demandeur d'aide : Amine Hatibi
Détails de la demande : Patient qui a besoin d'aide pour son opération de foie
Réponse du service pour la demande : Demande de Amine Hatibi ajoutée avec succès.
Voulez-vous accepter cette demande ? (Oui/Non) : Oui
Réponse du service pour l'acceptation : La demande à l'index 0 a été acceptée.

```

Figure 15 : Version améliorée avec intégration de la fonctionnalité d'acceptation ou de rejet

```

http://localhost:8080/RESST/webapi/requests
Liste des demandes d'aide :
Request{requesterName='Amine Hatibi', requestDetails='Patient qui a besoin d'aide pour son opÃ©ration de foie'}

```

Figure 16 : Réponse de Postman après l'utilisation de la version améliorée

Nous avons introduit la méthode `addVolontaire` dans la classe `RequestResource` pour gérer les demandes d'inscription en tant que volontaire. Cette fonctionnalité évalue les nouveaux volontaires en fonction de critères spécifiques tels que les antécédents judiciaires ou l'âge. Si ces critères ne sont pas conformes aux exigences définies, la demande est rejetée. En revanche, si le volontaire répond aux critères, il est ajouté à la liste statique des volontaires. La décision d'accepter ou de refuser un volontaire dépend donc des informations fournies par le client et de leur conformité aux critères établis.

```

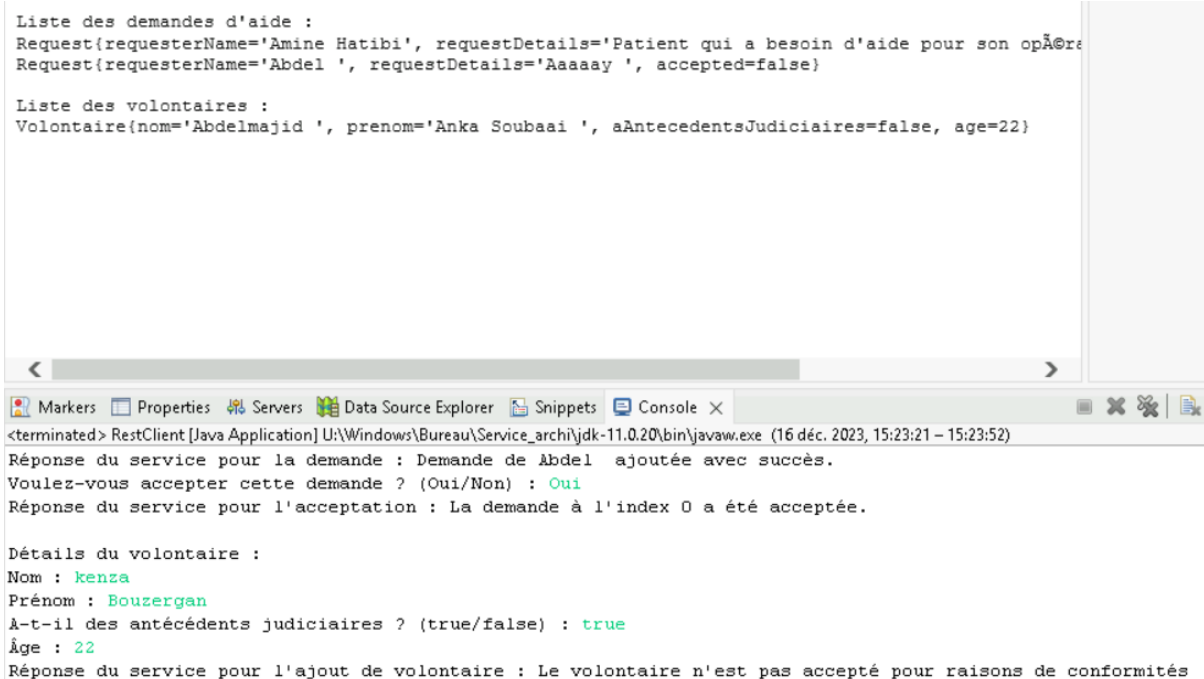
Détails du volontaire :
Nom : Abdelmajid
Prénom : Anka Soubaai
A-t-il des antécédents judiciaires ? (true/false) : false
Âge : 22
Réponse du service pour l'ajout de volontaire : Le volontaire a été accepté.

```

Figure 17 : Simulation du client pour l'ajout d'un volontaire

```
Liste des volontaires :
Volontaire{nom='Abdelmajid ', prenom='Anka Soubaai ', aAntecedentsJudiciaires=false, age=22}
```

Figure 18 : Liste des volontaires après l'ajout d'un nouvel inscrit



```
Liste des demandes d'aide :
Request{requesterName='Amine Hatibi', requestDetails='Patient qui a besoin d'aide pour son opÃ©ration', accepted=false}
Request{requesterName='Abdel ', requestDetails='Aaaaay ', accepted=false}

Liste des volontaires :
Volontaire{nom='Abdelmajid ', prenom='Anka Soubaai ', aAntecedentsJudiciaires=false, age=22}

<terminated> RestClient [Java Application] U:\Windows\Bureau\Service_archi\jdk-11.0.20\bin\javaw.exe (16 d c. 2023, 15:23:21 - 15:23:52)
R ponse du service pour la demande : Demande de Abdel ajout e avec succ s.
Voulez-vous accepter cette demande ? (Oui/Non) : Oui
R ponse du service pour l'acceptation : La demande   l'index 0 a  t  accept e.

D tails du volontaire :
Nom : kenza
Pr nom : Bouzergan
A-t-il des ant c dents judiciaires ? (true/false) : true
 ge : 22
R ponse du service pour l'ajout de volontaire : Le volontaire n'est pas accept  pour raisons de conformit s
```

Figure 19 : Exemple de refus d'un volontaire en raison de ses ant c dents judiciaires

3) Bilan Rest Vs SOAP:

Une analyse comparative entre les mod les SOAP et REST r v le des diff rences fondamentales. SOAP, reconnu pour sa robustesse et sa s curit , pr sente cependant une complexit  plus  lev e et des performances parfois moins rapides. En revanche, REST offre une approche plus l g re et simple, favorisant des performances rapides, mais avec moins de fonctionnalit s de s curit  avanc es. Le choix entre les deux d pend des besoins sp cifiques du projet : SOAP convient aux environnements n cessitant une s curit  et une robustesse  lev es, tandis que REST est pr f rable pour des applications o  la simplicit  et la rapidit  priment.

III-Microservice :

La transition vers un microservice dans notre application a été une étape cruciale pour améliorer la flexibilité et la scalabilité de notre système. Nous avons choisi de découper notre application en services distincts pour mieux isoler les fonctionnalités, favoriser la modularité et simplifier la maintenance. Cela nous a permis de concentrer chaque service sur une tâche spécifique, réduisant ainsi la complexité de notre code.

Nous avons structuré notre architecture en plusieurs microservices : un pour gérer les demandes d'aide, un autre pour les volontaires, et ainsi de suite. Chaque microservice est responsable d'une fonctionnalité particulière, ce qui facilite l'évolutivité de notre application et nous permet d'apporter des modifications spécifiques à chaque service sans impacter l'ensemble du système.

En réorganisant notre application en microservices, nous avons également gagné en agilité. Chaque microservice peut être développé, testé, déployé et mis à l'échelle de manière indépendante. Cela favorise une approche plus modulaire et permet à différentes équipes de travailler simultanément sur différents services, accélérant ainsi le développement.

L'ajout de fonctionnalités spécifiques à chaque microservice, comme la gestion des demandes pour celui dédié aux volontaires, a été motivé par la nécessité d'une gestion plus fine des données et des opérations pour répondre aux exigences métier spécifiques à chaque domaine.

En somme, cette transition vers un microservice a été guidée par notre désir de modularité, de maintenabilité, et de scalabilité. Elle nous a permis de mieux structurer notre application, de simplifier la gestion des fonctionnalités et de faciliter son évolution continue.

1) Première Version de nos microservice en utilisant Spring Boot :

Tout commence par une première version utilisant Spring Boot avec une unique dépendance liée au service web. Cette itération initiale se trouve dans le dossier "microservice Spring Boot" de notre dépôt Git.

Lors de la création de notre microservice avec Spring Boot, nous avons rencontré un problème de compatibilité entre les versions Java prises en charge par notre IDE et celles requises par Spring Boot.

The image shows a configuration window for a Spring Boot project. It has a dark theme. The 'Project' section has radio buttons for 'Gradle - Groovy' (selected), 'Gradle - Kotlin', and 'Maven'. The 'Language' section has radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'. The 'Spring Boot' section has radio buttons for '3.2.1 (SNAPSHOT)', '3.2.0' (selected), '3.1.7 (SNAPSHOT)', and '3.1.6'. The 'Project Metadata' section contains text input fields for 'Group' (com.example), 'Artifact' (demo), 'Name' (demo), 'Description' (Demo project for Spring Boot), and 'Package name' (com.example.demo). At the bottom, the 'Packaging' section has radio buttons for 'Jar' (selected) and 'War', and the 'Java' section has radio buttons for '21' and '17' (selected).

Project		Language	
<input checked="" type="radio"/> Gradle - Groovy		<input checked="" type="radio"/> Java	<input type="radio"/> Kotlin
<input type="radio"/> Gradle - Kotlin	<input type="radio"/> Maven	<input type="radio"/> Groovy	
Spring Boot			
<input type="radio"/> 3.2.1 (SNAPSHOT)	<input checked="" type="radio"/> 3.2.0	<input type="radio"/> 3.1.7 (SNAPSHOT)	<input type="radio"/> 3.1.6
Project Metadata			
Group	com.example		
Artifact	demo		
Name	demo		
Description	Demo project for Spring Boot		
Package name	com.example.demo		
Packaging	<input checked="" type="radio"/> Jar	<input type="radio"/> War	
Java	<input type="radio"/> 21	<input checked="" type="radio"/> 17	

Figure 20 : Utilisation de Spring Boot pour développer notre microservice

Comme on peut le voir sur la figure suivante, notre environnement de développement n'intégrait que des versions de Java de 1 à 16, alors que Spring Boot nécessitait Java 17.

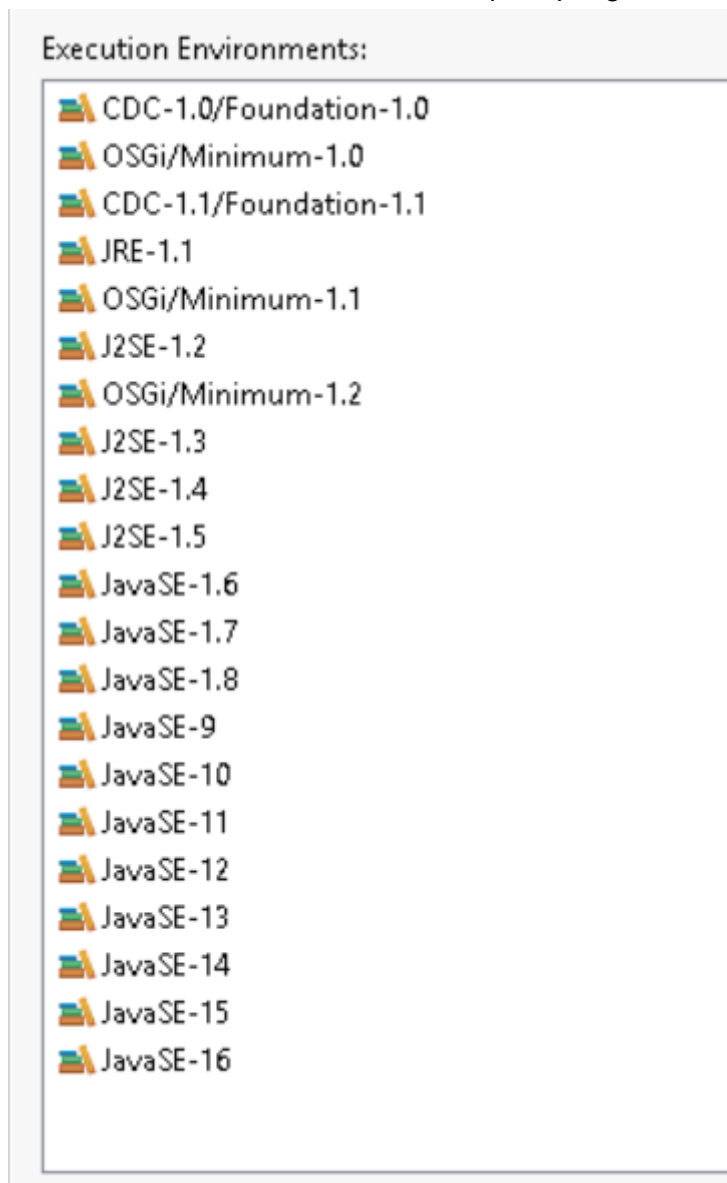


Figure 21 : Version de Java prise en charge par notre IDE

Pour contourner cela, nous avons ajusté les versions localement dans notre fichier `pom.xml`. Nous avons spécifié l'utilisation de la version Spring Boot 2.5.6 et avons fixé la version Java à 11 pour assurer la compatibilité et permettre le bon fonctionnement de notre application.

```

        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>fr.insa.mas</groupId>
    <artifactId>Help</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>Help</name>
    <description>Microservice Part</description>
    <properties>
        <java.version>11</java.version>
    </properties>
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.6</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

```

Figure 21 : Contournement du problème en ajustant le fichier Pom.xml

Notre microservice repose sur une architecture constituée de quatre classes dédiées aux contrôleurs, orchestrant la gestion des requêtes et des données, accompagnées d'une classe de lancement de l'application.

- **HelpApplication** : Point d'entrée de l'application Spring Boot, cette classe initialise et démarre l'ensemble du service.
- **Request** : Représente une demande d'aide avec des attributs tels que le nom du demandeur, les détails de la demande, l'état d'acceptation et le statut de la demande.
- **RequestController** : Contrôleur REST responsable de la gestion des requêtes relatives aux demandes d'aide. Il supervise l'ajout de nouvelles demandes, la récupération de toutes les demandes existantes, l'acceptation ou le rejet d'une demande spécifique et la modification du statut d'une demande.
- **Volontaire** : Modèle de données pour un volontaire, contenant des informations telles que le nom, le prénom, les antécédents judiciaires et l'âge.
- **VolontaireController** : Contrôleur REST gérant les requêtes liées aux volontaires. Il autorise l'ajout de nouveaux volontaires tout en vérifiant certains critères et récupère la liste de tous les volontaires existants.

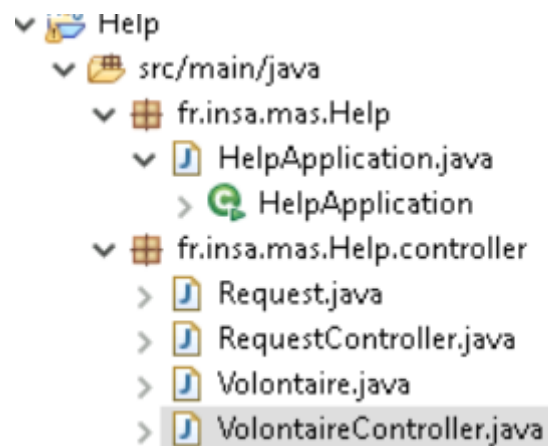


Figure 22 : Architecture de la première version

Nos services sont accessibles localement via le port 8081, un paramètre configurable disponible dans le fichier `application.properties`.

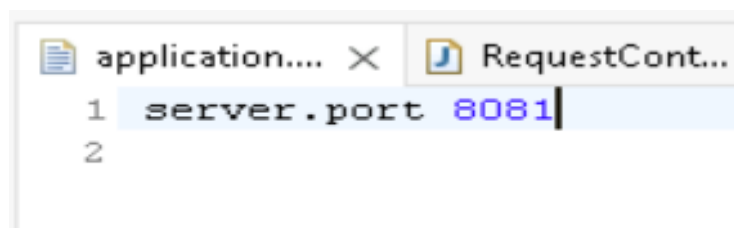


Figure 23 : Paramétrage du port pour nos microservices

Pour les services, nous avons maintenu la même logique qu'auparavant, mais nous avons introduit une méthode permettant de modifier l'état d'une demande d'aide (par exemple, en cours, non traitée, etc.).

Nous effectuons des tests de nos microservices avec Postman en envoyant des requêtes POST pour chaque service, sauf pour celui permettant de visualiser toutes les demandes d'aide où nous utilisons une requête GET. Chacun de ces services est accessible via des URL spécifiques telles que <http://localhost:port/endpoint>.

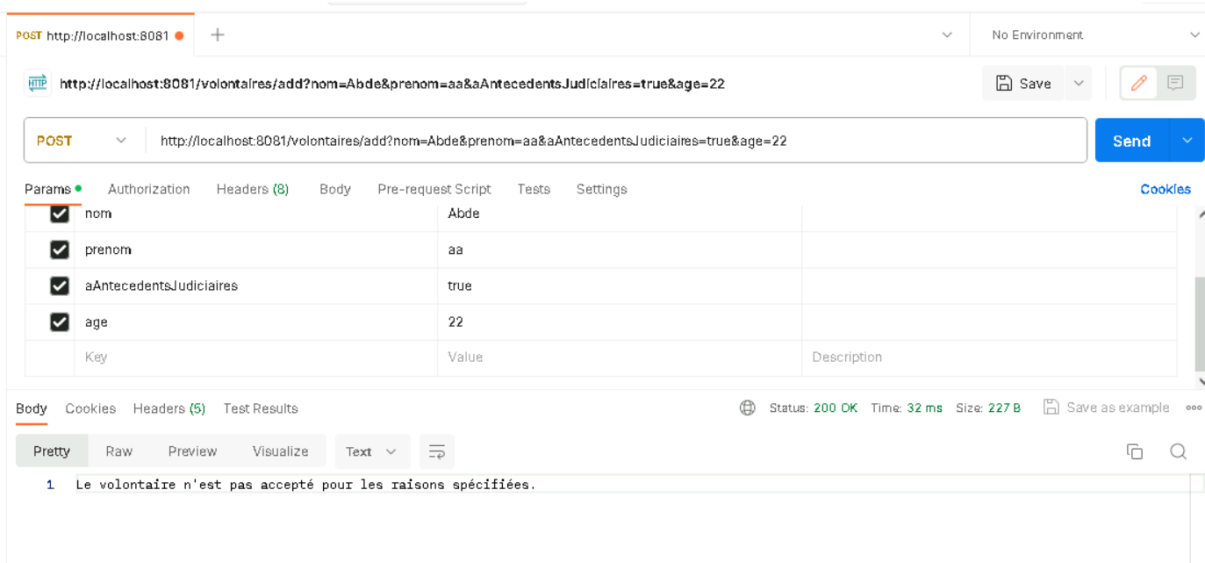


Figure 24 : Test de l'ajout de volontaires pour l'ajout de volontaires, condition non vérifiée

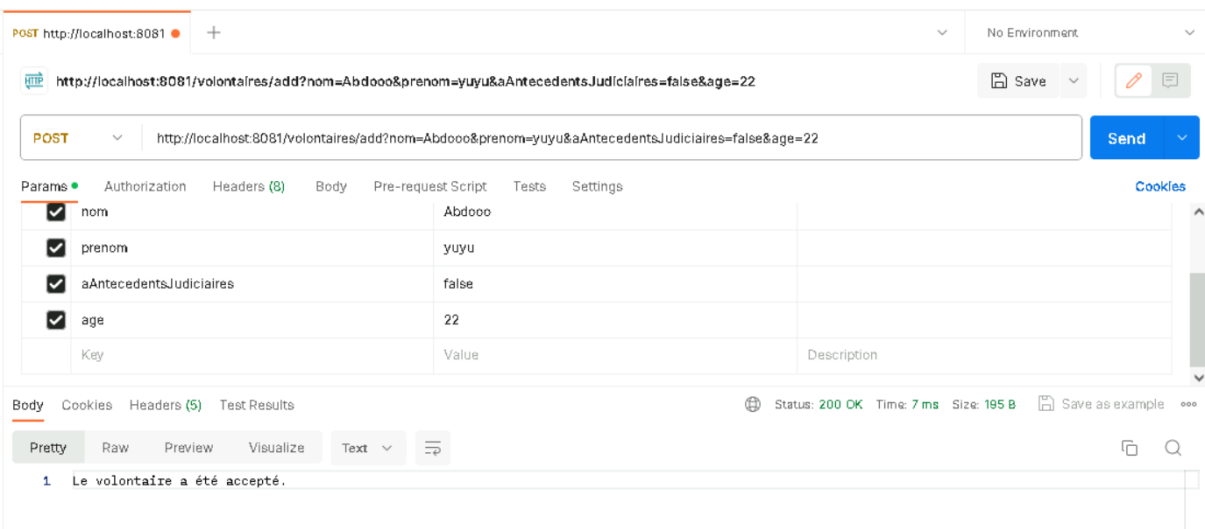


Figure 25 : Test de l'ajout de Volontaires pour l'ajout de volontaires, condition vérifiée

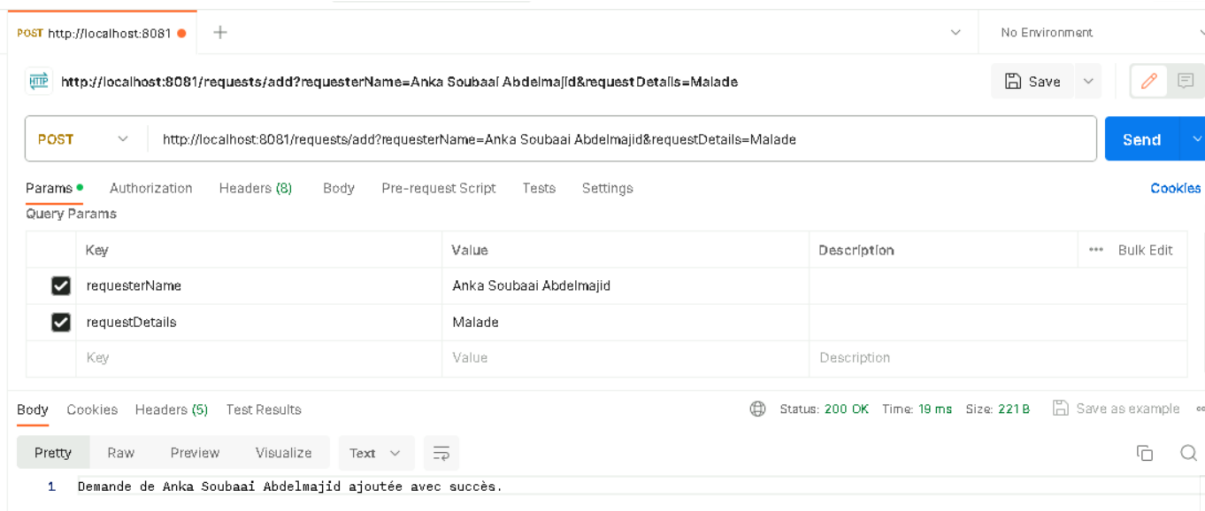


Figure 26 : Test de l'ajout de demande d'aide

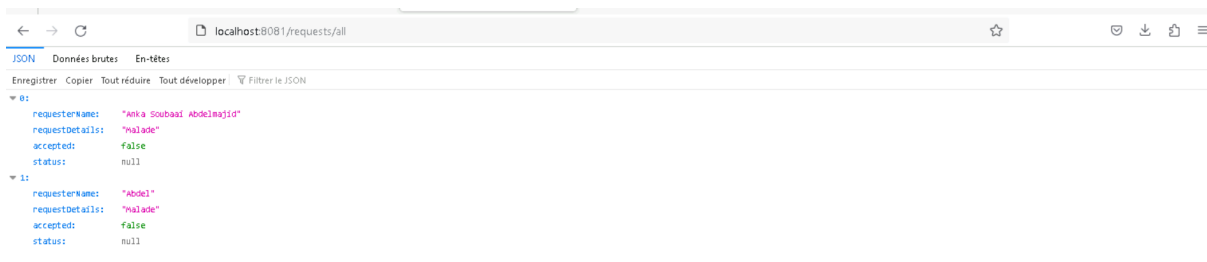


Figure 27 : Visualisation de toutes les demandes d'aide, aucune demande n'a été acceptée pour le moment

Pour accepter, rejeter ou modifier le statut d'une demande, une requête POST est envoyée à l'URL suivante : <http://localhost:port/endpoint> (où le endpoint est soit reject, accept ou status), suivi de l'index de la demande. L'index est visible sous forme JSON pour toutes les demandes existantes à l'URL <http://localhost:port/request/all>.

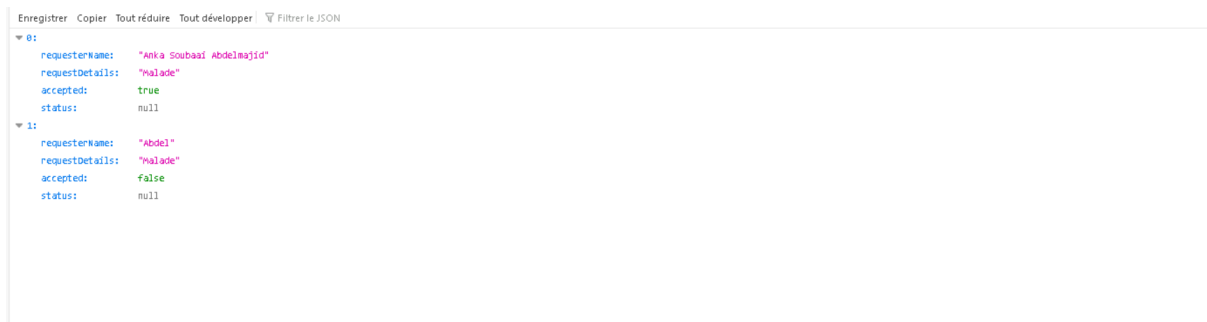


Figure 28 : Acceptation de la demande d'index 0

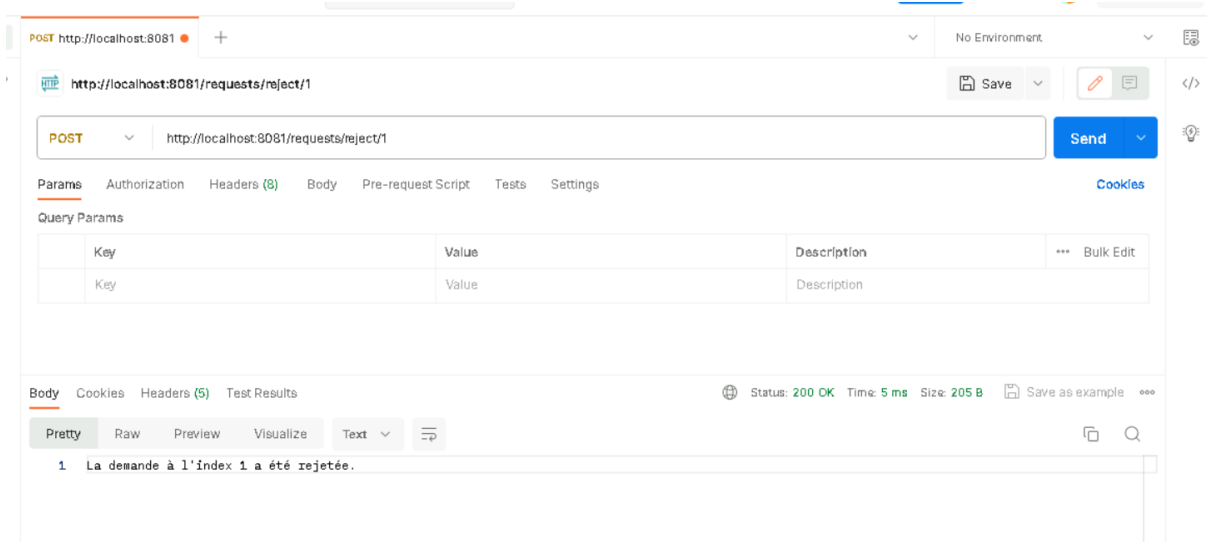


Figure 29 : Rejet de la demande d'index 1

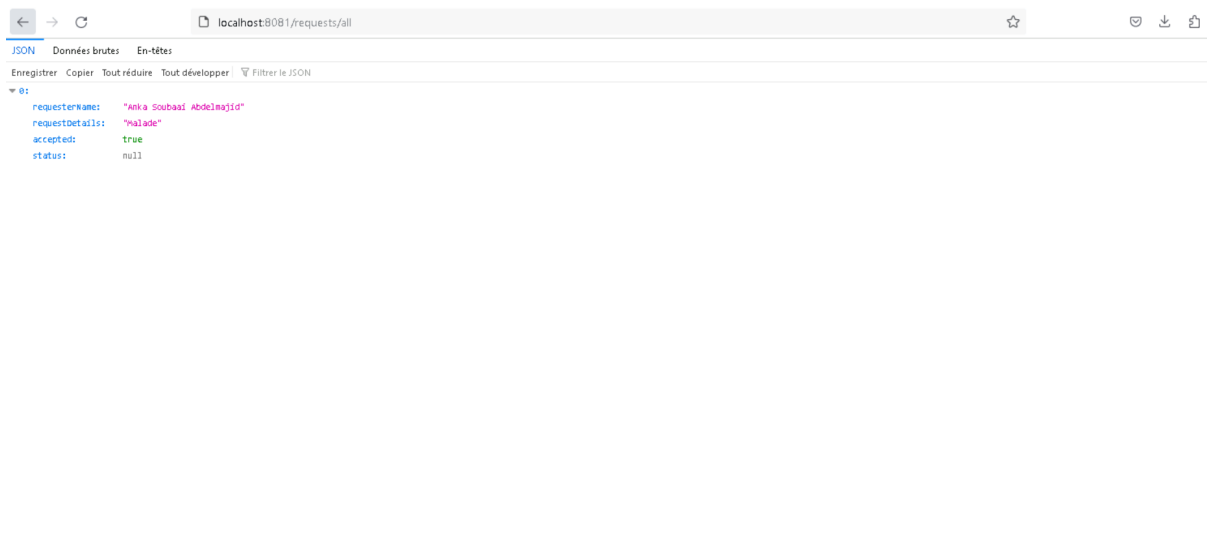


Figure 30 : Vue des demandes suite au rejet de la demande à l'index 1

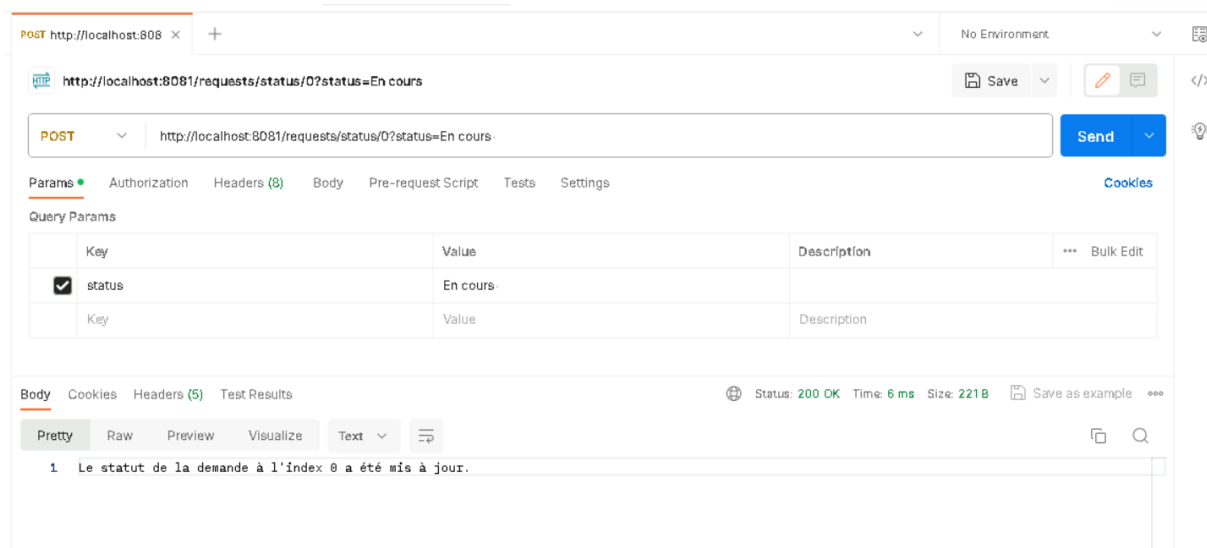


Figure 31 : Changement de status de la demande à l'index 0

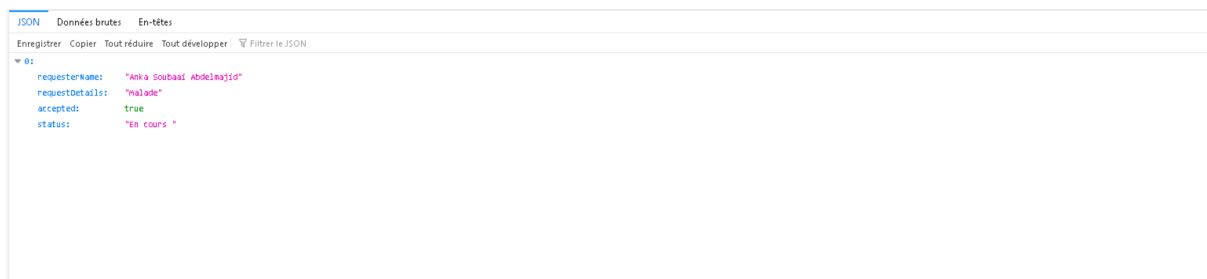


Figure 32 : Vue des demandes suite au changement de statut de la demande à l'index 0

Nos microservices travaillent en collaboration, chacun étant conçu pour une fonctionnalité spécifique. Nous avons adopté une architecture où chaque microservice est un projet autonome, permettant leur lancement indépendant. Cette approche assure une indépendance et une modularité des services, permettant leur instanciation séparée et leur fonctionnement cohérent, sans interdépendance directe.

2) Ajout du microservice de découverte :

Passer à Spring Cloud implique de rendre nos microservices découvrables, indépendamment de leur déploiement. Avant, nos appels étaient statiques, nécessitant des modifications de code en cas de changement d'adresse de déploiement. Avec Spring Cloud, nous adoptons une approche dynamique grâce à des outils comme Eureka, un service d'enregistrement offert par l'API Netflix OSS. Eureka agit comme un annuaire où les microservices s'enregistrent pour être découverts par d'autres, évitant ainsi les dépendances statiques sur les adresses de déploiement. Cette flexibilité est essentielle pour gérer la volatilité des déploiements de microservices.

La deuxième version de notre travail est disponible dans le dépôt Git sous le dossier "n". Dans cette étape, nous nous concentrons principalement sur le service permettant l'ajout d'utilisateurs demandant de l'aide, incluant la possibilité de modifier leur statut/accept/rejet , etc .

Nous débutons en définissant initialement le port et le nom de notre service auprès d'Eureka.

```
1 # Configuration du port du microservice
2 server.port=8083
3
4 # Configuration du service Eureka
5 spring.application.name=ask-help-service
6
7 |
```

Figure 33 : Configuration de notre service

The screenshot shows the Spring Eureka web interface. At the top, there's a header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. Below the header, the 'System Status' section contains two tables. The first table lists 'Environment' and 'Data center', both with 'N/A' values. The second table lists 'Current time' (2023-12-20T08:55:28 +0100), 'Uptime' (00:00), 'Lease expiration enabled' (false), 'Renews threshold' (3), and 'Renews (last min)' (0). Below this, the 'DS Replicas' section shows 'localhost'. The 'Instances currently registered with Eureka' section features a table with columns 'Application', 'AMIs', 'Availability Zones', and 'Status'. It lists one instance: 'ASK-HELP-SERVICE' with 'n/a (1)' AMIs, '(1)' Availability Zones, and a status of 'UP (1) - srv-tp01.insa-toulouse.fr:ask-help-service:8083'.

System Status	
Environment	N/A
Data center	N/A

Current time	2023-12-20T08:55:28 +0100
Uptime	00:00
Lease expiration enabled	false
Renews threshold	3
Renews (last min)	0

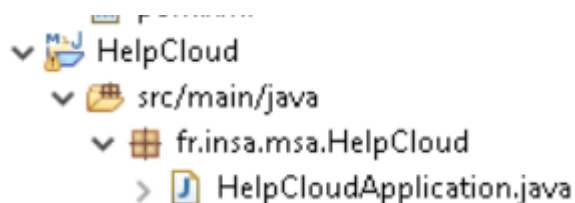
DS Replicas

localhost

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
ASK-HELP-SERVICE	n/a (1)	(1)	UP (1) - srv-tp01.insa-toulouse.fr:ask-help-service:8083

Figure 34 : Enregistrement de notre service dans Eureka

La classe qui nous permet de lancer le serveur Eureka est la suivante



La configuration de ce serveur a été définie à l'aide d'une classe principale **HelpCloudApplication**, où nous activons Eureka en utilisant les annotations Spring Boot nécessaires. De plus, nous avons configuré les paramètres du serveur Eureka via un fichier de propriétés, définissant le port de déploiement et spécifiant que les microservices ne doivent pas s'enregistrer automatiquement auprès du serveur Eureka ni récupérer automatiquement le registre comme le montre la figure suivante :

```
1 server.port=8761
2 eureka.client.register-with-eureka=false
3 eureka.client.fetch-registry=false
4
```


Pour le service AskHelp, nous avons mis en place une configuration essentielle pour son fonctionnement en tant que microservice. La classe principale `AskHelpApplication` est annotée avec `@SpringBootApplication`, et elle contient une méthode pour initialiser et gérer notre application. De plus, un `RestTemplate` a été configuré en tant que bean pour gérer les requêtes REST entre les microservices, utilisant l'annotation `@LoadBalanced` pour la répartition de charge.

Dans la classe `RequestController`, nous gérons les requêtes liées aux demandes d'aide. Chaque méthode annotée est responsable d'une action spécifique sur une demande, telles que l'acceptation, le rejet ou la modification de son statut. Ces méthodes utilisent le `RestTemplate` pour envoyer des requêtes aux autres microservices, spécifiant l'adresse des services nécessaires par leur nom de service (`ask-help-service`) plutôt que leur adresse physique. Cette abstraction permet une indépendance vis-à-vis des adresses de déploiement réelles, simplifiant ainsi l'orchestration des microservices et leur collaboration au sein de l'architecture globale.

```
@RestController
@RequestMapping("/requests")
public class RequestController {

    @Autowired
    private RestTemplate restTemplate;

    private final String SERVICE_NAME = "ask-help-service";

    private List<Request> requests = new ArrayList<>();
```

Lorsque nous exécutons les deux classes en tant qu'applications Java distinctes, cela ouvre deux consoles distinctes nous permettant de consulter les logs des requêtes qu'on envoie pour tester le fonctionnement de notre microservice (voir la section suivante pour plus de détails).

```
1 HelpCloudApplication [Java Application] U:\Windows\Bureau\Service_archi\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_16.0.2.v20210721-1149\jre\bin\javaw.exe (19 déc. 2023, 14:28:46)
2 AskHelpApplication [Java Application] U:\Windows\Bureau\Service_archi\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_16.0.2.v20210721-1149\jre\bin\javaw.exe (19 déc. 2023, 14:29:54)
```

Pour tester notre service, nous utilisons Postman à l'adresse `http://localhost:port/endpoint`. L'utilisation du nom du service n'est pas possible car celui-ci n'est connu qu'à l'intérieur d'Eureka.

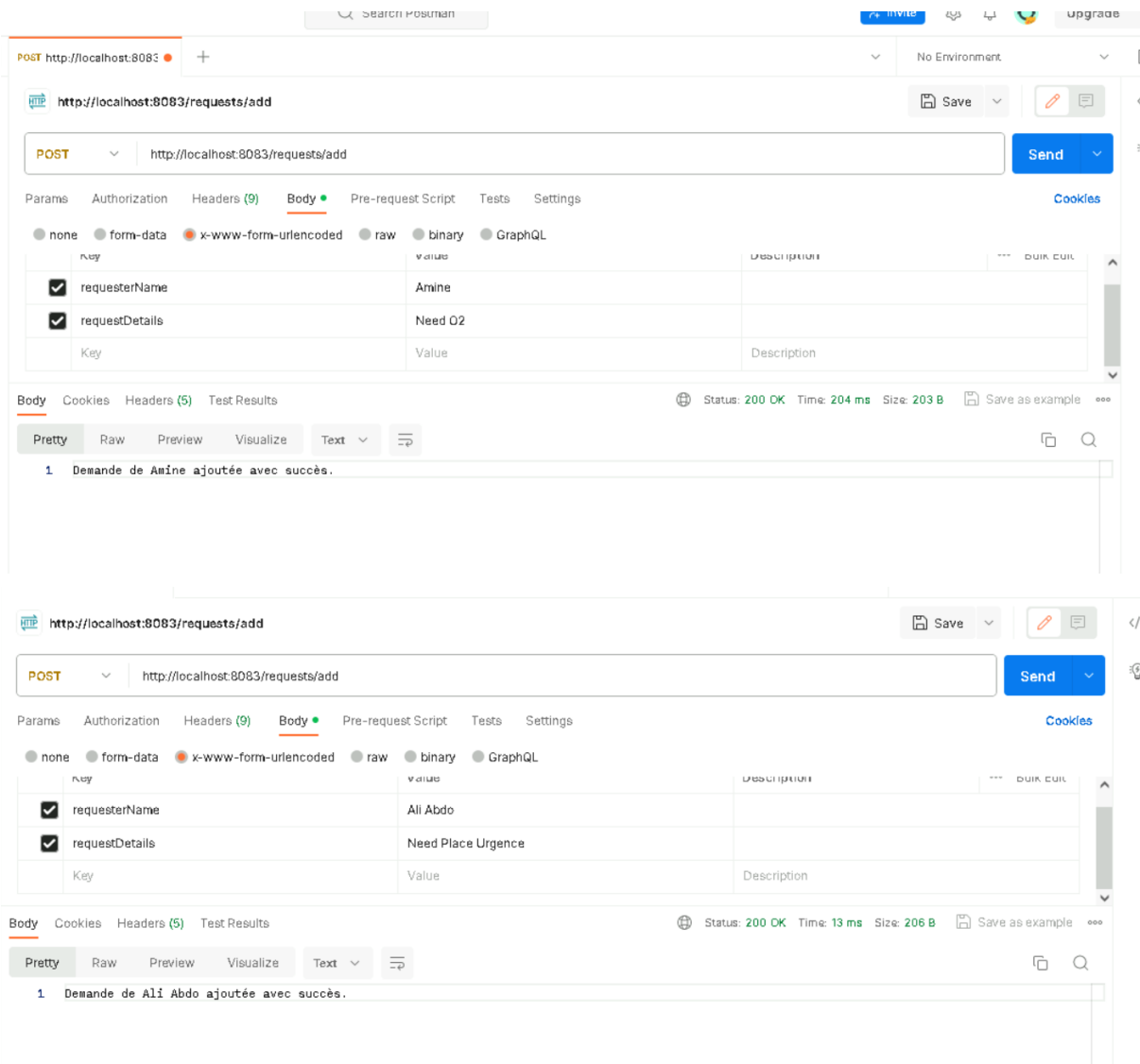


Figure 35 : Test de l'ajout d'un demandeur d'aide avec Postman

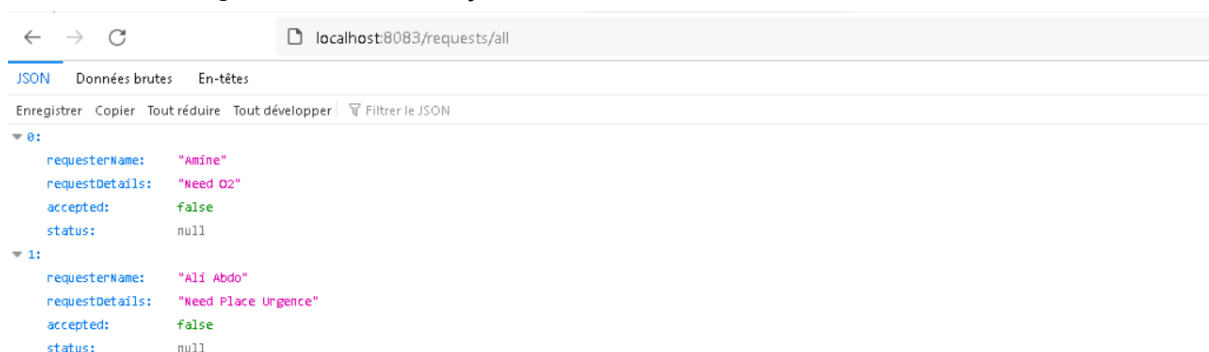


Figure 36 : Affichage de tous les demandeurs d'aide enregistrés

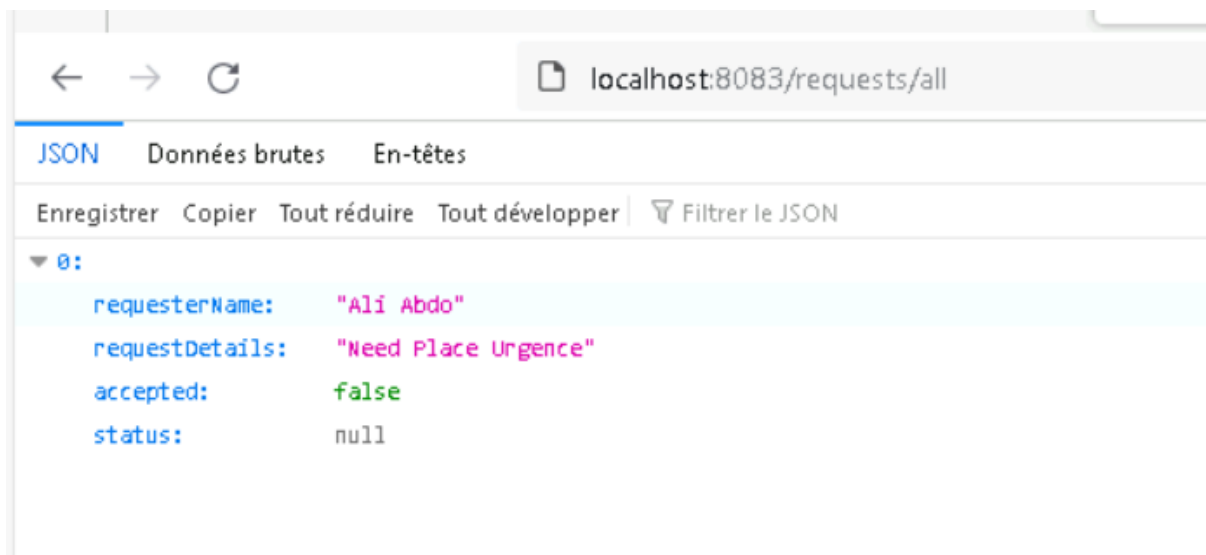
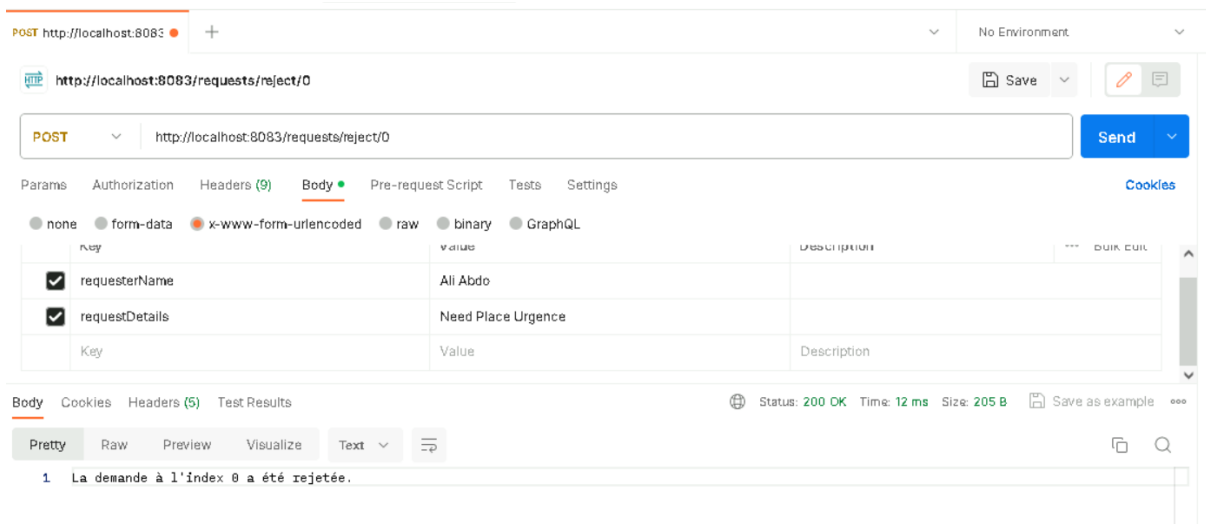


Figure 37 : Rejet de la demande à l'index 0

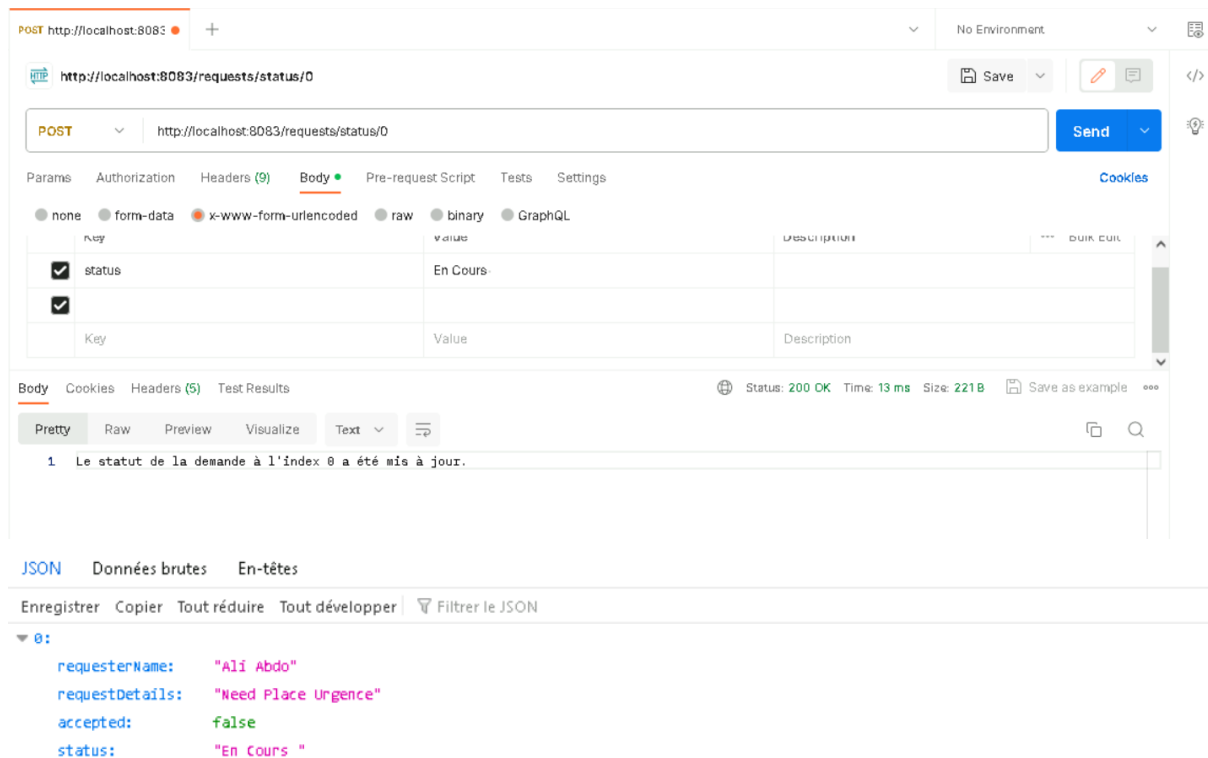


Figure 38 : Changement de status de la demande à l'index 0

3) Ajout du microservice de configuration :

En exploitant Spring Cloud, nous avons pu élaborer des fichiers de configuration spécifiques à divers environnements de développement. Cette approche permet de définir des paramètres de configuration en fonction de l'environnement utilisé, communément appelés "profils" dans Spring Cloud.

Dans le cadre de notre démarche, nous avons mis en place un serveur de configuration à l'aide de Spring Cloud. Pour ce faire, nous avons élaboré une classe Java ConfigServerApplication.

Ce code, faisant partie intégrante de notre architecture, est crucial pour notre système. En utilisant les annotations fournies par Spring, notamment **@EnableConfigServer** et **@SpringBootApplication**, nous avons configuré notre application pour qu'elle agisse comme un serveur de configuration.

La méthode main dans la classe ConfigServerApplication sert de point d'entrée de notre application. En exécutant cette classe Java, nous lançons notre serveur de configuration, permettant ainsi la gestion centralisée des configurations pour nos différents microservices.

```

package fr.insa.ms.server.config.ConfigS

import org.springframework.boot.SpringApplication
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.cloud.config.

@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {

    public static void main(String[] arg
        SpringApplication.run(ConfigServ
    }

}

```

Pour illustrer cette méthode, nous avons créé deux profils distincts : default et dev. Deux fichiers, client-service.properties et client-service-dev.properties, ont été constitués pour ces profils. Ces fichiers ont été conçus pour être utilisés ultérieurement par un microservice client nommé client-service. Le fichier client-service-dev est spécifique au profil "dev", tandis que le fichier client-service est destiné au profil par défaut.

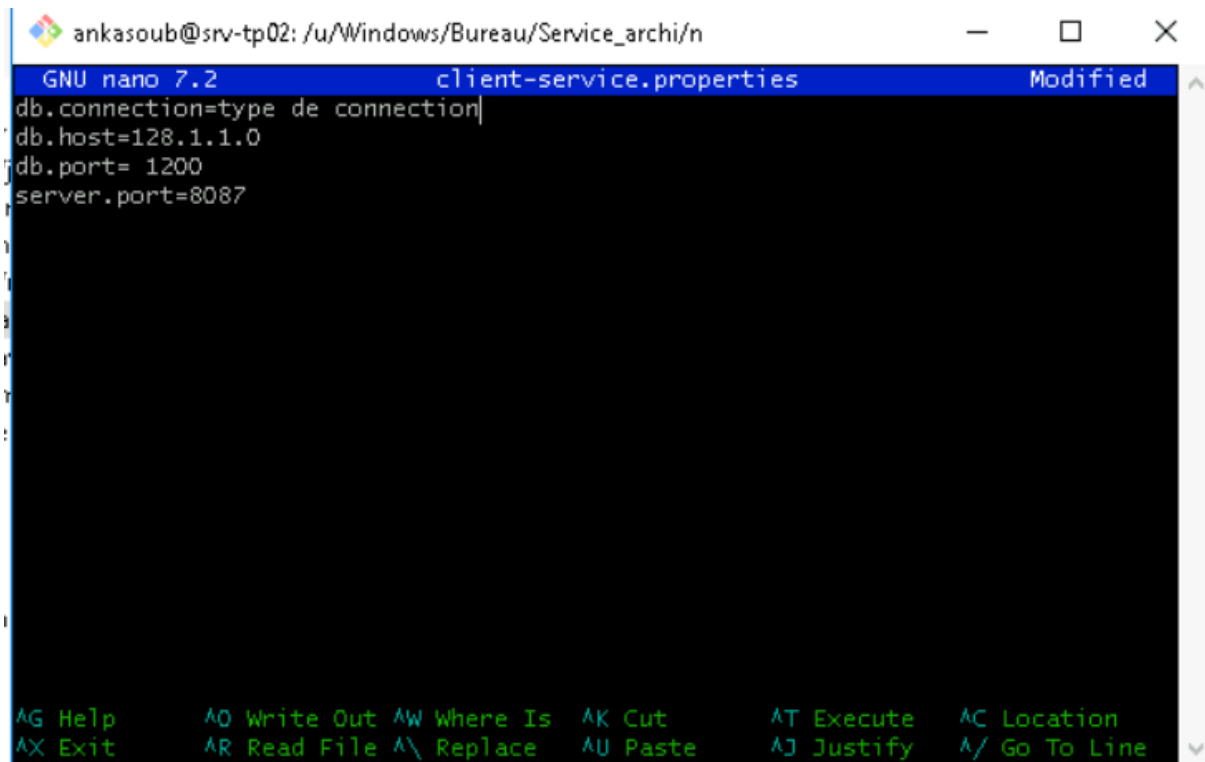
Dans notre démarche, nous avons mis en place un dépôt contenant ces deux fichiers de configuration. Ces fichiers comprennent des paramètres simulés pour des démonstrations.

```

ankasoub@srv-tp02:/u/Windows/Bureau/Service_archi/n$ git init
Initialized empty Git repository in //netapp2/ankasoub/Windows/Bureau/Service_ar
chi/n/.git/
ankasoub@srv-tp02:/u/Windows/Bureau/Service_archi/n$ |

```

Figure 39 : Création du référentiel Git pour ces deux fichiers

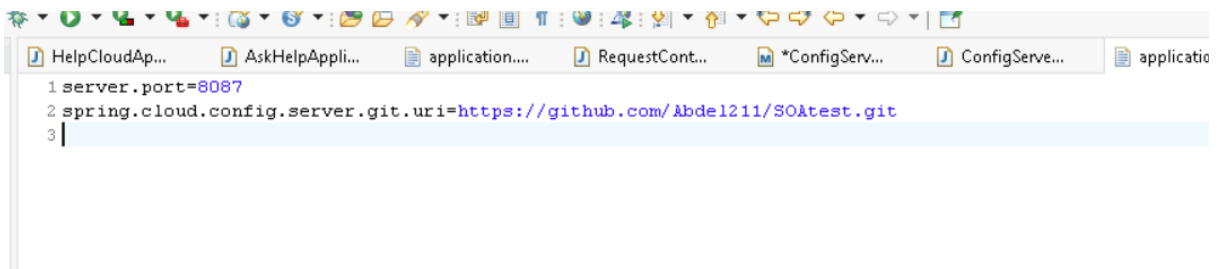


```
ankasoub@srv-tp02: /u/Windows/Bureau/Service_archi/n
GNU nano 7.2 client-service.properties Modified
db.connection=type de connection|
db.host=128.1.1.0
db.port= 1200
server.port=8087

AG Help      AO Write Out AW Where Is AK Cut      AT Execute  AC Location
AX Exit      AR Read File AL Replace  AU Paste    AJ Justify  A/ Go To Line
```

Figure 40 : Contenu d'un des fichiers de configuration

Dans le fichier de configuration `application.properties` de notre microservice, nous avons indiqué le port (ici 8087) ainsi que l'URI du dépôt Git dans lequel sont stockés nos fichiers de configuration (cf image ci dessous). Cette pratique permet de centraliser et de gérer les configurations spécifiques à chaque environnement de manière organisée et cohérente.



```
HelpCloudAp... AskHelpAppli... application.... RequestCont... *ConfigServ... ConfigServe... applicatio
1 server.port=8087
2 spring.cloud.config.server.git.uri=https://github.com/Abdel1211/SO&test.git
3
```

Nous n'avons pas eu l'opportunité de créer le client pour notre microservice. Cependant, cette absence de réalisation ne remet pas en question notre compréhension du tutoriel associé ni ce que nous avons appris pendant ce cours.

4) Bilan SOAP vs REST vs Microservices :

Lorsqu'on compare les architectures de services telles que SOAP, REST et les microservices, il est essentiel de reconnaître les avantages uniques qu'elles apportent à différents types d'applications. SOAP, avec sa structure rigide basée sur XML, offre une approche plus formelle et orientée contrat, ce qui en fait un choix pertinent pour les applications où la sécurité et la fiabilité sont primordiales. Cependant, sa complexité et sa lourdeur peuvent limiter sa flexibilité et sa facilité d'utilisation dans des environnements agiles.

De l'autre côté, REST, avec son approche plus légère et flexible, se base sur des standards web comme HTTP et JSON, offrant une adoption plus aisée et une interaction intuitive avec les ressources via des requêtes HTTP. Cela en fait un choix idéal pour les systèmes distribués et les applications orientées web où la scalabilité, la simplicité et la rapidité sont des éléments clés.

Les microservices, quant à eux, transcendent le débat REST vs SOAP en adoptant une architecture globale modulaire, fragmentant les services en composants indépendants. Cette approche favorise la scalabilité horizontale, facilite la maintenance et permet une agilité exceptionnelle en matière de développement, favorisant ainsi une évolutivité plus aisée et une gestion optimisée des ressources. Cependant, leur adoption nécessite une réflexion minutieuse sur la conception, la gestion des données et les interactions entre les services.

En somme, chacune de ces architectures offre ses propres forces et convient à des contextes spécifiques. La sélection entre SOAP, REST ou les microservices doit être guidée par les besoins de l'application, l'évolutivité future envisagée, et les priorités en matière de performance, de sécurité et de maintenabilité. Une approche mixte ou hybride peut également s'avérer efficace pour répondre à des besoins variés au sein d'une même application.

IV-Conclusion :

En rétrospective, le développement de nos microservices a été un parcours captivant et enrichissant. De la conception initiale à la mise en œuvre concrète en utilisant Spring Boot et Spring Cloud, chaque étape a été une opportunité d'apprentissage. L'implémentation de REST pour les services a offert une flexibilité remarquable, tandis que la configuration du serveur Eureka pour la découverte de services a ouvert des perspectives nouvelles en termes de gestion et d'évolutivité.

La mise en œuvre du Config Server a également marqué un tournant important, offrant une gestion centralisée des configurations, ce qui est vital pour une architecture de microservices bien gérée. Malgré le manque de temps pour achever la mise en place complète de tous les services, notre compréhension approfondie des concepts sous-jacents et des outils utilisés a été précieuse.

Ce projet a été bien plus qu'une simple application de concepts théoriques. Il a été une immersion complète dans l'écosystème des microservices, de leurs avantages à leurs défis. Chaque défi a été une opportunité de croissance, et chaque solution a ouvert de nouvelles perspectives pour des applications distribuées robustes et évolutives.