INSA
TOULOUSE

Compte-rendu
-
Service Oriented Architecture

BIENDOU DJAYOU Brian - HENRIET Baptiste - 5ISS

Introduction

- Contexte et objectifs du projet
- Vue d'ensemble de l'architecture REST et des microservices

Analyse des Besoins et de l'Architecture

- Analyse fonctionnelle et technique
- Choix de l'architecture REST et des microservices

Conception de l'Architecture

- Définition des composants principaux
- Interactions entre les microservices

Implémentation et Développement

- Développement des services REST et des microservices
- Configuration et intégration des bases de données
- Gestion des dépendances et des versions

Intégration continue et tests

Conclusion

- Bilan des travaux réalisés
- Évolutions et perspectives futures

1. Introduction

Contexte et objectifs du projet

Le projet s'inscrit dans le cadre du développement d'un système logiciel visant à gérer les données relatives aux bénévoles dans un environnement associatif. L'objectif principal est de concevoir une application scalable et modulaire permettant de gérer les informations des bénévoles, notamment leurs coordonnées et leur disponibilité, dans un système réparti et hautement disponible.

Les contraintes de ce projet incluent la gestion efficace de la communication entre les services, l'accès aux données de manière sécurisée et en temps réel, ainsi que l'évolutivité pour intégrer de nouvelles fonctionnalités à l'avenir. De plus, il est essentiel que l'application soit robuste et capable de supporter un grand nombre de requêtes concurrentes tout en maintenant une bonne performance.

Vue d'ensemble de l'architecture REST et des microservices

L'architecture choisie repose sur une **architecture REST** combinée à des **microservices**. Cette approche présente plusieurs avantages, notamment une plus grande flexibilité et une évolutivité accrue. En utilisant l'architecture REST, chaque service du système est accessible via des endpoints HTTP standardisés, permettant ainsi de simplifier la communication entre les différents composants logiciels, et de garantir une interopérabilité optimale.

Les **microservices** permettent de décomposer l'application en petites unités indépendantes, chacune étant responsable d'une fonctionnalité spécifique. Chaque microservice peut être développé, déployé et mis à l'échelle indépendamment des autres. Cette approche favorise également l'isolement des erreurs et simplifie la maintenance, car chaque service peut être mis à jour sans affecter le reste du système.

Dans le cadre de ce projet, chaque microservice se voit attribuer une responsabilité bien définie, et la communication entre ces services se fait via des API REST. L'architecture repose donc sur un ensemble de services distincts qui collaborent pour accomplir des tâches communes, tout en maintenant une séparation claire des préoccupations et une gestion efficace des ressources. Cela garantit également une plus grande flexibilité pour l'intégration de nouvelles fonctionnalités dans le futur.

Ainsi, l'utilisation combinée de l'architecture REST et des microservices permet de répondre aux exigences de modularité, de scalabilité, et de flexibilité imposées par ce projet, tout en garantissant une gestion optimisée des données et une communication fluide entre les différentes parties du système.

2. Analyse des Besoins et de l'Architecture

Analyse fonctionnelle et technique

L'analyse fonctionnelle et technique permet de définir précisément les besoins du système ainsi que les contraintes techniques à respecter pour garantir l'efficacité, la sécurité et la robustesse de l'application.

Besoins fonctionnels :

- **Gestion des bénévoles :** L'application doit permettre la gestion des données des bénévoles, telles que leurs informations personnelles (nom, email, téléphone) et leur disponibilité.
- **Stockage des données :** Les informations des bénévoles doivent être stockées de manière centralisée dans une base de données relationnelle, permettant une gestion efficace et sécurisée des données.
- **Communication entre services :** L'application doit permettre la communication entre différents services (par exemple, un service de gestion des utilisateurs et un service de gestion des bénévoles), via des appels d'API standardisés.
- **Modification des données :** Le système doit permettre aux utilisateurs autorisés d'ajouter, mettre à jour et supprimer les informations des bénévoles.
- **Évolutivité et performance :** L'architecture doit permettre de supporter une augmentation du nombre d'utilisateurs et de bénévoles tout en maintenant une bonne performance.

Besoins techniques :

- **Sécurité :** L'architecture doit garantir la sécurité des données sensibles (par exemple, les informations personnelles des bénévoles) en mettant en place des mécanismes de validation des entrées et d'authentification des utilisateurs.
- **Scalabilité :** Le système doit être conçu pour supporter un nombre croissant de bénévoles et d'utilisateurs, en permettant l'ajout de nouveaux services sans perturber le fonctionnement existant.
- **Interopérabilité :** Les services doivent pouvoir interagir entre eux de manière fluide, grâce à des API bien définies.
- **Fiabilité :** L'architecture doit être résiliente face aux pannes et permettre une récupération rapide des données en cas de défaillance.

Choix de l'architecture REST et des microservices

Pour répondre à ces besoins, l'architecture **REST** (Representational State Transfer) a été choisie en combinaison avec l'approche **microservices**.

1. **Architecture REST :** L'architecture REST permet une communication simple et performante entre les services via des requêtes HTTP standardisées (GET, POST, PUT, DELETE). Elle offre plusieurs avantages, notamment :

    - **Simplicité et flexibilité :** REST repose sur des principes simples d'utilisation des verbes HTTP pour manipuler les ressources, ce qui rend l'intégration et l'interopérabilité des services plus faciles.
    - **Scalabilité :** Grâce à sa nature stateless, chaque requête HTTP est indépendante, ce qui permet à l'architecture de mieux gérer les volumes élevés de requêtes.
    - **Accessibilité des services :** Les API REST permettent de rendre les services accessibles à d'autres applications et systèmes via des URLs bien définies, ce qui facilite l'intégration de nouvelles fonctionnalités à l'avenir.
2. **Architecture des microservices :** L'architecture des microservices permet de décomposer l'application en plusieurs services autonomes, chacun responsable d'une fonctionnalité spécifique. Ces microservices communiquent entre eux via des API REST et sont déployés indépendamment les uns des autres. Les principaux avantages de cette approche sont :
    - **Indépendance et modularité :** Chaque microservice peut être développé, déployé et mis à jour indépendamment, ce qui réduit le risque de perturbation des autres services et facilite l'intégration de nouvelles fonctionnalités.
    - **Scalabilité et performance :** Chaque microservice peut être mis à l'échelle de manière indépendante en fonction de la charge de travail, ce qui permet une gestion optimale des ressources.
    - **Facilité de maintenance :** L'architecture microservices permet de maintenir un codebase plus propre et moins complexe, car chaque service est plus petit et focalisé sur une tâche spécifique.
    - **Résilience et tolérance aux pannes :** En cas de défaillance d'un service, les autres services peuvent continuer à fonctionner normalement, ce qui améliore la robustesse du système global.

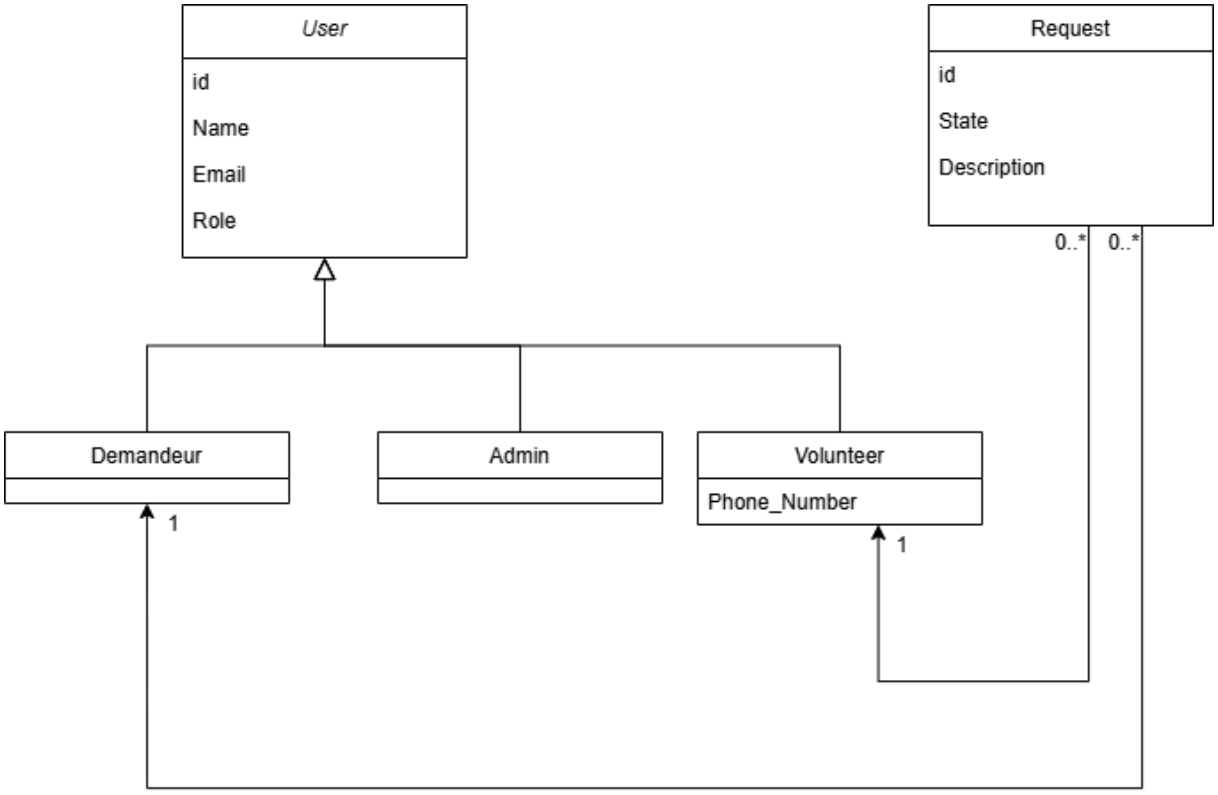
Cette combinaison d'architecture REST et de microservices répond de manière optimale aux exigences du projet, en offrant une solution flexible, scalable, et facile à maintenir. Les services peuvent évoluer indépendamment, tout en maintenant une communication fluide entre eux grâce à des API REST bien définies.

3. Conception de l'Architecture

Définition des composants principaux

Notre projet consiste en la manipulation d'une base de données à l'aide de micro-services, il est donc essentiel que cette base de données soit conçue en accord avec les problématiques du sujet.

Notre base de données présente cinq tables. Quatre de ces tables servent à stocker les différents types d'utilisateur et la dernière permet de stocker les requêtes créées.



Les services **AdminService**, **VolunteerService** et **DemandeurService** fonctionnent sur des principes similaires, chaque service étant responsable d'une gestion spécifique d'entités avec des spécificités qui les distinguent. Ces services suivent tous l'architecture de microservices, communiquant via des API REST, mais avec des responsabilités et des actions différentes selon leur rôle.

La table user contient tous les attributs communs des différents types d'utilisateurs et permet de lister tous les utilisateurs enregistrés. Le seul attribut non commun à tous les types d'utilisateurs et le numéro de téléphone, propre à la table volunteer. Les clés primaires des 3 sous types d'utilisateurs, qui sont leurs id, renvoient toutes à une clé étrangère appartenant à un membre de user. L'id d'un user est généré automatiquement à son ajout dans la table afin d'assurer de ne pas avoir 2 id communs.

La table request contient toutes les requêtes créées par un utilisateur du type demandeur. Elles possèdent toutes un id généré automatiquement, et l'id du demandeur l'ayant créée. Un demandeur peut créer autant de requêtes qu'il le souhaite. Les requêtes peuvent également posséder une description, un état et l'id d'un volontaire, tous optionnels. La description est ajoutée lors de la création de la requête. Lors de cette création, le state est normalement établi comme "pending", il peut ensuite changer selon les actions des utilisateurs (notamment admin et volunteer). Un volunteer peut choisir d'accepter un requête, auquel cas le state devient "accepted" et le volunteerID devient l'id du volunteer en question.

Le **VolunteerService** est responsable de la gestion des bénévoles, incluant la création, la consultation, la mise à jour et la suppression des bénévoles.

Opérations :

- **Création d'un bénévole** : Le service reçoit une requête **POST** pour ajouter un bénévole. Les informations de base (nom, email, numéro de téléphone) sont sauvegardées dans la base de données. Ce service interagit avec le **UserService** pour créer un utilisateur dans le système.
- **Consultation des bénévoles** : Une requête **GET** permet de lister les bénévoles enregistrés. Cette opération peut être utilisée pour afficher tous les bénévoles disponibles dans le système.
- **Mise à jour d'un bénévole** : Lorsque des informations d'un bénévole sont modifiées, une requête **PUT** est envoyée, mettant à jour les informations dans la base de données.
- **Suppression d'un bénévole** : Une requête **DELETE** permet de supprimer un bénévole du système.

Les technologies utilisées incluent **Spring Boot**, **Spring Data JPA** et **RestTemplate** pour la communication entre les services.

AdminService

Le **AdminService** partage une structure similaire avec le **VolunteerService**, mais il est spécifiquement destiné à la gestion administrative du système. Il permet à un administrateur de gérer tous les aspects des utilisateurs et des bénévoles, incluant des actions plus sensibles telles que la gestion des rôles et des permissions.

Opérations :

- **Gestion des utilisateurs** : Ce service permet à l'administrateur de gérer les rôles des utilisateurs, tels que promouvoir un utilisateur au rôle de bénévole ou le rétrograder à un statut d'utilisateur standard.
- **Création et modification des rôles** : Les administrateurs peuvent attribuer ou révoquer des rôles spécifiques à chaque utilisateur, y compris les bénévoles et les demandeurs.
- **Consultation des activités des utilisateurs** : Une requête **GET** permet à l'administrateur de voir l'historique des actions des bénévoles ou des demandeurs. Cela inclut les requêtes en attente, acceptées ou rejetées.
- **Suppression des utilisateurs** : Une requête **DELETE** permet à l'administrateur de supprimer un utilisateur du système.

La principale différence ici réside dans la gestion des permissions et des rôles. L'administrateur peut non seulement ajouter et modifier des utilisateurs, mais aussi gérer leur niveau d'accès et leur fonction dans le système.

DemandeurService

Le **DemandeurService** est destiné à la gestion des requêtes créées par les utilisateurs de type **demandeur**. Ce service gère les opérations liées à la création, la consultation, la modification et la suppression des requêtes de service que les demandeurs soumettent.

Opérations :

- **Création d'une requête** : Lorsqu'un demandeur souhaite soumettre une nouvelle requête, il envoie une requête **POST** avec des informations telles que la description de la requête, son état (initialement "pending") et éventuellement l'**id** d'un volontaire.
- **Consultation des requêtes** : Une requête **GET** permet au demandeur de consulter toutes ses requêtes passées ou en cours.
- **Mise à jour du statut d'une requête** : Lorsque le statut d'une requête change (par exemple, une requête est acceptée par un bénévole), une requête **PUT** est envoyée pour mettre à jour l'état de la requête dans la base de données.
- **Suppression d'une requête** : Le demandeur peut supprimer une requête avec une requête **DELETE**, si nécessaire.

L'interaction entre **DemandeurService** et **VolunteerService** est clé, car un volontaire peut accepter une requête, ce qui change l'état de la requête du demandeur à "accepted".

Points Communs

Tous les services suivent un modèle de gestion similaire :

- **Création** : Ajout de nouveaux éléments (bénévoles, utilisateurs, requêtes).
- **Consultation** : Récupération d'informations existantes via des requêtes **GET**.
- **Mise à jour** : Modification des éléments via des requêtes **PUT**.
- **Suppression** : Suppression d'éléments via des requêtes **DELETE**.

Les services sont interconnectés via des appels d'API, où chaque service communique avec les autres services via **RestTemplate**. Par exemple :

- Le **VolunteerService** appelle **UserService** pour enregistrer un utilisateur comme bénévole.
- Le **DemandeurService** interagit avec le **VolunteerService** lorsque les bénévoles acceptent des requêtes.

Cette architecture modulaire, où chaque service est autonome mais communique de manière fluide avec les autres, offre une grande flexibilité, scalabilité et résilience. Grâce à l'utilisation de REST et de microservices, il devient plus facile d'ajouter de nouveaux services ou de modifier les services existants sans impacter l'ensemble du système.

4. Implémentation et Développement

Développement des services REST et des microservices

L'implémentation des services REST et des microservices a été réalisée en utilisant **Spring Boot**, un framework puissant pour le développement d'applications Java basées sur les principes de l'architecture REST. Chaque microservice a été conçu pour remplir une tâche spécifique en suivant les meilleures pratiques du développement microservices et en veillant à la modularité et à l'indépendance des composants.

Les microservices développés incluent les services suivants :

1. **VolunteerService (Service de gestion des bénévoles)** : Ce service gère les informations des bénévoles telles que leur nom, email et numéro de téléphone. Il expose des API REST permettant de réaliser des opérations CRUD (Créer, Lire, Mettre à jour, Supprimer) sur les bénévoles.
    - **Méthodes principales** :
        - `GET /api/volunteers/get` : Récupérer tous les bénévoles.
        - `GET /api/volunteers/{id}` : Récupérer un bénévole par son identifiant.
        - `POST /api/volunteers` : Ajouter un nouveau bénévole.
        - `PUT /api/volunteers/{id}` : Mettre à jour un bénévole existant.
        - `DELETE /api/volunteers/delete/{id}` : Supprimer un bénévole.
2. **UserService (Service de gestion des utilisateurs)** : Ce service est responsable de l'authentification des utilisateurs et de la gestion de leurs rôles. Il interagit directement avec le service **VolunteerService** pour intégrer les bénévoles dans la gestion des utilisateurs. Il attribue des rôles (comme celui de bénévole) aux utilisateurs pour assurer la bonne gestion des profils.
3. **RequestService (Service de gestion des demandes)** : Ce service permet la gestion des demandes de bénévoles ou des utilisateurs. Il inclut des API pour créer, gérer et traiter les demandes associées à différentes actions, telles que des demandes de bénévolat ou de services.
4. **MSDiscovery (Service de découverte)** : Ce service utilise Spring Cloud Eureka pour permettre la découverte et l'enregistrement dynamique des autres microservices dans l'architecture distribuée. Cela permet à chaque service de se découvrir et de communiquer sans nécessiter de configurations statiques.
5. **DemandeurService (Service de gestion des demandeurs)** : Ce service gère les informations des demandeurs (utilisateurs ayant besoin de bénévoles). Il interagit avec **RequestService** pour traiter les demandes émises par les demandeurs, et permet de suivre l'état des demandes en fonction des besoins.
6. **AdminService (Service d'administration)** : Ce service offre des fonctionnalités d'administration pour la gestion des bénévoles, des utilisateurs, des demandes et de l'ensemble des autres services. Il permet de superviser, ajouter, supprimer et mettre à jour les informations liées aux bénévoles et utilisateurs.

Configuration et intégration des bases de données

Chaque microservice utilise sa propre base de données afin de garantir l'indépendance des services et un découplage optimal. Les bases de données sont configurées et intégrées de manière à ce que chaque microservice puisse gérer ses propres données sans dépendre d'un autre service.

- **Base de données MySQL pour VolunteerService** : La base de données utilisée pour **VolunteerService** contient une table dédiée aux bénévoles avec les colonnes suivantes :
  - `id` : Identifiant unique du bénévole (clé primaire).
  - `name` : Nom du bénévole.
  - `email` : Email du bénévole.
  - `phoneNumber` : Numéro de téléphone du bénévole.

La connexion à la base de données est gérée par **Spring Data JPA**, ce qui permet une gestion simplifiée des entités et de leurs interactions avec la base de données. Les propriétés de connexion à la base de données sont définies dans le fichier `application.properties`.

Gestion des dépendances et des versions

La gestion des dépendances dans ce projet est assurée par **Maven**. Le fichier `pom.xml` contient toutes les dépendances nécessaires au fonctionnement des microservices, ainsi que les versions des bibliothèques utilisées.

Dépendances principales :

- **Spring Boot Starter Web** pour la création d'API REST.
- **Spring Boot Starter Data JPA** pour la gestion de la persistance des données avec Spring Data JPA.
- **MySQL Connector** pour connecter l'application à la base de données MySQL.
- **Spring Cloud Eureka Client** pour la gestion centralisée des services (utilisé dans **MSDiscovery**).

Les versions des dépendances sont gérées dans le fichier `pom.xml` pour garantir la compatibilité entre les bibliothèques et éviter les conflits. Par exemple, la version de **Spring Boot** utilisée est 3.4.0, et celle de **Spring Cloud** est 2024.0.0.

Cette phase d'implémentation a permis de développer les services REST de manière sécurisée et efficace, tout en intégrant les bases de données nécessaires pour stocker les informations relatives aux bénévoles et utilisateurs. La gestion des dépendances et des versions garantit la stabilité et la compatibilité du projet avec les différentes bibliothèques et frameworks utilisés.

5. Intégration continue et tests

La partie intégration continue du projet a été réalisée avec Microsoft Azure. Pour cela, on crée une Application Web sur Microsoft Azure, que l'on relie au repository GitHub du projet. On obtient alors un fichier .yml contenant les paramètres du déploiement, nous permettant d'indiquer le chemin vers le projet UserService, sur lequel on applique ainsi le workflow. Ce workflow permet de réaliser la compilation maven du projet UserService automatiquement à chaque commit réalisé sur le Git, et donc de vérifier automatiquement le passage des tests JUnit présents dans le projet UserService. Cependant, la compilation maven du projet UserService renvoie une erreur que nous ne sommes pas parvenus à résoudre. Ainsi, malgré la bonne exécution du projet, nous n'avons pas pu réaliser de test unitaire.

Pour cette raison, tous les tests ont été réalisés manuellement en utilisant Postman et en surveillant la base de données. Les différentes requêtes réalisables sur ce projet sont présentées et expliquées dans le README.

Conclusion

Ce projet était notre introduction aux concepts de SOA, d'API RESTful et des applications basées sur des micro-services. Le projet nous a permis d'appréhender les problématiques liées à ces concepts, comme la décentralisation de l'application, à travers un projet relativement simple conceptuellement. Nous n'avons malheureusement pas pu réaliser la partie "Intégration continue" du projet, nous avons pu cependant tout de même développer une application fonctionnelle et relativement complète, remplissant globalement le cahier des charges donné. De plus, bien que l'intégration continue soit une partie importante de tout projet de développement, il ne s'agissait pas de l'objectif principal de ce projet, le fait que nous n'ayons pas pu la réaliser, bien que dommageable, reste donc au final assez inconséquent quant à l'enseignement concerné.

Au-delà de ça, le projet présente encore des points améliorables. Il n'est par exemple pas possible de se connecter à un profil utilisateur présent dans la base de données. Il aurait également pu être utile de présenter une interface graphique, bien que cela ne soit pas demandé, afin de faciliter les interactions entre l'utilisateur et l'application. Certaines méthodes sont également imprécises, par exemple, il est possible de mettre n'importe que String (varchar(255) plus précisément) dans la colonne "state" de la table "request". Un set de méthodes permettant de changer le "state" en des valeurs prédéfinies uniquement aurait été plus proche du cahier des charges.