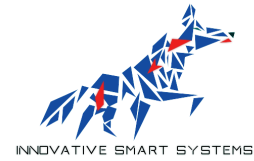




INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
TOULOUSE



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

**Benchekroun Amine
Diani Badr**

**2024/2025
5ISS**

Table de matieres:

Introduction	2
I- Architecture SOAP.....	3
II- Architecture REST.....	4
III- Architecture Microservice.....	6
a- Collaboration des 3 microservices.....	7
b- Découverte des microservices	10
c- Instanciation de microservice et répartition de charges (load balancing).....	12
d- Externalisation des configurations.....	12
IV- Mise en Place d'un Service Spring Boot sur Microsoft Azure.....	19
Conclusion.....	23

Introduction :

Les évolutions rapides des besoins et des exigences technologiques dans le domaine du développement d'applications modernes ont encouragé une exploration approfondie des différentes architectures logicielles. Dans le cadre de notre projet, axé sur la création d'une application destinée à venir en aide aux personnes en situation de vulnérabilité, nous avons été amenés à analyser et à expérimenter plusieurs solutions architecturales. Ce projet s'appuie sur une problématique majeure : de nombreuses personnes se retrouvent isolées pour diverses raisons, qu'il s'agisse d'éloignement géographique, de contraintes de santé, ou encore de longues hospitalisations.

L'objectif principal était de développer une plateforme numérique permettant de connecter les personnes ayant besoin d'un soutien avec des bénévoles prêts à leur apporter une aide. Pour répondre à ce défi, nous avons exploré trois approches architecturales majeures : SOAP, REST et les microservices. Ces solutions, bien que différentes, ont chacune apporté des réponses spécifiques à nos besoins, tout en soulevant des défis techniques et organisationnels.

Ce document retrace les différentes étapes de notre démarche, de la mise en œuvre initiale basée sur SOAP, à la transition vers REST, jusqu'à l'adoption finale d'une architecture microservices, plus agile et mieux adaptée aux exigences évolutives du projet. Chaque phase est examinée à travers ses motivations, les ajustements réalisés et l'impact des choix techniques sur le fonctionnement de l'application.

En explorant ces approches, nous mettons en perspective leurs atouts, leurs limites et leur pertinence dans le contexte de notre projet. Cette analyse vise à offrir un éclairage sur les implications pratiques de chaque architecture et sur leur capacité à répondre aux besoins spécifiques d'une application au service des personnes vulnérables.

I. Architecture SOAP

SOAP (Simple Object Access Protocol) est un protocole de communication structuré qui utilise le langage XML pour échanger des informations entre applications de manière standardisée. Ce protocole définit une enveloppe formelle pour les messages, assurant une description précise des données échangées, ce qui facilite leur interprétation par des systèmes variés.

Les messages SOAP sont généralement transmis via des protocoles de transport comme HTTP ou SMTP, ce qui permet leur intégration dans des environnements réseau existants. Grâce à sa nature indépendante des langages de programmation et des plates-formes, SOAP garantit une interopérabilité élevée, rendant possible la communication entre des systèmes différents.

En outre, SOAP est extensible et peut intégrer des fonctionnalités avancées, telles que la sécurité ou la gestion des erreurs, ce qui en fait un choix adapté pour des systèmes complexes nécessitant une communication fiable et rigoureuse.

Concernant notre architecture, nous avons créé 5 classes :

Classes	Description
User	Structure de base représentant un utilisateur, permettant de définir et d'accéder au nom et à l'identifiant. Extensible pour ajouter des fonctionnalités spécifiques comme dans <i>Admin</i> .
UserService	Fournit une interface web pour gérer les utilisateurs (demandeurs d'aide, bénévoles, administrateurs). Permet d'ajouter des utilisateurs et de confirmer leurs identifiants via console.
Admin	Représente les administrateurs en ajoutant une clé d'administration à l'utilisateur générique. Permet leur intégration dans le système de l'application
Request	Gère les informations et le suivi des requêtes. Les statuts possibles incluent : "EN ATTENTE," "VALIDÉE," "REJETÉE," "CHOISIE," et "RÉALISÉE."
RequestService	Propose des fonctionnalités pour administrer et suivre les requêtes, facilitant la gestion des demandes au sein de l'application.

Pour tester notre architecture, nous utilisons un navigateur web pour accéder aux services publiés. Le service UserService est disponible à l'adresse suivante : `http://localhost:8809/UserService`, tandis que le service RequestService peut être consulté à l'adresse : `http://localhost:8809/RequestService`. Ces adresses permettent de vérifier le bon fonctionnement des services déployés et de tester les fonctionnalités associées via une interface utilisateur ou des outils d'appel d'API.

```

Main [Java Application]
WARNING: All illegal access operations will be denied in a future release
déc. 05, 2024 12:03:44 PM com.sun.xml.ws.server.MonitorBase createRoot
INFO: Metro monitoring rootname successfully set to: com.sun.metro:pp=/,type=WSEndpoint,name=UserServiceService-UserServicePor
UserService SOAP service published at: http://localhost:8809/UserService
déc. 05, 2024 12:03:44 PM com.sun.xml.ws.server.MonitorBase createRoot
INFO: Metro monitoring rootname successfully set to: com.sun.metro:pp=/,type=WSEndpoint,name=RequestServiceService-RequestServ
RequestService SOAP service published at: http://localhost:8809/RequestService
User asking for help added :Amine4
  
```

En utilisant le Web Explorer, nous avons observé l'ajout d'un utilisateur nommé Amine, associé à l'identifiant 4.

Body

addRequest

arg0 Add Remove

Content
<div>status string Add Remove</div> <div> <div>WAITING</div> </div> <div>userid int</div> <div>4</div>

Go Reset

Status

return

status (string): WAITING

userid (int): 4

On voit dans Status que la requête est bien ajoutée.

L'adoption de l'architecture SOAP a permis d'établir une communication sécurisée et structurée entre le client et le serveur, grâce à l'interopérabilité assurée par les messages XML. Parmi ses avantages, on peut citer une description précise des services via le fichier WSDL, ainsi qu'une sécurité accrue grâce à WS-Security. Cependant, cette approche présente des limites, notamment une certaine complexité et rigidité, ainsi qu'un surcoût potentiel en termes de bande passante en raison de la structure XML des messages. Bien que SOAP offre de nombreux bénéfices, sa complexité peut représenter un frein dans des scénarios où une plus grande flexibilité est requise.

II. Architecture REST

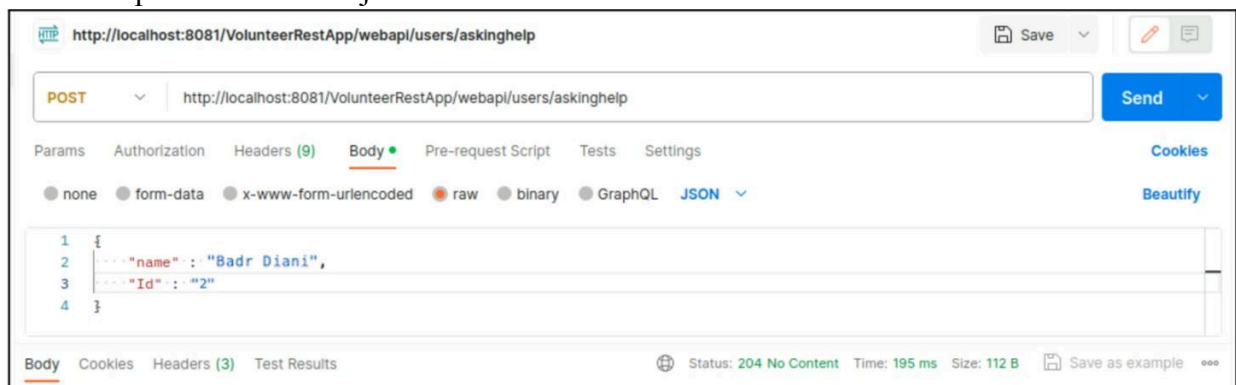
L'architecture REST (Representational State Transfer) est un style architectural largement adopté pour les systèmes web, caractérisé par sa simplicité et son efficacité. Elle repose sur des URI pour représenter et adresser des ressources, ainsi que sur des opérations standardisées telles que GET, POST, PUT et DELETE, réalisées via des requêtes HTTP. Grâce à son utilisation de formats de représentation universels comme JSON ou XML, REST garantit une interopérabilité

optimale entre les systèmes, en faisant une approche de choix pour le développement d'API web modernes et robustes.

Dans le cadre de notre transformation technologique, nous avons migré de notre ancienne architecture SOAP, souvent jugée complexe en raison de sa dépendance au XML, vers une architecture REST, plus flexible et adaptée aux besoins actuels. Cette migration a impliqué une redéfinition des interactions entre nos services : les ressources sont désormais identifiées et accessibles via des URI dédiées, tandis que les opérations CRUD (Create, Read, Update, Delete) s'alignent naturellement sur les verbes HTTP appropriés. L'utilisation de JSON comme format de représentation par défaut a considérablement simplifié les échanges de données entre les différents composants de notre application.

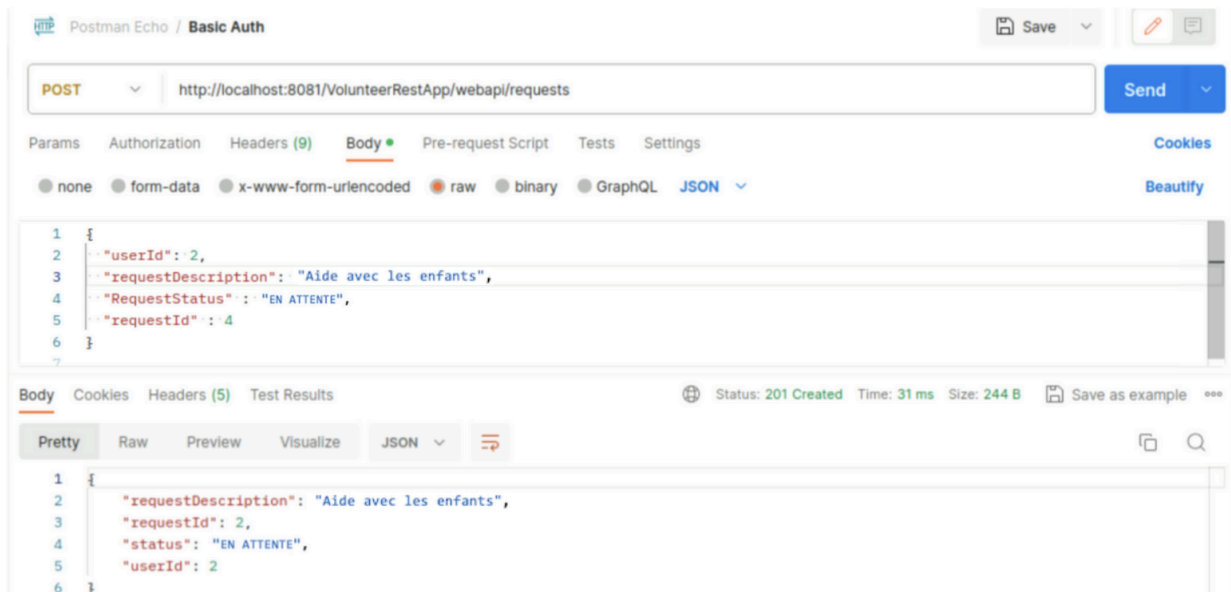
Sur le plan technique, bien que les classes de notre architecture soient restées inchangées, les annotations ont été adaptées pour correspondre à REST : des annotations telles que `@POST`, `@GET`, `@PUT` et des définitions de chemins comme `@Path("/user")` ont été intégrées. Cette transition vers REST a permis de moderniser notre système tout en améliorant sa scalabilité, sa performance et son adaptabilité face aux évolutions technologiques. En conséquence, notre plateforme est désormais mieux équipée pour répondre aux défis et opportunités de notre environnement en constante mutation.

Nous avons testé notre architecture en effectuant plusieurs requêtes via Postman. La requête POST permettant d'ajouter un `aidSeeker` s'est exécutée avec succès:

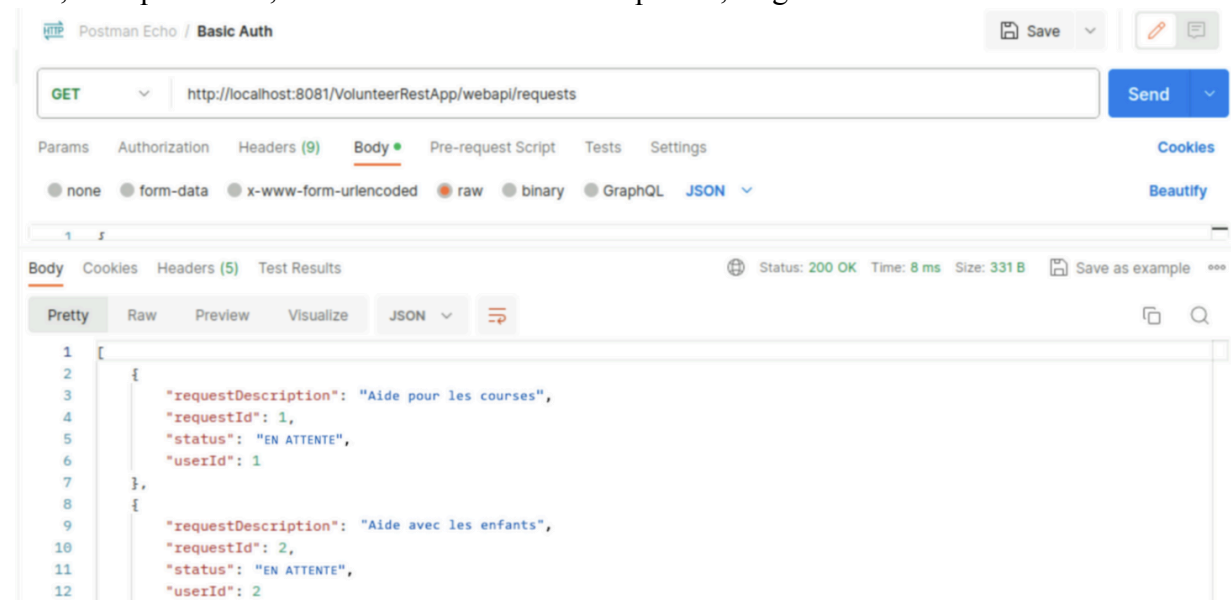


```
INFO: Au moins un fichier JAR a été analysé pour trouver des TLDs mais il n'en contenait pas, le mode "debug" du journal peut
déc. 10, 2024 16:59:20 PM org.apache.catalina.core.StandardContext reload
INFO: Le rechargement de ce contexte est terminé
User asking for help added: Badr Diani0
```

De même, la requête POST visant à ajouter une "request" a été validée sans problème:



Enfin, la requête GET, destinée à retourner les "requests", a également fonctionné correctement:



Ces résultats confirment le bon fonctionnement des principales fonctionnalités testées.

Une analyse comparative des modèles SOAP et REST met en évidence des différences essentielles. SOAP se distingue par sa robustesse et ses fonctionnalités avancées en matière de sécurité, mais il est également plus complexe et peut afficher des performances moins rapides. À l'inverse, REST adopte une approche plus légère et simplifiée, offrant ainsi des performances optimisées, bien qu'il soit moins équipé en termes de sécurité avancée. Le choix entre ces deux modèles dépend des besoins spécifiques du projet : SOAP est idéal pour des environnements exigeant une sécurité renforcée et une grande robustesse, tandis que REST s'impose pour les applications où la simplicité et la rapidité sont prioritaires.

III. Architecture Microservice

Évolution vers une architecture microservices pour surmonter les limites du REST

Jusqu'à présent, notre système reposait sur une architecture REST, s'appuyant sur la gestion des

requêtes HTTP. Bien que cette approche ait prouvé son efficacité, elle présente certaines limites, notamment une complexité croissante dans la coordination des interactions entre diverses ressources. Pour répondre à ces défis et améliorer notre système, nous avons décidé de faire évoluer notre architecture vers un modèle basé sur les microservices.

Qu'est-ce qu'une architecture microservices ?

L'architecture microservices repose sur la division d'une application en une collection de services indépendants, appelés microservices. Chaque service est conçu pour répondre à une tâche ou fonctionnalité spécifique et communique avec les autres via des API clairement définies. Ce découpage modulaire permet de rendre les applications plus flexibles, évolutives et faciles à maintenir. En adoptant ce modèle, nous visons à simplifier la maintenance, répondre efficacement aux besoins évolutifs de notre application, et faciliter son adaptation à des charges de travail croissantes.

Avantages clés des microservices

- **Indépendance et autonomie** : Chaque microservice peut être développé, déployé et mis à l'échelle de manière indépendante, ce qui accélère les mises à jour et les corrections.
- **Flexibilité technologique** : Les microservices permettent d'utiliser les outils et langages les mieux adaptés à leurs besoins spécifiques, offrant ainsi une grande liberté dans le choix des technologies.
- **Résilience et tolérance aux pannes** : Une défaillance dans un microservice n'affecte pas nécessairement l'ensemble du système, ce qui renforce sa stabilité globale.
- **Facilité d'évolution** : Grâce à leur modularité, les microservices permettent d'ajouter de nouvelles fonctionnalités ou d'adapter celles existantes sans impacter le reste de l'application.

Pourquoi adopter cette transition ?

Cette transition vers une architecture microservices répond à notre ambition d'améliorer la flexibilité, l'évolutivité et la robustesse de notre système. En rationalisant la gestion des interactions et en réduisant les interdépendances, elle simplifie non seulement la maintenance, mais également l'ajout de nouvelles fonctionnalités. Nous serons ainsi mieux armés pour relever les défis techniques et accompagner la croissance continue de notre application.

a. Collaboration des 3 microservices

Microservice	Rôle	Entrée	Sortie
UserListService	Obtenir la liste complète des utilisateurs selon leur rôle (admin, volunteer, aidseeker).	Rôle (e.g., admin)	Liste des utilisateurs (id)
UserManagement	Récupérer les informations d'un utilisateur donné.	Identifiant (id)	Nom de l'utilisateur

RequestManagement	Fournir la description d'une requête effectuée par un utilisateur (aidesoignant).	Identifiant (id)	Description de la requête
-------------------	---	------------------	---------------------------

Dans cette première étape, il s'agit de développer trois microservices indépendants, chacun avec un rôle spécifique au sein de l'architecture.

Le microservice UserListResource a pour mission de :

1. Récupérer une liste d'identifiants d'utilisateurs en fonction de leur rôle.
2. Pour chaque identifiant, interroger le microservice UserManagement afin d'obtenir les informations détaillées sur l'utilisateur.
3. Ensuite, faire appel au microservice RequestController pour récupérer la description de la demande associée à cet utilisateur.
4. Enfin, compiler les informations collectées (détails de l'utilisateur et description de la demande) pour produire un résultat final structuré.

La collaboration entre les différents microservices sera testée à l'aide d'une base de données qui suit:

- Pour les utilisateurs :

```
List<User> userInfos = Arrays.asList(
    new User( id: 0, name: "Amine" ),
    new User( id: 1, name: "Badr" ),
    new User( id: 2, name: "Ahmed" ),
    new User( id: 3, name: "Quentin" )
);
```

- Pour les requêtes :


```
List<Request> requestList= Arrays.asList(
    new Request( requestId: 0, userId: 0, requestDescription: "aide pour les courses", "EN ATTENTE"),
    new Request( requestId: 1, userId: 1, requestDescription: "aide avec les enfants", "VALIDÉE"),
    new Request( requestId: 2, userId: 2, requestDescription: "aide pour les études", "REJETÉE"),
    new Request( requestId: 3, userId: 3, requestDescription: "aide pour déménager", "EN ATTENTE")
);
```

- Pour simuler les rôles : (un utilisateur peut avoir à la fois 2 rôles)

```
private List<Integer> getUserIdsByRole(String role) {
    switch (role.toLowerCase()) {
        case "admin":
            return List.of(0, 2);
        case "aidseeker":
            return List.of(2, 3);
        case "volunteer":
            return List.of(1, 3);
        default:
            return List.of();
    }
}
```

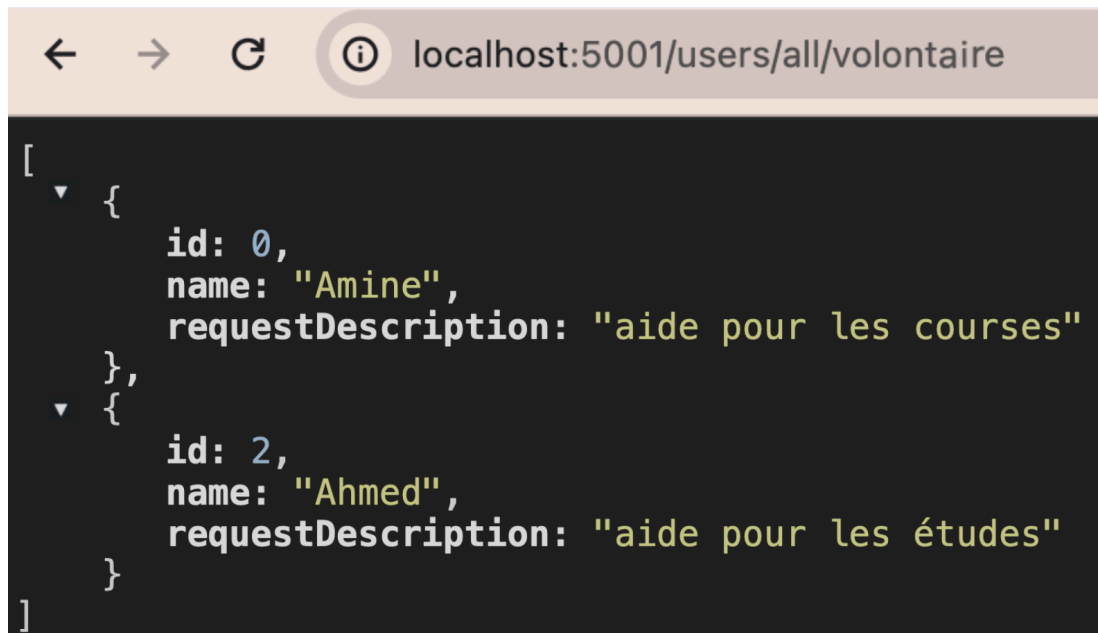
Voila les resultats :

- Pour les admins :



```
[
  {
    id: 0,
    name: "Amine",
    requestDescription: "aide pour les courses"
  },
  {
    id: 1,
    name: "Badr",
    requestDescription: "aide avec les enfants"
  },
  {
    id: 2,
    name: "Ahmed",
    requestDescription: "aide pour les études"
  }
]
```

- Pour les volunteers :



```
[
  {
    id: 0,
    name: "Amine",
    requestDescription: "aide pour les courses"
  },
  {
    id: 2,
    name: "Ahmed",
    requestDescription: "aide pour les études"
  }
]
```

- Pour aidSeeker :



```
[
  {
    id: 1,
    name: "Badr",
    requestDescription: "aide avec les enfants"
  },
  {
    id: 3,
    name: "Quentin",
    requestDescription: "aide pour déménager"
  }
]
```

Nous concluons que la collaboration entre les trois microservices fonctionne correctement.

b. Découverte des microservices

Dans cette section, l'objectif est de développer des microservices capables de se découvrir mutuellement à l'aide de Spring Cloud.

Configuration du serveur Eureka : La première étape consiste à créer un projet Spring Boot et à y intégrer les dépendances nécessaires pour configurer un serveur Eureka. Ce serveur agira comme un registre central permettant de découvrir et d'enregistrer les microservices.

Publication des microservices : Une fois le serveur Eureka opérationnel, les trois microservices

précédemment développés doivent être publiés et enregistrés dans le registre Eureka, afin qu'ils puissent interagir de manière transparente au sein de l'écosystème distribué.

Ce processus garantit une architecture orientée services où chaque composant est découvrable dynamiquement.

The screenshot shows the Spring Eureka web interface. At the top, there's a header with the 'spring Eureka' logo and navigation links 'HOME' and 'LAST 1000 SINCE STARTUP'. Below the header, the 'System Status' section displays two columns of information. The left column shows 'Environment: test' and 'Data center: default'. The right column shows 'Current time: 2024-12-16T22:15:23 +0100', 'Uptime: 00:44', 'Lease expiration enabled: true', 'Renews threshold: 6', and 'Renews (last min): 8'. Below this, the 'DS Replicas' section shows 'localhost'. The 'Instances currently registered with Eureka' section contains a table with the following data:

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 désormais enregistrés et accessibles via le service de découverte Eureka, permettant leur identification par d'autres services clients.

Dans le cadre de l'intégration d'Eureka au sein du microservice UserListService, plusieurs ajustements ont été effectués pour abstraire les appels aux microservices de leurs adresses de déploiement réelles. Voici les principales étapes de cette implémentation :

- **Création d'une instance unique de RestTemplate :**
Une méthode annotée avec `@Bean` a été ajoutée dans la classe principale pour générer une instance unique de RestTemplate lors du démarrage de l'application. Cette instance est enrichie de l'annotation `@LoadBalanced`, indiquant qu'elle doit tirer parti de la fonctionnalité de découverte de services fournie par Eureka.
- **Injection de RestTemplate dans le microservice :**
Dans la classe du microservice UserListResource, une variable RestTemplate a été déclarée et annotée avec `@Autowired`. Cela permet à Spring Boot d'injecter automatiquement l'instance configurée de RestTemplate lorsque nécessaire.
- **Adaptation des appels aux microservices :**
Les appels directs utilisant des adresses physiques ont été remplacés par des requêtes basées sur les noms des services enregistrés dans Eureka, renforçant ainsi la flexibilité et la robustesse du système.

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

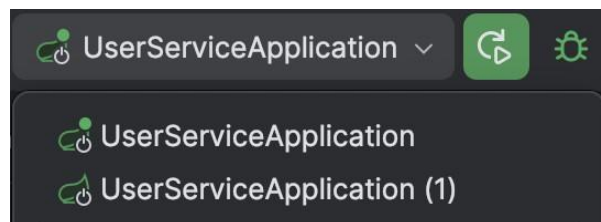
// Call RequestResource microservice to get request description
Request request = restTemplate.getForObject( url: "http://requestManagement/request/" + userId, Request.class);
```

Ces modifications assurent une interaction transparente et dynamique entre les différents microservices au sein de l'écosystème distribué.

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

Dans cette section, l'objectif est de déployer plusieurs instances du microservice UserManagement et de configurer une répartition de charge efficace à l'aide des outils fournis par Spring Cloud. Cela permettra d'améliorer la scalabilité et la disponibilité du service tout en garantissant une distribution équilibrée des requêtes entre les différentes instances.

On ajoute la configuration suivante:



Deux instances du microservice UserManagement sont désormais actives : l'une est déployée sur le port 8081, tandis que l'autre fonctionne sur le port 8181.

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 :

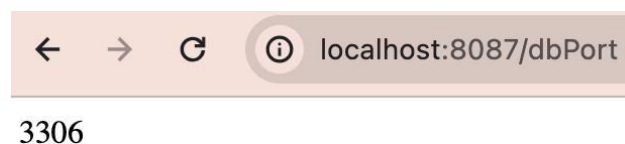


```
1 db.host=srv-bdens.insa-toulouse.fr
2 db.port=3306
3 db.name=projet_gei_072
4 db.login=projet_gei_072
5 db.pwd=eiKie7Ah
```

On a de même changé le fichier applications properties :

```
server.port=8888
spring.cloud.config.server.git.uri= https://github.com/amineben0712/Archi_Config/blob/main/client-service-dev.properties
```

Une fois la configuration terminée, l'accès à l'URL <http://localhost:8087/dbHost> permet de vérifier que le serveur est correctement détecté, ainsi que le port et les autres paramètres associés.



Nous allons maintenant externaliser les paramètres de configuration des microservices UserManagement et RequestManagement. Ces services seront connectés à une base de données relationnelle, et nous allons simuler la récupération des paramètres de connexion à cette base.

Étapes de mise en œuvre :

- **Ajout des dépendances :** Les dépendances du client Eureka doivent être ajoutées au fichier `pom.xml` des services afin qu'ils soient découvrables et accessibles depuis les autres microservices.
- **Mise à jour des contrôleurs :** Les contrôleurs des microservices UserManagement et RequestManagement seront modifiés pour intégrer la gestion des paramètres de configuration et simuler l'accès à la base de données.

Remarque :

Nous avons créé les classes Volunteer et AidSeeker, qui auraient pu être utilisées à la place de la classe User pour représenter des entités spécifiques, selon les besoins du projet.

UserController :

```

@GetMapping("/dbInfo")
public String getDbInfo() {
    String dbHost = restTemplate.getForObject("http://client-service/dbHost", String.class);
    String dbPort = restTemplate.getForObject("http://client-service/dbPort", String.class);

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

@PostMapping(value = "/idUser/{nom}", produces = MediaType.APPLICATION_JSON_VALUE)
public User addUser(
    @PathVariable int idUser,
    @PathVariable String nom,
    @RequestParam String role) {

    String dbHost = restTemplate.getForObject("http://client-service/dbHost", String.class);
    String dbPort = restTemplate.getForObject("http://client-service/dbPort", String.class);

    User user = new User(idUser, nom);

    String sql = "INSERT INTO users(nom, role) VALUES (?, ?)";
    try (Connection conn = DriverManager.getConnection("jdbc:mysql://" + dbHost + ":" + dbPort, "projet_gei_072",
        PreparedStatement stmt = conn.prepareStatement(sql);

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

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

    return user;
}

```

Cette méthode prend en charge la création d'un nouvel utilisateur en lui attribuant un rôle spécifique (admin, volunteer ou aidSeeker). Elle effectue des appels au serveur client-service pour récupérer les paramètres de la base de données, insère les informations de l'utilisateur dans une table users (composée de trois colonnes : id, nom et rôle), et retourne une réponse appropriée en fonction du succès ou de l'échec de l'opération.

Requêtes POST depuis POSTMAN :

The screenshot shows a POSTMAN interface for a POST request to the URL `http://localhost:8081/user/1/Amine?role=aidesoignant`. The 'Params' tab is active, showing a 'Query Params' table with one entry: 'role' with the value 'aidesoignant'. The status bar at the bottom indicates a successful response with status '200 OK', a response time of '1485 ms', and a content size of '187 B'.

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

Body Cookies Headers Test Results 200 OK 1485 ms 187 B

POST ▼ `http://localhost:8081/user/1/Ahmed?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 485 ms 187 B

Les requêtes s'exécutent avec succès, retournant un message de confirmation : 200 OK.

```

benchekroun@insa-11284: ~/Bureau
Fichier Édition Affichage Recherche Terminal Aide

mysql> SELECT * FROM users;
+----+-----+-----+
id | nom   | role      |
+----+-----+-----+
1  | Amine | aidesoignant |
2  | Badr  | admin      |
3  | Ahmed | admin      |
4  | Quentin | volontaire |
+----+-----+-----+
rows in set (0.00 sec)

```

On voit que les données ont été ajoutées avec succès à la base de données.

RequestController :


```

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

    try (Connection conn = DriverManager.getConnection(
        "jdbc:mysql://" + dbHost + ":" + dbPort + "/projet_gei_072",
        "projet_gei_072", "eikie7Ah")) {
        Statement stmt1 = conn.createStatement();
        ResultSet rs = stmt1.executeQuery(
            "SELECT role FROM users WHERE id = " + seeker);

        while (rs.next()) {
            String role = rs.getString("role");
            if ("aidSeeker".equals(role)) {
                PreparedStatement stmt = conn.prepareStatement(
                    "INSERT INTO request(seeker, request_description, status) VALUES (?, ?, 'WAITING')");
                stmt.setInt(1, seeker);
                stmt.setString(2, requestDescription);
                stmt.executeUpdate();
                System.out.println("aidesoignant -> création de requête");
            } else {
                System.out.println("pas un aide soignant");
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return new Request();
}

```

Cette méthode permet d'ajouter une demande d'aide. Le microservice commence par interroger la base de données pour vérifier si l'ID fourni en paramètre correspond à un utilisateur ayant le rôle AidSeeker. Si la vérification est positive, la demande est ajoutée à la table request, qui contient les colonnes suivantes : request_id, seeker, volunteer, request_description et status. Dans le cas contraire, un message d'erreur "Not an AidSeeker" est retourné.

```

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

    try (Connection conn = DriverManager.getConnection(
        "jdbc:mysql://" + dbHost + ":" + dbPort + "/projet_gei_072",
        "projet_gei_072", "eikiei7Ah")) {
        Statement stmt1 = conn.createStatement();
        ResultSet rs = stmt1.executeQuery(
            "SELECT role FROM users WHERE id = " + volantaire);

        while (rs.next()) {
            String role = rs.getString("role");
            if ("volantaire".equals(role)) {
                PreparedStatement stmt = conn.prepareStatement(
                    "INSERT INTO request(volantaire, request_description, status) VALUES (?, ?, 'en attente')");
                stmt.setInt(1, volantaire);
                stmt.setString(2, requestDescription);
                stmt.executeUpdate();
                System.out.println("volantaire -> offre de l'aide");
            } else {
                System.out.println("non volantaire");
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return new Request();
}

```

Cette méthode fonctionne selon une logique similaire à la précédente, mais elle permet à un volontaire d'offrir son aide.

Requêtes POST depuis POSTMAN :

The screenshot shows the Postman interface for a POST request. The URL is `http://localhost:8082/request/addRequest/5`. The 'Body' tab is selected, and the 'form-data' radio button is chosen. A table of key-value pairs is visible, with 'requestDescription' set to 'souhaite adhérer pour aider'. The status bar at the bottom indicates a successful response: '200 OK 1357 ms 251 B'.

	Key	Value	Description
<input type="checkbox"/>		Text	
<input checked="" type="checkbox"/>	requestDescription	souhaite adhérer pour aider	

POST ▼ `http://localhost:8082/request/offerHelp/2`

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

☐ none ☒ form-data ☐ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

	Key		Value	Description
<input type="checkbox"/>		Text ▼		
<input checked="" type="checkbox"/>	requestDescription	Text ▼	souhaite adhérer pour aider	

Body Cookies Headers Test Results 200 OK 557 ms 251 B

Les requêtes s'exécutent avec succès, retournant un message de confirmation : 200 OK.

```

2024-12-19T21:49:19.404+01:00 INFO 33868 --- [requestManagement] [nio-8082-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring Disp
2024-12-19T21:49:19.404+01:00 INFO 33868 --- [requestManagement] [nio-8082-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'di
2024-12-19T21:49:19.405+01:00 INFO 33868 --- [requestManagement] [nio-8082-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization
AideSoignant -> création de requête
Volantaire -> offre de l'aide

```

Les opérations de création de demandes et d'offre d'aide ont été effectuées avec succès.

```

mysql> SELECT * FROM requests;
+-----+-----+-----+-----+-----+
| request_id | seeker | volontaire | request_description | statut |
+-----+-----+-----+-----+-----+
| 2 | 4 | NULL | NULL | 0 |
| 4 | 5 | NULL | souhaite adhérer pour aider | en attente |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

```

Les demandes ont été ajoutées avec succès à la table des requêtes.

Si l'id ne correspond pas dans la table des users on le resultat suivant:

```

2024-12-19T21:49:19.404+01:00 INFO 33868 --- [requestManagement] [nio-8082-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring Disp
2024-12-19T21:49:19.404+01:00 INFO 33868 --- [requestManagement] [nio-8082-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'di
2024-12-19T21:49:19.405+01:00 INFO 33868 --- [requestManagement] [nio-8082-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization
AideSoignant -> création de requête
Volantaire -> offre de l'aide
non volontaire
pas un aide soignant

```

Bilan SOAP vs REST vs Microservices:

Lorsqu'on compare les architectures de services telles que SOAP, REST et les microservices, chaque approche répond à des besoins spécifiques et présente des avantages distincts. SOAP, grâce à sa structure rigoureuse basée sur XML et son orientation contractuelle, est idéal pour les applications où la sécurité et la fiabilité sont des priorités. Cependant, sa complexité et sa lourdeur peuvent poser des défis dans des environnements nécessitant une grande flexibilité.

À l'inverse, REST, avec son approche légère et flexible, s'appuie sur des standards web comme HTTP et JSON. Cela facilite son adoption et offre une interaction intuitive avec les ressources,

le rendant particulièrement adapté aux systèmes distribués et aux applications orientées web qui nécessitent scalabilité, simplicité et rapidité.

Les microservices transcendent le débat REST vs SOAP en adoptant une architecture modulaire. En fragmentant les services en composants indépendants, cette approche favorise la scalabilité horizontale, simplifie la maintenance et permet une agilité accrue dans le développement. Cependant, leur mise en œuvre nécessite une conception réfléchie et une gestion optimisée des interactions entre les services.

En conclusion, chaque architecture a ses points forts et ses limites. Le choix entre SOAP, REST ou les microservices dépend des exigences spécifiques du projet en termes de performance, de sécurité, de flexibilité et de maintenance. Bien que nous ayons acquis une compréhension approfondie de ces concepts, le manque de temps nous a empêchés de finaliser certaines étapes, notamment le déploiement complet de notre solution. Cependant, les bases établies permettront de poursuivre ce travail dans des contextes futurs similaires.

IV. Mise en Place d'un Service Spring Boot sur Microsoft Azure

Dans le cadre de notre projet, nous avons suivi un tutoriel détaillant les étapes nécessaires pour déployer une application Spring Boot sur Microsoft Azure. Ce tutoriel combine deux outils essentiels : GitHub Actions pour l'intégration et le déploiement continus, et Microsoft Azure comme plateforme d'hébergement. L'objectif principal de ce travail était de comprendre et de mettre en œuvre un pipeline CI/CD (Continuous Integration/Continuous Deployment) automatisé, permettant un déploiement fluide et rapide d'un microservice.

Ce chapitre revient sur les différentes étapes réalisées : depuis la création d'un compte Azure et l'utilisation des groupes de ressources, jusqu'à la configuration du centre de déploiement, en passant par l'écriture et l'utilisation d'un workflow YAML pour automatiser le processus. Nous mettons également en avant les points clés appris durant cette expérience, tels que la gestion des permissions entre GitHub et Azure, et l'importance de la structuration des workflows dans un contexte de déploiement.

[De base](#) [Déploiement](#) [Réseau](#) [Superviser + sécuriser](#) [Balises](#) [Vérifier + créer](#)

App Service Web Apps vous permet de créer, de déployer et de mettre à l'échelle des applications d'API, web et mobiles de classe Entreprise exécutées sur n'importe quelle plateforme rapidement. Respectez les exigences strictes en termes de performances, de scalabilité, de sécurité et de conformité lors de l'utilisation d'une plateforme complètement managée pour effectuer la maintenance de l'infrastructure. [En savoir plus](#)

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.

Abonnement * ⓘ

Groupe de ressources * ⓘ

[Créer nouveau](#)

Détails de l'instance

Nom ✓
-aggfhsazdtfwhbdr.canadacentral-01.azurewebsites.net

☒ Nom d'hôte unique par défaut (préversion) activé. [Découvrir plus d'informations sur cette mise à jour](#)

Publier * ☒ Code ☐ Conteneur

Pile d'exécution *

Pile serveur web Java *

Système d'exploitation * ☒ Linux ☐ Windows

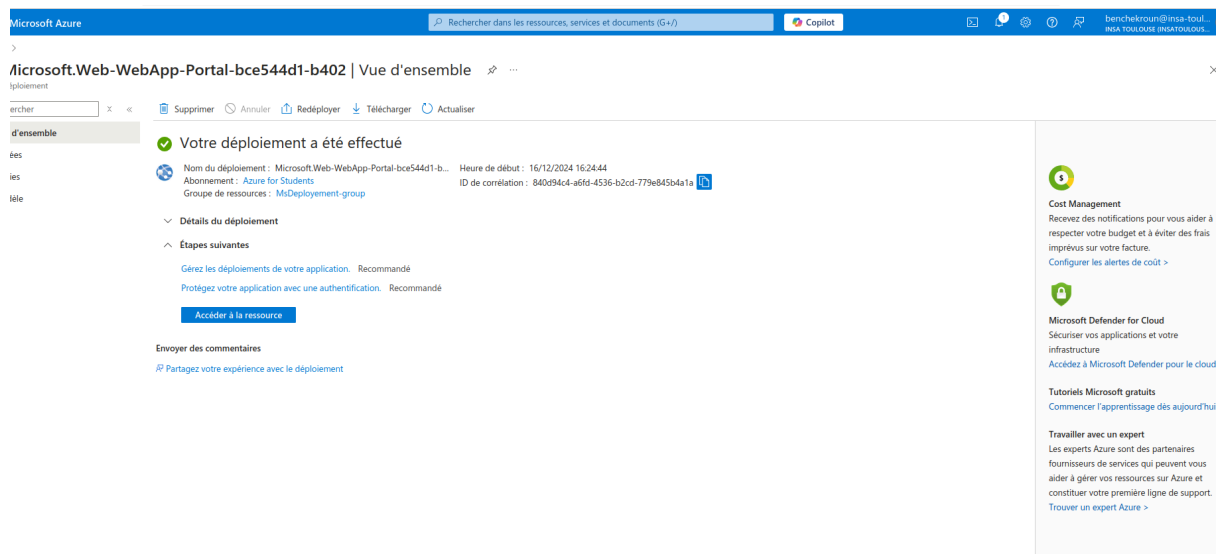
Région *

Vous ne trouvez pas votre plan App Service ? Essayez une autre région ou sélectionnez votre environnement App Service Environment.

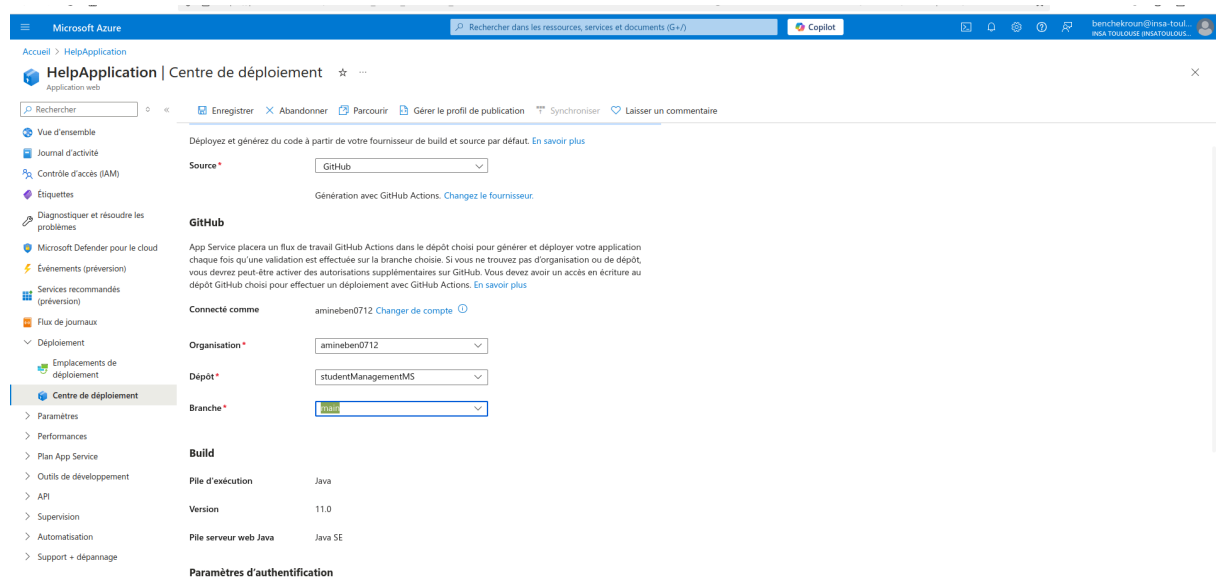
Lors de la configuration de l'App Service pour déployer une application web, plusieurs éléments sont définis pour garantir un déploiement efficace :

- **Abonnement** : L'abonnement sélectionné, ici Azure for Students, offre un environnement cloud adapté aux projets éducatifs.
- **Groupe de ressources** : Le groupe MsDeployment-group est utilisé pour organiser les différentes ressources associées au projet.
- **Nom de l'application** : L'application est nommée HelpApplication, avec un hôte unique généré automatiquement pour faciliter l'accès via une URL spécifique.
- **Pile d'exécution** : L'environnement d'exécution choisi est Java 11, adapté pour les projets basés sur Java.
- **Serveur web Java** : Le serveur utilisé est Java SE (Embedded Web Server), idéal pour des applications légères.
- **Système d'exploitation** : L'application sera déployée sur un environnement Linux, connu pour sa stabilité et son efficacité en milieu cloud.

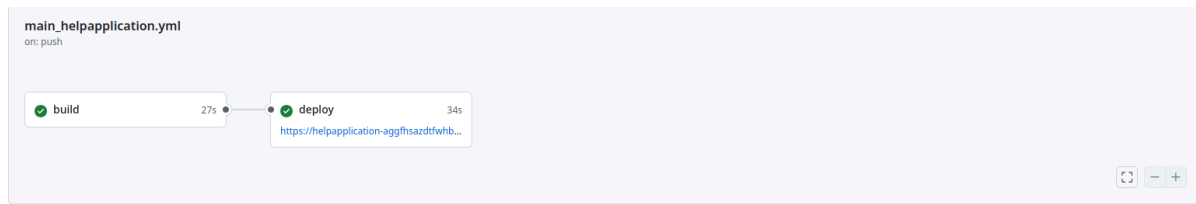
Cette configuration permet de déployer une application web de manière structurée et performante.



Le tableau de bord confirme que le déploiement de l'application web sur Microsoft Azure a été effectué avec succès. Il affiche les informations liées au déploiement, notamment le nom du projet, l'abonnement utilisé, et le groupe de ressources associé. Un bouton permet d'accéder directement à la ressource déployée pour la tester ou l'utiliser. Des options supplémentaires sont proposées pour optimiser la gestion des coûts et renforcer la sécurité de l'infrastructure via Microsoft Defender for Cloud.



Le centre de déploiement de l'application "HelpApplication" sur Microsoft Azure est configuré pour automatiser le processus de déploiement via GitHub. La source du code est liée à un dépôt GitHub, et le déploiement est géré à l'aide de GitHub Actions. L'application est connectée au compte GitHub "amineben0712", en utilisant le dépôt "studentManagementMS" et la branche "main". Cette configuration permet d'intégrer un flux de travail CI/CD, garantissant que tout nouveau push sur la branche principale déclenche automatiquement un processus de build et de déploiement. L'environnement d'exécution est basé sur Java 11 avec Java SE comme serveur web.



Ce diagramme représente le flux de travail CI/CD configuré pour l'application "HelpApplication". Ce workflow, défini dans le fichier `main_helpapplication.yml`, est déclenché par chaque push sur la branche principale du dépôt GitHub. Il se compose de deux étapes principales : le build, qui vérifie et compile le projet pour s'assurer de sa stabilité, suivi du deploy, qui déploie l'application sur Microsoft Azure. Le déploiement réussi est indiqué par le lien généré, permettant d'accéder directement à l'application déployée en ligne. Ce processus garantit une intégration et un déploiement continus, rendant le déploiement fiable et automatisé.

Conclusion :

Le projet présenté dans ce rapport a permis d'explorer en profondeur les différentes architectures logicielles, de SOAP à REST, pour aboutir à une architecture basée sur les microservices. Chacune de ces étapes a contribué à enrichir notre compréhension des systèmes distribués, tout en mettant en évidence leurs atouts et leurs limites dans un contexte applicatif réel.

La transition progressive vers des solutions plus modernes et agiles a démontré l'importance d'une architecture adaptée aux besoins spécifiques du projet. L'implémentation des microservices a particulièrement illustré les bénéfices d'une modularité accrue, d'une évolutivité facilitée et d'une maintenance simplifiée. En parallèle, l'utilisation de Spring Boot et le déploiement sur Microsoft Azure ont permis de renforcer nos compétences en gestion des infrastructures cloud et en intégration continue.

Cependant, par manque de temps, nous n'avons pas pu ajouter davantage de méthodes ou de microservices pour mieux simuler l'application Volunteering. Par exemple, il aurait été pertinent d'intégrer un microservice permettant d'ajouter des feedbacks, ainsi qu'un autre dédié à la gestion de l'attribution des demandes. Ces ajouts auraient enrichi les fonctionnalités de l'application et renforcé son utilité pratique.

Enfin, ce projet a non seulement répondu aux objectifs fixés, mais il a également contribué à approfondir nos connaissances techniques et à consolider nos compétences en ingénierie logicielle, tout en nous confrontant aux défis inhérents aux architectures modernes. Il ouvre la voie à de nouvelles perspectives pour des solutions encore plus performantes et adaptées aux évolutions technologiques à venir.