Projet M2 ACSI

Réalisation d'une API métier permettant la gestion d'offres de stages, de la publication jusqu'au recrutement.

Sofiane CHELH - Gauthier JACQUES

Introduction

Nous avons été amenés à réaliser un projet visant à développer nos compétences en matière de conception et de mise en œuvre d'APIs RESTful. Ce projet s'inscrit dans le contexte du cours de distribution Web et Sécurité et vise à nous familiariser avec les concepts et les technologies modernes utilisées dans le développement d'APIs.
L'objectif principal de ce projet est de créer une API RESTful permettant de gérer des offres de stage et les interactions entre les organisations proposant des stages et les personnes intéressées. Pour ce faire, nous avons choisi d'utiliser Java et Spring Boot en tant que technologies principales, ainsi que HATEOAS pour faciliter la navigation et la découverte des ressources de l'API.
Ce rapport présente le processus de conception et d'implémentation de notre API, en mettant l'accent sur les défis techniques rencontrés, les choix effectués et les résultats obtenus. Nous commencerons par analyser les besoins et modéliser les entités de notre domaine, puis nous passerons à la conception des routes et des méthodes pour l'API. Ensuite, nous aborderons l'implémentation de l'API avec Spring Boot et HATEOAS, ainsi que les tests effectués pour valider notre solution. Enfin, nous conclurons en discutant des objectifs atteints, des limitations de notre solution et des perspectives d'évolution du projet.

Conception de l'API

Analyse des besoins

Avant de commencer la conception de notre API, il était crucial d'analyser les besoins et les attentes des utilisateurs potentiels. Nous avons identifié les principales fonctionnalités suivantes pour notre API de gestion des offres de stage :

- Création des offres de stage par les organisations
- Consultation des offres de stage par les personnes intéressées
- Filtrage des offres en fonction de divers critères (domaine, date de début, organisation, lieu, etc.)
- Postulation des personnes aux offres de stage
- Suivi des candidatures par les personnes et les organisations
- Mise à jour et suppression des offres de stage par les organisations

Modélisation des entités

Pour représenter notre domaine métier, nous avons identifié quatre entités principales :

Offre

- id
- nomstage
- domaine

- nomorga
- descstage
- datepub
- nivetu
- exprequise
- datedeb
- duree
- salaire
- indemnisation
- orgaadrpays
- orgaadrville
- orgaadrcp
- orgaadrrue
- orgaemail
- orgaurl
- stageadrpays
- stageadrville
- stageadrcp
- stageadrrue
- latitude
- longitude
- tel
- url
- statut
- nomuserret

Statut permet de définir l'état d'avancement de la procédure de recrutement sur l'offre et nomuserret et l'utilisateur retenu pour l'offre de stage.

Personne

- id
- nom
- prénom
- niveau
- type

Type correspond si la personne est admin ou un simple user cela ne peut être modifier que dans la BDD

Candidature

- id
- idpersonne (relation avec Personne)
- idoffre (relation avec Offre)

Recrutement

- id
- nomOffre
- idCandidature
- nbCandidats

Conception des routes et des méthodes

Nous avons défini les routes et les méthodes suivantes pour notre API :

- POST /offres : créer une nouvelle offre de stage
- GET /offres : récupérer toutes les offres de stage. Filtrage avec des paramètres dans l'url (domaine, nomstage, nomorga)
- GET /offres/{id} : récupérer une offre de stage spécifique par son identifiant
- POST /offres/{id} : postuler à une offre de stage
- GET /personnes/{userId}/candidatures : récupérer les candidatures d'une personne
- GET /personnes/{userId}/candidatures/{id} : récupérer le statut d'une candidature spécifique
- GET /offres/{id}/personnes : récupérer les candidatures pour une offre de stage spécifique
- PUT /offres/{id} : mettre à jour une offre de stage
- DELETE /offres/{id} : supprimer logiquement une offre de stage
- DELETE /personnes/{userId}/candidatures/{id} : abandonner un processus de candidature
- POST /personnes : créer une nouvelle personne
- GET /personnes/{id} : récupérer une personne spécifique par son identifiant
- GET /offres/recrutement/{id} : démarrer une campagne de recrutement
- PATCH /offres/{id} : modifier le statut d'une offre

Implémentation de l'API avec Spring Boot

Configuration de Spring Boot

Nous avons utilisé Spring Boot pour faciliter la création et la configuration de notre application. Spring Boot nous a permis de gérer les dépendances et de configurer notre projet avec une facilité optimale. Pour ce faire, nous avons ajouté les dépendances nécessaires au fichier pom.xml et configuré l'application à l'aide du fichier application.properties.

Définition des entités et des relations

Afin de représenter notre modèle de données, nous avons défini des classes Java pour chacune des entités identifiées dans la phase de conception.

Exemple de classe entité Personne:

```java
package org.miage.ProjetAPI.entity;

import java.io.Serializable;

@Entity
@Getter
@Setter
@NoArgsConstructor
@Table(name = "personne")
public class Personne extends RepresentationModel<Personne> implements Serializable {

    private static final long serialVersionUID = 356898653245L;
    @Id
    private String id;
    private String prenom;
    private String nom;
    private String niveau;
    private String type;
```

Création des contrôleurs et des méthodes

Afin de gérer les requêtes HTTP entrantes, nous avons créé des contrôleurs pour chaque entité et défini les méthodes correspondantes pour chaque route.
Exemple de contrôleur pour les offres de stage :

```java
@Controller
@ResponseBody
@RequestMapping(value = "/offres", produces = MediaType.APPLICATION_JSON_VALUE)
public class OffreRepresentation {

    private static OffreResource or = null;
    private final HttpClient httpClient = HttpClient.newHttpClient();
    private final CandidatureResource cr;

    @Autowired
    public OffreRepresentation(OffreResource or, CandidatureResource cr) {
        this.or = or;
        this.cr = cr;
    }

    @GetMapping(value = "/{offreId}")
    public EntityModel<Offre> getOneById(@PathVariable("offreId") String id){
        Optional<Offre> offreOpt = or.findById(id);

        if (offreOpt.isPresent()) {
            Offre offre = offreOpt.get();
            Link selfLink = linkTo(OffreRepresentation.class).slash(offre.getId()).withSelfRel();

            return EntityModel.of(offre, selfLink);
        } else {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Offre introuvable");
        }
    }
```

Implémentation de HATEOAS

Nous avons utilisé HATEOAS pour enrichir notre API avec des liens permettant de naviguer entre les différentes ressources. Pour ce faire, nous avons ajouté des liens aux objets de

réponse renvoyés par notre API, en utilisant les classes et les méthodes fournies par Spring HATEOAS.
Exemple d'ajout de liens HATEOAS pour l'entité Offre :

```java
@GetMapping
public CollectionModel<Offre> getAllOffres(
        @RequestParam(value = "domaine", required = false) String domaine,
        @RequestParam(value = "nomorga", required = false) String nomorga,
        @RequestParam(value = "nomstage", required = false) String nomstage){

    Specification<Offre> spec = where(null);

    if (domaine != null) {
        spec = spec.and(OffreSpecification.hasDomaine(domaine));
    }
    if (nomorga != null) {
        spec = spec.and(OffreSpecification.hasNomorga(nomorga));
    }
    if (nomstage != null) {
        spec = spec.and(OffreSpecification.hasNomstage(nomstage));
    }

    List<Offre> lsOffre = or.findAll(spec);

    List<Offre> lsOffreRes = new ArrayList<>();

    for (Offre ofr : lsOffre) {
        if(ofr.getStatut().equals("ouvert")) {
            String ofrId = ofr.getId();
            Link selfLink = linkTo(OffreRepresentation.class).slash(ofrId).withSelfRel();
            ofr.add(selfLink);
            lsOffreRes.add(ofr);
        }

    }
    Link link = linkTo(OffreRepresentation.class).withSelfRel();
    CollectionModel<Offre> result = CollectionModel.of(lsOffreRes, link);
    return result;
}
```

Gestion des erreurs et des exceptions

Afin de gérer les erreurs et les exceptions de manière cohérente, nous avons créé des classes d'exception personnalisées et les avons utilisées dans nos contrôleurs et services. Nous avons également utilisé des gestionnaires d'exceptions globales pour capturer les exceptions et renvoyer des réponses HTTP appropriées.
Exemple de gestionnaire d'exceptions :

```java
@PostMapping(value = "/{offreId}")
public ResponseEntity<EntityModel<Candidature>> postulerOffre(@RequestParam String idPersonne, @PathVariable("offreId") String idOffre) {
    Link selfLink = null;
    EntityModel<Candidature> entityModel = null;
    Optional<Offre> offreOpt = or.findById(idOffre);
    try {
        String url = "http://localhost:8081/personnes/" + idPersonne;
        HttpRequest httpRequest = HttpRequest.newBuilder()
                .uri(URI.create(url))
                .GET()
                .build();
        HttpResponse<String> httpResponse = httpClient.send(httpRequest, HttpResponse.BodyHandlers.ofString());

        int statusCode = httpResponse.statusCode();
        if (statusCode == 200 && offreOpt.isPresent()) {
            Candidature newCand = new Candidature(UUID.randomUUID().toString(), idOffre, idPersonne);
            Candidature savedCand = cr.save(newCand);
            selfLink = linkTo(OffreRepresentation.class).slash(idOffre).withSelfRel();
            entityModel = EntityModel.of(savedCand, selfLink);
        }

    }catch (Exception e) {
        e.printStackTrace();
    }
    if(selfLink == null) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Personne ou Offre introuvable");
    }
    return ResponseEntity
            .created(selfLink.toUri())
            .body(entityModel);

}
```

Tests de l'API

Stratégie de test

Pour s'assurer que notre API fonctionne correctement et de manière fiable, nous avons adopté une stratégie de test comprenant des tests unitaires et des tests d'intégration. Les tests unitaires ont été réalisés pour vérifier le comportement des différentes classes et méthodes. Les tests d'intégration ont été utilisés pour valider le comportement de l'API dans son ensemble, y compris les interactions avec la base de données.

Test des routes et des méthodes

Nous avons utilisé des frameworks de test tels que JUnit et Mockito pour tester les routes et les méthodes de notre API. Les tests ont été réalisés pour chaque contrôleur et méthode, en s'assurant que les réponses HTTP appropriées sont renvoyées pour chaque cas de test. Exemple de test d'une méthode GET pour récupérer toutes les offres :

```java
@SpringBootTest
@AutoConfigureMockMvc
class ProjetApiApplicationTests {

     @Autowired
     private MockMvc mockMvc;

    @Autowired
    private OffreResource or;

    @Test
    public void testGetAllOffres() throws Exception {
        mockMvc.perform(get("/offres"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$._embedded.offreList").exists());
    }
```

Conclusion

Résumé des objectifs atteints

Au cours de ce projet, nous avons réussi à développer une API RESTful en utilisant le framework Spring Boot pour gérer les offres de stage et les candidatures des personnes. Nous avons implémenté des fonctionnalités pour créer, récupérer, mettre à jour et supprimer des offres et des candidatures, ainsi que pour filtrer et rechercher des offres en fonction de divers critères. En outre, nous avons intégré HATEOAS pour fournir une expérience utilisateur riche et pour faciliter la navigation entre les ressources associées.

Limitations et améliorations possibles

Bien que notre API réponde aux exigences du projet, il y a plusieurs limitations et améliorations possibles. Nous n'avons pas traité les aspects liés à la sécurité, tels que l'authentification et l'autorisation, qui pourraient être ajoutés pour protéger les données sensibles. De plus, nous pourrions envisager d'améliorer les performances en ajoutant des mécanismes de mise en cache pour réduire les temps de réponse et optimiser l'utilisation des ressources.