Projet API
William VONDERSCHER - Matthieu JACQUINET

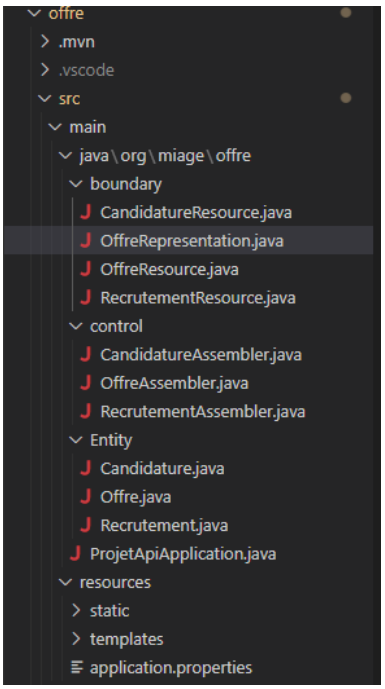
Conception

Nous avons réalisé 2 services distincts : Offre et Personne, le premier service va gérer les offres, candidatures et processus de recrutement, le deuxième service gère les personnes qui peuvent candidater sur les offres.
Nous avons commencé par mettre en place les fichiers du projet, 2 projets spring boot distincts (un pour chaque service).

Nous avons utilisé Docker, nos deux services sont des images de plus avec une image PostgreSQL pour la base de données.

```
FROM eclipse-temurin:17-jdk-jammy
VOLUME /tmp
ADD target/offre-0.0.1-SNAPSHOT.jar offre-0.0.1-SNAPSHOT.jar
RUN bash -c 'touch /offre-0.0.1-SNAPSHOT.jar'
ENTRYPOINT ["java","-jar" ,"offre-0.0.1-SNAPSHOT.jar"]
```

Ci-dessus est le dockerfile pour le service offre, on a utilisé l'image eclipse-temurin pour l'environnement JAVA, c'est la même chose pour le service personne. Dans le fichier docker-compose, on indique toutes les images du projet pour la création du container (service-offre, service-personne, postgresql).



L'arborescence de notre projet est celui ci-dessus (exemple avec le service offre), dans le package boundary se trouve les controllers (OffreResource) qui vont gérer les différents chemins. Nous avons aussi les Repository JPA (RecrutementResource…) qui vont permettre d'utiliser les méthodes CRUD. Dans le dossier control, nous avons les fichiers "assembler" qui vont nous permettre de mettre en place HATEOAS.
Le dossier entity contient les objets du projets qui vont peupler la base de données:

- L'objet Offre contient les attributs décrits dans le sujet, nous avons ajouté en plus un attribut "vacante" qui est à true lors de la création d'une offre qui permet d'indiquer si cette offre peut recevoir des candidatures ou non.

```java
@Id
private String id;
private String nomStage;
private String domaine; // trier
private String nomOrganisation; //trier
private String descriptionStage;
private String datePublicationOffre;
private String niveauEtudeStage;
private String experienceRequiseStage;
private String dateDebutStage; //trier
private String dureeStage;
private String salaireStage;
private String indemnisation;
private String organisationAdresse;
private String organisationMail;
private String organisationTel;
private String organisationURL;
private String lieuStageAdresse; //trier
private String lieuStageTel;
private String lieuStageURL;
private boolean vacante;
```

- L'objet Candidature contient lui un ID, l'id de l'offre qui est associé, l'id du user qui candidate et le nom du candidat

```
private String id;
private String idOffre;
private String idUser;
private String nomCandidat;
```

- L'objet Recrutement est l'objet qui représente le processus de recrutement : il contient un ID, l'id de la candidature, le nombre d'entretiens et la décision qui est par défaut "aucune".

```
private String id;
private String idCandidature;
private String nombreEntretien;
private String decision;
```

- Du côté du service Personne, il y a l'objet Personne qui décrit un user il a un ID, un nom et un numéro de téléphone.

```
private String id;
private String nomUser;
private String telUser;
```

Le service offre se trouve sur le port 8000 et le service personne se trouve sur le port 8100.

Réalisation

Dans cette partie nous allons voir comment nous avons réalisé les différentes fonctionnalités du projet.

Fonctionnalités

- Les offres de stages sont créées par l'organisation qui propose les stages (POST offres).

POST "/offres".
Cette fonctionnalité se trouve dans le service offre dans lequel nous récupérons les valeurs du body de la requête post puis nous les transférons dans un objet Offre qui contient les différents champs. Nous terminons par sauvegarder grâce au repository l'offre.
A noter que les ID sont générés automatiquement avec UUID.

- une personne peut accéder aux offres de stages (GET offres). Les offres peuvent être filtrées en fonction du domaine, de la date de début, de l'organisation, du lieu… Une personne peut également avoir accès aux détails d'une offre (GET offres/ 425e7701-02c6-4de3-9333-a2459eece1c8).

Pour la première partie de cette fonctionnalité, nous avons deux méthodes, une pour récupérer toutes les offres (chemin : **GET "/offres"**) qui s'occupe de retourner toutes les offres. Une autre méthode pour récupérer une offre particulière (chemin : **GET "/offres/{idOffre}"**) qui retourne une offre particulière.
La deuxième partie concerne l'utilisation de chemins spécifiques en fonction du domaine, de la date de début du stage, de l'organisation et du lieu du stage. Pour ce faire nous avons utilisé des chemins spécifiques (exemple pour trier sur les domaines : **GET "/offres/domaine/{nomDomaine}"** Nous avons, dans les méthodes pour récupérer les offres, utiliser une nouvelle méthode que nous avons ajouté dans le fichier repository, par exemple pour le domaine, nous avons une méthode nommée : "findByDomaine" qui va permettre de récupérer la liste des offres pour un domaine donné. On réalise le même traitement pour chaque "filtre".

- Par défaut, une offre est vacante (statut non représenté dans les informations ci-dessus).

Pour cette partie, nous avons ajouté un champ dans l'entité Offre un booléen "vacante" qui est initialisé à true lors de la création (**POST /offres**), on peut rendre une Offre non vacante en réalisant une requête **PATCH sur /offres/{idOffre}/fermer**, on peut envoyer un body JSON vide car ce patch ne sert qu'à rendre l'offre non vacante, on rend ainsi toutes nouvelles candidatures sur cette offre impossible.

- une personne peut postuler à une offre (POST offres/425e7701-02c6-4de3-9333-a2459eece1c8). Il faudra veiller à associer cette personne à l'offre sélectionnée (non représenté ci-dessus).

En réalisant un **POST sur le chemin /offres/{idOffre}**, on réalise une candidature, on doit renseigner les informations suivantes : 'id de l'offre qui est associé, l'id du user qui candidate et le nom du candidat, cela va nous permettre de lier le candidat à l'offre. C'est ici qu'on vérifie si la candidature est possible, c'est-à-dire si l'offre est vacante et que la personne n'a pas déjà candidaté sur l'offre.

- l'organisation peut alors commencer un processus de recrutement. C'est l'organisation qui détermine le processus (nombre d'entretiens, décision,…). Les informations liées au processus ne sont pas représentées ci-dessus. Une offre peut éventuellement avoir plusieurs candidatures simultanées.

Pour réaliser cette fonctionnalité, on réalise une requête **POST sur une candidature au chemin : /offres/candidature/{candidatureId}**. Cela va créer un objet Recrutement qui représente le processus, on donne les informations suivantes dans le body de la requête : l'id de la candidature, le nombre d'entretiens. La décision est par défaut à "aucune". On peut passer la décision à "accepté" avec une requête **PATCH sur le chemin /offres/{recrutementId}/accepter** et un body vide. On peut aussi supprimer un processus de recrutement : **DELETE "/offres/recrutements/{recrutementId}"**

- une personne peut connaître les offres sur lesquelles elle a postulé (GET users/johndoe/ candidatures), et le statut d'une offre particulière (GET users/johndoe/candidatures/afd55459-2768-48a4-9573-6a8d9928904).

GET "/users/{nomCandidat}/candidatures" pour toutes les candidatures
GET "/users/{nomCandidat}/candidatures/{offreId}"pour récupérer une candidature
Cette fonctionnalité se trouve dans le service Personne. Pour que la personne puisse connaître ses candidatures, elle doit récupérer les objets associés, mais ces objets se trouvent du côté du service offre, donc le service personne doit communiquer avec le service offre, pour se faire, nous utilisons resttemplate :

```
// GET all candidature
@GetMapping("/{nomCandidat}/candidatures")
public ResponseEntity<?> getAllCandidature(@PathVariable("nomCandidat") String nomCandidat) {
    String url ="http://offreService:8000/offres/{nomCandidat}/candidatures";
    return ResponseEntity.ok(template.getForEntity(url, responseType:CollectionModel.class, nomCandidat)).getBody();
}
```

On définit l'url avec le chemin où se trouve les candidatures de la personne. Il faut noter qu'on utilise "http://offreService:8000" et non localhost, c'est à cause de l'utilisation de docker, si on utilise localhost, la communication ne fonctionnera pas.
Finalement on utilise la méthode "getForEntity" qui va permettre de réaliser une requête GET sur l'url précédent. on affiche ensuite le résultat (une collection).
Pour récupérer une seule candidature, c'est le même raisonnement sauf que l'url n'aura pas le même chemin "/offres/{nomCandidat}/candidatures/{offreId}" et on retourne une Entity et pas une collection.

- l'organisation peut connaître les candidatures à une offre (GET offres/ 425e7701-02c6-4de3-9333-a2459eece1c8/users).

GET "/offres/{offreId}/users"
On cherche toutes les candidatures pour l'offre puis on les affiche depuis le service offre.

- l'organisation peut mettre à jour une offre si besoin (PUT offres/ 425e7701-02c6-4de3-9333-a2459eece1c8). Lorsqu'une offre n'est plus disponible, elle peut être supprimée logiquement (DELETE offres/425e7701-02c6-4de3-9333-a2459eece1c8).

PUT "/offres/{offreId}"
Depuis le service offre, on peut modifier les attributs d'une offre

DELETE "/offres/{offreId}"
Depuis le service offre, on peut supprimer une offre

- une personne peut abandonner un processus de candidature (DELETE users/johndoe/ candidatures/afd55459-2768-48a4-9573-6a8d9928904).

DELETE "users/{nomCandidat}/candidatures/{offreId}"
Depuis le service Personne, une personne peut supprimer une de ses candidatures. Cela effectue une requête vers le service offre vers l'url : "offres/{nomCandidat}/candidatures/{offreId}/delete", on utilise la méthode delete de resttemplate avec l'url et ses paramètres.

Il y a aussi d'autres fonctionnalités sur le service Personne comme la possibilité d'ajouter des personne avec un **POST "/users"**, modifier une personne avec un **PUT "/users/{personneId}"** et en supprimer un avec **DELETE "/users/{personneId}"**.

HATEOAS

HATEOAS va permettre une meilleure fluidité dans la navigation des données de nos services.
Pour gérer HATEOAS, nous avons les classes Assembler (PersonneAssembler, CandidatureAssembler, OffreAssembler, RecrutementAssembler),

Pour les offres, nous avons un lien qui donne vers l'offre elle-même et vers toutes les autres offres existantes.
Pour les personnes, nous avons un lien vers tous les utilisateurs et un lien vers l'utilisateur actuel.
Pour les candidatures, nous avons un lien vers l'offre de la candidature, les offres de l'utilisateur et toutes les offres.
Finalement pour les recrutements, un lien vers le recrutement actuel et vers les autres recrutement.

Tests

Pour la réalisation des tests, nous avons utilisé une base H2 contrairement au reste du projet où nous utilisons une base postgre car si on utilise cette dernière, l'environnement de test ne peut pas se connecter au container postgre. Pour utiliser la base H2 seulement pour les tests, on peut spécifier les propriétés de la base dans la classe de test directement :

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT, properties = {
"spring.datasource.url=jdbc:h2:mem:m2db;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE",
"spring.datasource.driver-class-name=org.h2.Driver",
"spring.datasource.username=sa",
"spring.datasource.password=password",
"spring.jpa.hibernate.ddl-auto=create-drop",
"spring.jpa.defer-datasource-initialization=false"})
```

Comme dans le fichier application.properties dans les ressources du projet.
Nous avons réalisé des tests pour les fonctionnalités des 2 services. Nous utilisons les envoie de requête avec RestAssured et la méthode "when()", ce qui permet d'envoyer des requêtes et de pouvoir tester les résultats de la requête. On commence par remplir la base de données pour le test (la base est vidée à chaque début de test) puis on réalise la requête, on vérifie que le code HTTP de retour est le bon (quand on réalise un GET qui est censé renvoyer des objets, on vérifie bien que le code HTTP est 200) puis on regarde le contenu de la requête est bon.
On peut réaliser des requêtes get, post (on indique le texte JSON pour le payload), patch, put, delete.

Utilisation du projet

Notre projet se trouve dans Github à l'adresse suivante :
https://github.com/wvonderscher/projet_api
Il faut cloner le projet, puis rendez-vous dans le service offre (/projet_api/offre), il faut exécuter la commande : "mvn clean package" et "docker build -t m2/offre:1.0 ." .
Ensuite dans le service personne (/projet_api/personne), il faut réaliser la même chose : "mvn clean package" et "docker build -t m2/personne:1.0 ." .

Maintenant, à la racine (/projet_api), il faut faire "docker compose up" pour lancer le container et utiliser le projet

insérer les données : Il faut le faire depuis les commandes postgresql.
Les commandes de peuplement se trouvent dans le fichier peuplement.txt à la racine du projet, il faut copier le contenu et l'utiliser dans la console pour peupler la bdd. On commence par vider la base et ensuite on la remplit.
Il y a assez de données pour tester les différents aspects du projet.

commande pour se connecter à la base : psql -d postgres -U postgres
puis coller le contenu du fichier txt.