ENSEIGNEMENT DE PROMOTION ET DE FORMATION CONTINUE DE
L'UNIVERSITE LIBRE DE BRUXELLES

RAPPORT DE FIN D'ÉTUDES

WorkMatch : Développement d'une plateforme de matching emploi

Année académique 2024-2025

Réalisé par : Özkan ÖZKARA

Superviseur : Boris VERHAEGEN

Table des Matières

1. Introduction

Aujourd'hui, trouver un emploi ou recruter quelqu'un, ce n'est pas toujours évident. Il existe plein de plateformes, mais souvent, elles ne proposent pas des offres ou des candidats vraiment adaptés. C'est pour ça que j'ai voulu créer WorkMatch, une application qui aide à mieux faire correspondre les employeurs et les chercheurs d'emploi.

Pour développer WorkMatch, j'ai utilisé Spring Boot pour le backend, React pour le frontend et MongoDB pour la base de données. J'ai aussi intégré l'API Adzuna pour récupérer des offres d'emploi en temps réel.

Ce rapport va expliquer comment j'ai construit l'application, les choix techniques que j'ai faits et les difficultés rencontrées. Je vais aussi parler des améliorations possibles pour rendre WorkMatch encore plus efficace.

1.1. Objectifs du projet

Quelle problématique WorkMatch résout ?

Aujourd'hui, beaucoup de plateformes d'emploi existent, mais elles fonctionnent souvent sur un modèle où les candidats envoient des CVs à la chaîne et où les recruteurs reçoivent trop de candidatures non adaptées. Il manque un vrai système de correspondance intelligent pour que les offres soient proposées aux bons profils sans que les entreprises aient à faire un tri manuel énorme.

WorkMatch cherche à rendre ce processus plus fluide en utilisant un système de matching basé sur les compétences et l'expérience des candidats pour ne proposer que les offres réellement pertinentes.

En quoi est-ce innovant ?

Contrairement aux sites classiques qui fonctionnent comme un simple moteur de recherche, WorkMatch automatise le matching entre les candidats et les offres d'emploi grâce à des critères spécifiques comme les compétences, l'expérience et les préférences du candidat. L'objectif est de réduire le temps passé à chercher un emploi ou un candidat, et d'améliorer la qualité des recrutements.

1.2. Méthodologie adoptée

Pour développer WorkMatch, j'ai suivi une approche assez progressive, en avançant étape par étape. Au départ, je voulais créer une plateforme simple pour permettre aux chercheurs d'emploi et aux recruteurs de se trouver plus facilement, un peu comme une application de rencontre, mais pour le travail.

J'ai commencé par définir les besoins du projet, en me basant sur les plateformes existantes et en réfléchissant aux améliorations possibles. Ensuite, j'ai choisi les technologies les plus adaptées : Spring Boot pour le backend, React Native pour le frontend, et MongoDB pour la base de données.

Une fois ces bases posées, j'ai découpé le projet en plusieurs étapes :

1. **Mise en place du backend avec Spring Boot**, création des endpoints et gestion de la base de données.
2. **Intégration de l'API Adzuna**, récupération et stockage des offres d'emploi.
3. **Développement du frontend avec React Native**, en créant une interface simple avec un système de swiping (like = droite/dislike = gauche).
4. **Implémentation du système de matching**, avec un algorithme qui compare les compétences et l'expérience des candidats aux offres disponibles.
5. **Ajout des fonctionnalités de sécurité**, notamment l'authentification avec JWT et la gestion des utilisateurs avec Spring Security.
6. **Optimisation et correction des bugs**, notamment des problèmes liés aux tokens JWT, aux requêtes CORS et aux formats de données.

À chaque étape, j'ai dû ajuster certains choix techniques en fonction des problèmes rencontrés. Par exemple, j'ai eu des erreurs avec JWT mal formés et des soucis de compatibilité entre l'API Adzuna et MongoDB, que j'ai résolus en adaptant les requêtes et en optimisant le stockage des données.

Cette approche m'a permis d'avancer de manière structurée et de tester chaque fonctionnalité au fur et à mesure, tout en gardant une certaine flexibilité pour améliorer l'application au fil du développement.

1.3. Technologies utilisées

Spring Boot (Backend)

Spring Boot est un framework Java qui permet de créer des applications backend rapidement. Il est pratique car il gère automatiquement les configurations et offre des outils intégrés pour l'authentification, la gestion des bases de données et la communication avec des API externes.

Dans WorkMatch, Spring Boot est utilisé pour :

- Gérer les **utilisateurs** (candidats, employeurs).
- Communiquer avec l'**API Adzuna** pour récupérer les offres d'emploi.
- Implémenter le **système de matching** entre offres et candidats.
- Sécuriser l'application avec **JWT et Spring Security**.

API Adzuna (Données d'offres d'emploi en temps réel)

Adzuna est un assembleur d'offres d'emploi qui propose une API pour récupérer des annonces.

L'API Adzuna est utilisée pour :

- Récupérer les offres d'emploi en temps réel.
- Filtrer les offres en fonction du profil du candidat.
- Mettre à jour automatiquement les offres disponibles sur WorkMatch.

L'avantage, c'est que ça évite d'avoir à remplir manuellement une base de données d'offres d'emploi.

React Native (Frontend)

Pour le frontend, j'ai choisi React Native, qui permet de développer une interface en JavaScript et de la déployer à la fois sur Android et IOS.

React Native est utilisé pour :

- Afficher les offres d'emploi sous forme de swiping (like/dislike).
- Permettre aux candidats de voir leur historique de matching.
- Créer une interface moderne et fluide pour une bonne expérience utilisateur.

L'avantage de React Native, c'est qu'il permet d'avoir une seule base de code pour plusieurs plateformes, ce qui évite d'écrire une application séparée pour Android et iOS.

MongoDB (Base de données NoSQL)

MongoDB est une base de données NoSQL qui permet de stocker les informations sous forme de documents JSON plutôt qu'en tableaux classiques comme MySQL.

MongoDB est utilisé pour :

- Stocker les **profils** des utilisateurs (candidats et recruteurs).
- Sauvegarder les **offres d'emploi** récupérées depuis Adzuna.
- Enregistrer les **matchs** entre candidats et offres.

L’avantage de MongoDB, c’est qu’il gère bien les données **dynamiques**, ce qui est parfait pour un projet où les offres changent constamment.

2. Analyse

2.1. Situation existante (avant WorkMatch)

Aujourd’hui, le recrutement en ligne repose sur des plateformes comme LinkedIn, Indeed ou Adzuna. Ces plateformes permettent aux entreprises de publier des offres d’emploi et aux candidats de postuler. Cependant, ce modèle présente plusieurs limites :

- **Trop de candidatures non pertinentes** : Les employeurs reçoivent trop de CV qui ne correspondent pas aux attentes.
- **Recherche chronophage** : Les candidats passent des heures à trier et postuler manuellement.
- **Absence de matching intelligent** : Les plateformes fonctionnent comme des bases de données, mais ne proposent pas d’algorithme avancé de mise en relation.

2.2. Situation finale désirée (avec WorkMatch)

WorkMatch vise à révolutionner le processus de recrutement en proposant un système de matching automatisé qui simplifie la mise en relation entre les candidats et les employeurs.

Le système repose sur plusieurs critères :

a) Les compétences et expériences des candidats
Chaque candidat renseigne ses compétences, son niveau d’expérience et ses attentes professionnelles (secteur, type de contrat, salaire souhaité, localisation, etc.). Plutôt que d’afficher une liste d’offres génériques, l’application sélectionne celles qui correspondent précisément au profil du candidat.

b) Les besoins et critères des employeurs
Les entreprises définissent des critères précis pour leurs offres d’emploi (compétences requises, expérience minimale, diplômes nécessaires, etc.). WorkMatch prend en compte ces informations et les met en relation avec les profils des candidats, évitant ainsi aux recruteurs de recevoir des candidatures non pertinentes.

c) Un algorithme de matching intelligent
L’application ne se contente pas de comparer des mots-clés comme sur les autres plateformes. L’algorithme analyse la compatibilité entre un poste et un candidat en fonction de plusieurs facteurs :

- La correspondance des compétences avec celles demandées dans l'offre.
- L'expérience et les qualifications du candidat.
- Les préférences du candidat (type de contrat, localisation, salaire, etc.).
- Les attentes spécifiques de l'employeur.

Grâce à ce système, les candidats ne perdent plus de temps à postuler à des annonces qui ne leur conviennent pas, et les recruteurs trouvent plus rapidement des profils adaptés, améliorant ainsi la qualité des recrutements et réduisant le temps de sélection.

2.3. Étude préalable (discussion avec l'utilisateur)

Avant de développer WorkMatch, j'ai pris le temps d'analyser les attentes des utilisateurs et les problèmes qu'ils rencontrent sur les plateformes d'emploi classiques. L'objectif était de comprendre ce qui pourrait être amélioré et d'adapter l'application en conséquence.

Côté candidats

Les personnes en recherche d'emploi font face à plusieurs difficultés qui rendent le processus long et frustrant.

- **Des offres peu adaptées** : Sur les plateformes classiques, les candidats reçoivent souvent des annonces qui ne correspondent pas vraiment à leur profil. Il faut filtrer manuellement, ce qui prend du temps et peut décourager.
- **Trop d'étapes pour postuler** : Souvent, pour une seule offre, il faut remplir plusieurs champs, télécharger un CV, rédiger une lettre de motivation… et au final, il n'y a aucune garantie d'avoir une réponse.
- **Peu de retour sur les candidatures** : Après avoir postulé, il est difficile de savoir où en est la candidature. Les recruteurs ne répondent pas toujours, et les candidats se retrouvent à envoyer des CV en masse sans réelle visibilité.

Ce qu'ils attendent ?
1) Un système plus intelligent qui leur propose uniquement les offres réellement adaptées à leur profil et à leurs attentes.
2) Une interface fluide et intuitive, qui facilite le processus de candidature sans le rendre trop long.
3) Un meilleur suivi de leurs candidatures et une mise en relation plus directe avec les recruteurs.

Côté recruteurs

Les employeurs aussi rencontrent des difficultés lorsqu'ils publient des offres et cherchent des candidats.

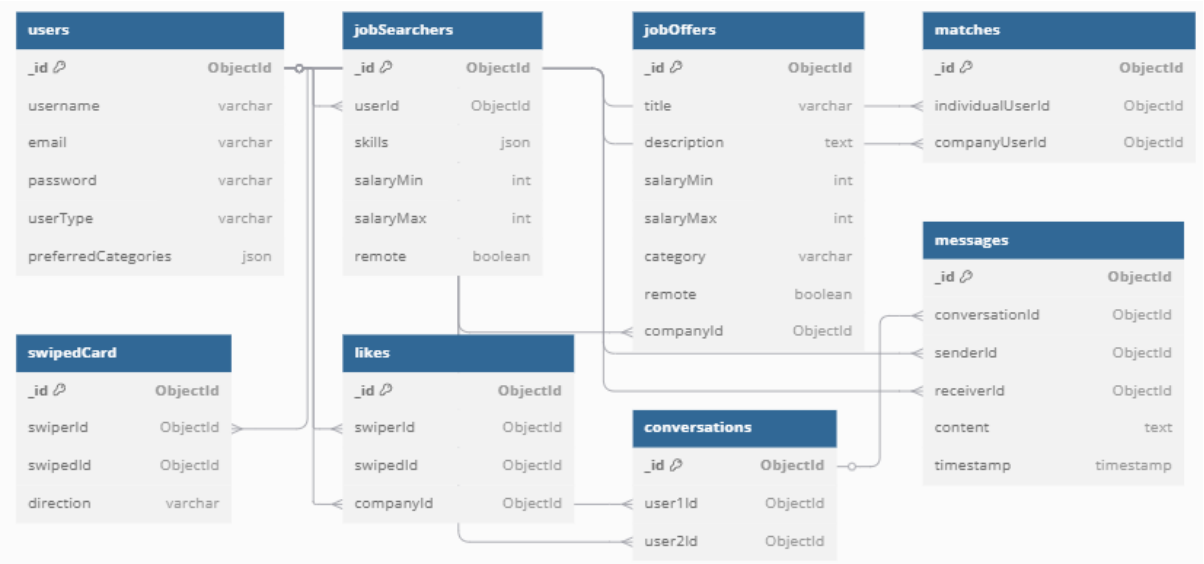
- **Un trop grand nombre de candidatures non pertinentes** : Lorsqu'une offre est mise en ligne, elle peut recevoir des centaines de candidatures, dont une majorité ne correspond pas aux attentes. Le tri manuel devient une tâche chronophage.
- **Pas de filtrage avancé** → La plupart des plateformes permettent aux candidats de postuler librement, sans prise en compte réelle des critères exigés par l'entreprise.
- **Des échanges parfois trop lents** → Une fois un bon profil repéré, il faut souvent passer par plusieurs étapes avant de pouvoir échanger avec le candidat, ce qui peut ralentir le processus de recrutement.

Ce qu'ils recherchent ?
1) Un système qui filtre automatiquement les candidatures en fonction des critères du poste.
2) Un accès rapide aux profils qui correspondent réellement aux besoins de l'entreprise.
3) Une messagerie intégrée pour pouvoir échanger rapidement avec les candidats pertinents.

Et voilà pourquoi j'ai voulu faire WorkMatch : un système où tout le monde gagne du temps. Les candidats voient direct les offres qui leur correspondent, les recruteurs trouvent des profils adaptés sans trier 200 CVs.

2.4. Analyse technique

Modèle Conceptuel des Données (MCD)



Présentation des entités principales

L'application WorkMatch utilise une base de données **MongoDB**, qui stocke les informations sous forme de documents JSON. Voici les principales collections et leur rôle :

- **users** : Contient les informations des utilisateurs, qu'ils soient candidats ou employeurs. Chaque utilisateur a un **userType**, qui peut être **INDIVIDUAL** (candidat) ou **COMPANY** (employeur).

- **jobSearchers** : Cette collection est dédiée aux candidats et est liée obligatoirement à un utilisateur de type **INDIVIDUAL**. Elle stocke des informations spécifiques comme les **compétences** (skills), la fourchette de salaire (salaryMin, salaryMax), et la **localisation**.
- **jobOffers** : Représente les offres d'emploi publiées sur la plateforme. Chaque offre contient un titre (title), une description (description), une fourchette salariale (salaryMin, salaryMax), la catégorie du poste (category), et un champ companyId qui lie l'offre à une entreprise (un utilisateur de type **COMPANY**).
- **matches** : Enregistre les correspondances entre un candidat et une offre d'emploi. Chaque entrée associe un utilisateur individuel (individualUserId) et une entreprise (companyUserId).
- **messages** : Contient les messages échangés entre utilisateurs après un match réussi. Chaque message appartient à une **conversation** via le champ conversationId et possède un expéditeur (senderId) et un destinataire (receiverId).
- **conversations** : Gère les discussions entre deux utilisateurs. Chaque conversation est définie par user1Id et user2Id, qui représentent les participants.
- **likes** : Enregistre les actions des utilisateurs lorsqu'ils interagissent avec des offres ou des candidats. Le champ swiperId identifie l'utilisateur qui interagit, tandis que swipedId stocke l'identifiant de l'offre ou du candidat concerné.
- **swipedCard** : Stocke les actions de **swiping** des utilisateurs (droite/gauche pour liker ou passer une offre). Il enregistre l'identifiant du **swiper** (swiperId), l'offre ou le candidat liké (swipedId), ainsi que la **direction** du swipe (direction).

Relations et contraintes

Le modèle de données repose sur plusieurs relations essentielles entre les collections :

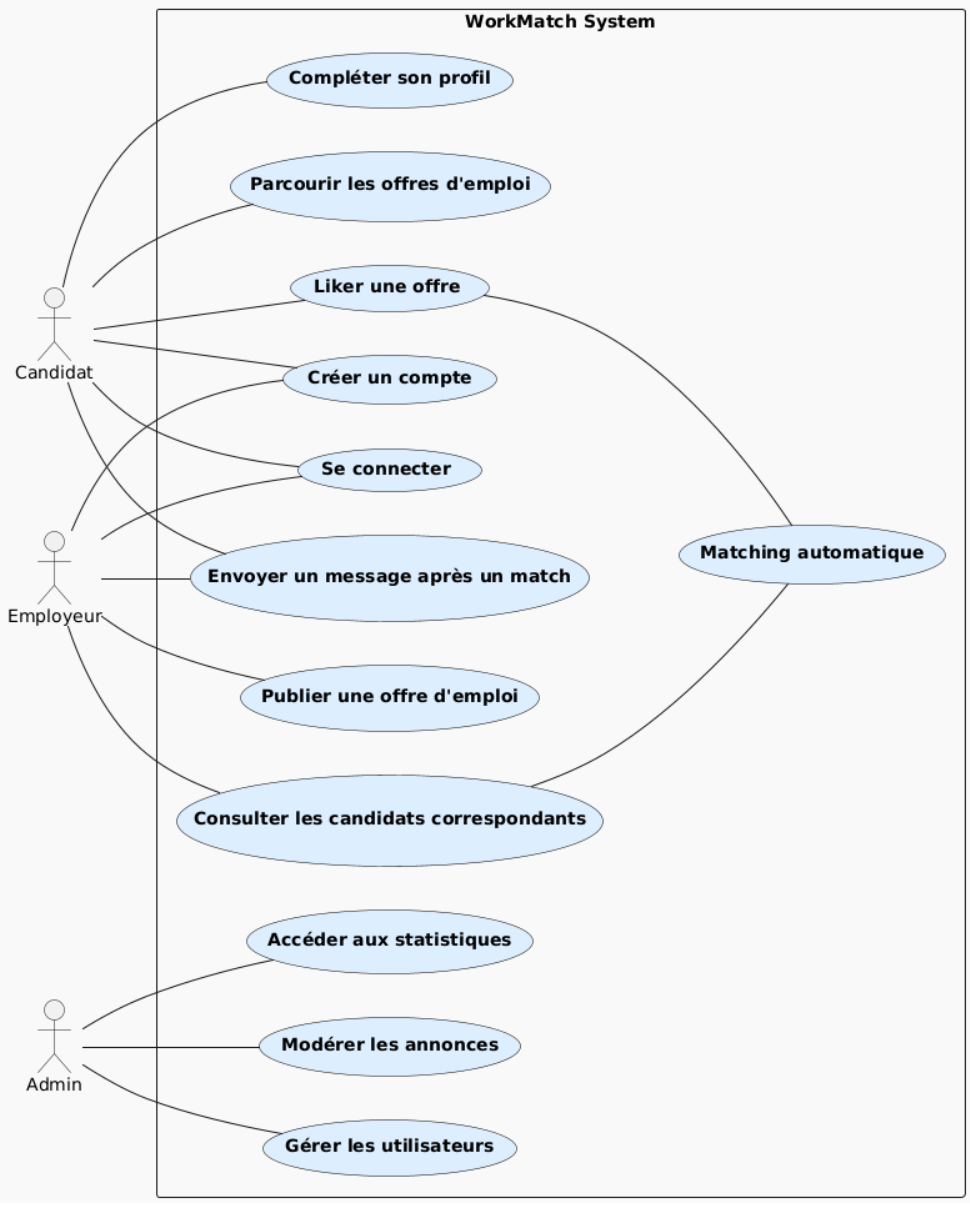
1. **Relation entre users, jobSearchers et jobOffers**
    a. Chaque utilisateur de type **INDIVIDUAL** doit être associé à un document dans jobSearchers, ce qui signifie qu'un candidat **existe à la fois dans users et jobSearchers**.
    b. Chaque utilisateur de type **COMPANY** doit être l'auteur d'une ou plusieurs **offres d'emploi** enregistrées dans jobOffers (via le champ companyId).
2. **Le système de matching (matches, likes, swipedCard)**
    a. Un candidat peut **swiper** une offre d'emploi via swipedCard, ce qui est enregistré avec un swiperId (l'utilisateur) et un swipedId (l'offre ou le candidat concerné).
    b. Lorsqu'un employeur **like** un candidat, cela est enregistré dans likes avec le swiperId comme employeur et swipedId comme candidat.
    c. Un match est créé si **un employeur et un candidat se sont mutuellement likés**.

3. **Les interactions (conversations, messages)**

a. Après un match réussi, une conversation est automatiquement créée avec user1Id et user2Id.
b. Chaque message envoyé appartient à une conversation (conversationId) et est défini par son senderId et receiverId.

4. **Contraintes techniques**
   a. **Dépendance à l'API Adzuna** : Toutes les offres d'emploi doivent provenir de **l'API Adzuna**, ce qui impose un traitement des données pour adapter leur structure à celle de **jobOffers**.
   b. **Performances et stockage** : Chaque interaction utilisateur (swipe, like, match) génère une entrée en base de données, ce qui peut entraîner une accumulation rapide de données nécessitant une **optimisation du stockage et de l'indexation**.
   c. **Sécurité des utilisateurs** : Les mots de passe sont stockés sous forme hachée dans users, et des mécanismes comme **JWT** sont utilisés pour sécuriser l'authentification.
   d. **Gestion des relations en base NoSQL** : Contrairement aux bases relationnelles, MongoDB ne propose pas de jointures natives (JOIN). Il est donc essentiel de bien structurer les documents pour éviter de devoir faire trop de requêtes séparées, ce qui pourrait ralentir l'application. Par exemple, dans WorkMatch, chaque offre d'emploi contient un companyId, ce qui permet d'éviter de stocker l'entreprise en double et de récupérer ses informations uniquement quand c'est nécessaire. De plus, pour assurer un bon fonctionnement du système de matching et de messagerie, les requêtes doivent être optimisées avec des index et une bonne gestion des relations entre les documents.

Cas d'utilisation



Le diagramme de cas d'utilisation ci-dessus illustre les interactions entre les différents acteurs de l'application WorkMatch et les fonctionnalités principales du système.

1. Présentation des Acteurs

L'application WorkMatch repose sur trois types d'utilisateurs :

- **Candidat** : Un chercheur d'emploi qui crée un profil et recherche des offres adaptées.
- **Employeur** : Un recruteur ou une entreprise qui publie des offres et sélectionne des candidats.
- **Admin** : Un modérateur qui gère le bon fonctionnement de la plateforme.

2. Cas d'Utilisation

Candidat :

1. **Créer un compte** : S'inscrire sur la plateforme.
2. **Se connecter** : Accéder à son espace personnel.
3. **Compléter son profil** : Ajouter ses compétences, expériences et préférences.
4. **Parcourir les offres d'emploi** : Voir les postes disponibles récupérés depuis Adzuna.
5. **Liker une offre** : Indiquer son intérêt pour un emploi, ce qui déclenche un matching.
6. **Envoyer un message après un match** : Discuter avec un recruteur si un match est confirmé.

Employeur :

1. **Créer un compte** : S'inscrire en tant qu'entreprise.
2. **Se connecter** : Accéder au tableau de bord employeur.
3. **Publier une offre d'emploi** : Ajouter un poste à pourvoir.
4. **Consulter les candidats correspondants** : Voir les profils des candidats qui matchent avec l'offre.
5. **Envoyer un message après un match** : Contacter directement un candidat intéressé.

Admin :

1. **Modérer les annonces** : Vérifier et valider les offres publiées.
2. **Gérer les utilisateurs** : Supprimer ou suspendre un compte en cas d'abus.
3. **Accéder aux statistiques** : Suivre l'évolution de la plateforme et les performances du matching.

3. Fonctionnalités Spécifiques

Matching automatique :

- Se déclenche **uniquement** lorsqu'un **candidat (userType = INDIVIDUAL) like une offre d'emploi**, puis que **l'employeur dont l'offre a été likée like aussi le candidat**.
- Lorsqu'il y a un **match mutuel**, un document est ajouté dans la collection `matches`.
- À partir de là, une **conversation** est créée pour permettre aux deux parties d'échanger.

Contraintes et Challenges Techniques :

- **Gestion des données en temps réel** : Il faut que le système récupère et mette à jour les offres via l'API Adzuna.
- **Optimisation du matching** : L'algorithme doit être rapide pour éviter de ralentir l'expérience utilisateur.
- **Sécurité et protection des données** : Les messages envoyés entre les utilisateurs doivent être chiffrés.

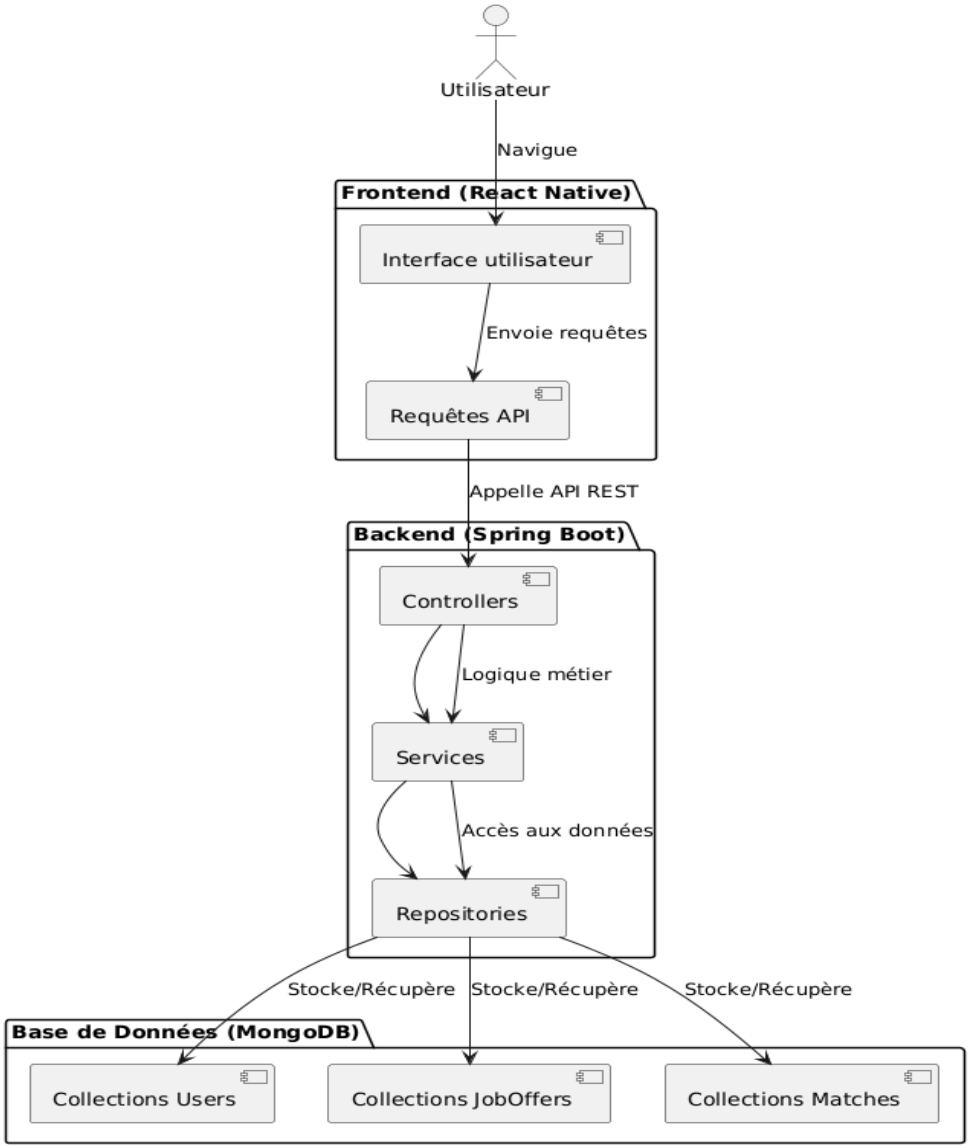
2.5. Architecture de l'application

Présentation générale

L'application WorkMatch est **composée de trois couches principales** :

- **Frontend (React Native)** : Interface utilisateur, interactions avec les API
- **Backend (Spring Boot)** : Logique métier, gestion des utilisateurs, des offres et des interactions
- **Base de Données (MongoDB)** : Stockage des utilisateurs, des offres d'emploi, des matchs et des conversations

Schéma exemple de l'architecture globale :



Le Backend (Spring Boot)

Le backend est structuré de manière modulaire avec les composants suivants :

- **Controller** : Reçoit les requêtes HTTP et renvoie les réponses
- **Service** : Contient la logique métier
- **Repository** : Interagit avec la base de données (MongoDB)
- **Model** : Décrit les entités de la base de données

Exemple de requête backend (JobSearcherController.java) :

```java
@RestController  no usages  unknown *
@RequestMapping("/jobsearchers")
@CrossOrigin(origins = "http://localhost:8081")
public class JobSearcherController {

    @Autowired  4 usages
    private JobSearcherService jobSearcherService;

    @GetMapping("/matching")  no usages  new *
    public List<JobSearcher> getMatchingCandidates(@RequestParam String jobOfferId) {
        return jobSearcherService.findMatchingCandidates(jobOfferId);
    }
```

Dans l'exemple ci-dessus, le contrôleur récupère les candidats correspondant à une offre d'emploi via un service.

Exemple d'implémentation dans le service (JobSearcherService.java) :

```java
@Service  1 usage  unknown *
public class JobSearcherService {

    private final JobSearcherRepository jobSearcherRepository;  4 usages

    private final JobOfferRepository jobOfferRepository;  2 usages

    public List<JobSearcher> findMatchingCandidates(String jobOfferId) {  no usages  new *
        Optional<JobOffer> jobOfferOpt = jobOfferRepository.findById(jobOfferId);
        if (jobOfferOpt.isEmpty()) {
            System.out.println("❌ Aucune offre trouvée pour l'ID : " + jobOfferId);
            return List.of();
        }
        JobOffer jobOffer = jobOfferOpt.get();
        System.out.println("🔍 Offre trouvée : " + jobOffer.getTitle());

        if (jobOffer.getSkills() == null || jobOffer.getSkills().isEmpty()) {
            System.out.println("⚠️ Aucune compétence requise pour cette offre.");
            return List.of();
        }
```

```java
public List<JobSearcher> findMatchingCandidates(String jobOfferId) {  no usages  new *
    Optional<JobOffer> jobOfferOpt = jobOfferRepository.findById(jobOfferId);
    if (jobOfferOpt.isEmpty()) {
        System.out.println("❌ Aucune offre trouvée pour l'ID : " + jobOfferId);
        return List.of();
    }
    JobOffer jobOffer = jobOfferOpt.get();
    System.out.println("🔍 Offre trouvée : " + jobOffer.getTitle());

    if (jobOffer.getSkills() == null || jobOffer.getSkills().isEmpty()) {
        System.out.println("⚠️ Aucune compétence requise pour cette offre.");
        return List.of();
    }

    System.out.println("📜 Compétences requises pour l'offre : " + jobOffer.getSkills());

    List<JobSearcher> matchingCandidates =
            jobSearcherRepository.findAll()
                    .stream()
                    .filter(js -> js.getSkills() != null)
                    .filter(js -> js.getSkills()
                            .stream()
                            .anyMatch(skill -> jobOffer.getSkills()
                                    .stream()
                                    .anyMatch(reqSkill ->
                                            skill.getName().equalsIgnoreCase(reqSkill.getName()) &&
                                                    skill.getExperience() >= reqSkill.getExperience()
                                    )
                            )
                    )
                    .collect(Collectors.toList());
    System.out.println("🔍 Vérification des compétences pour l'offre : " + jobOffer.getTitle());
    System.out.println("Compétences requises : " + jobOffer.getSkills());

    System.out.println("✅ Nombre de candidats correspondants : " + matchingCandidates.size());
    return matchingCandidates;
}
```

Dans les captures d'écrans ci-dessus, le service filtre les candidats en fonction des compétences requises par l'offre d'emploi.

La Sécurité et la Gestion des Tokens

Pour sécuriser les échanges entre le frontend et le backend, l'application utilise **JWT (JSON Web Token)**.

Extrait de la configuration de sécurité (à améliorer) :

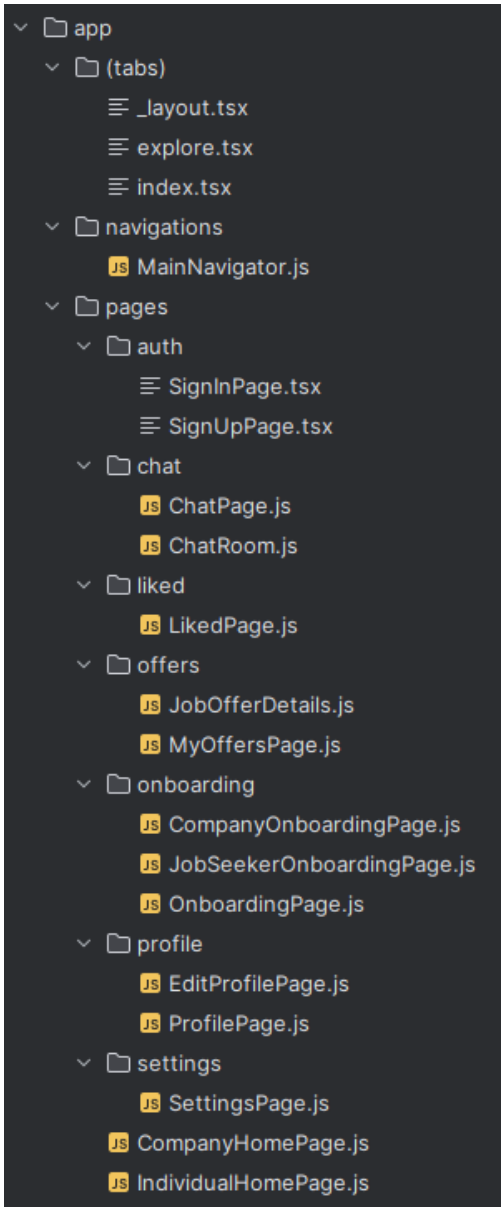
```java
@Configuration  no usages  unknown *
@EnableWebSecurity
public class SecurityConfig {

    @Bean  no usages  new *
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .csrf(csrf -> csrf.disable())
                .authorizeHttpRequests(auth -> auth
                        .anyRequest().permitAll()
                )
                .headers(headers -> headers.frameOptions(frameOptions -> frameOptions.disable()));

        return http.build();
    }
```

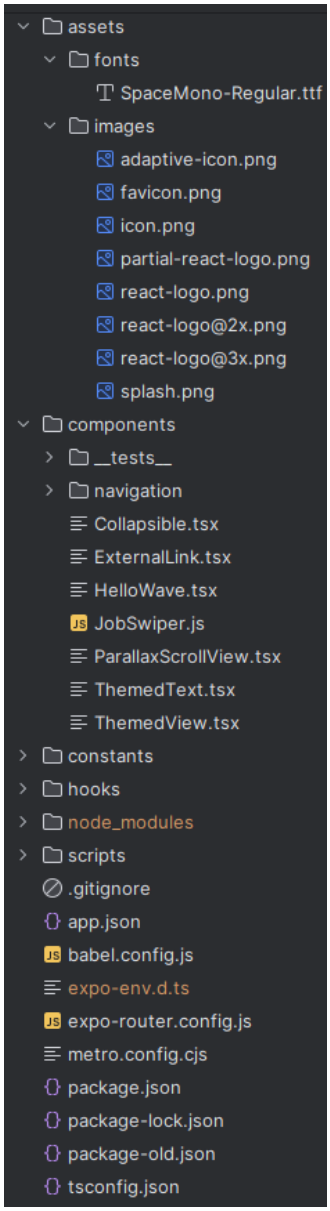
Dans l'exemple ci-dessus, on désactive CSRF, on applique une **authentification obligatoire** et on ajoute un filtre JWT.

Le Frontend (React Native)

Le frontend est développé avec **React Native** et utilise **Axios** pour les appels API.

Arborescence des fichiers du frontend :


app
(tabs)
_layout.tsx
explore.tsx
index.tsx
navigations
MainNavigator.js
pages
auth
SignInPage.tsx
SignUpPage.tsx
chat
ChatPage.js
ChatRoom.js
liked
LikedPage.js
offers
JobOfferDetails.js
MyOffersPage.js
onboarding
CompanyOnboardingPage.js
JobSeekerOnboardingPage.js
OnboardingPage.js
profile
EditProfilePage.js
ProfilePage.js
settings
SettingsPage.js
CompanyHomePage.js
IndividualHomePage.js



Exemple d’appel API pour récupérer les candidats (CompanyHomePage.js) :

```
const fetchMatchingCandidates = async (jobOffer) => {
    if (!jobOffer || !jobOffer._id) {
        console.error("❌ Erreur : jobOffer ou son ID est invalide !");
        return;
    }

    try {
        const token = await AsyncStorage.getItem("userToken");
        console.log("🔍 Chargement des candidats pour :", jobOffer.title);

        const response = await axios.get(`http://localhost:8080/jobsearchers/matching?jobOfferId=${jobOffer._id}`, {
            headers: { Authorization: `Bearer ${token}` },
        });

        setMatchingJobSearchers(response.data);
        console.log("✅ Candidats correspondants :", response.data);
    } catch (error) {
        console.error("❌ Erreur lors du chargement des candidats :", error);
    } finally {
        setIsLoading(false);
    }
};
```

Dans la capture d'écran ci-dessus, une **requête GET** est envoyée pour récupérer les candidats correspondant à une offre.

La base de données (MongoDB)

WorkMatch utilise MongoDB, une base NoSQL qui stocke les **données sous forme de documents JSON**.

Exemple de modèle MongoDB (User.java) :

```
@Document(collection = "users")
public class User {
    @Id
    private String id;

    private String username;
    private String email;
    private String password;
    private List<String> skills;
    private UserType userType;
    private List<String> preferredCategories = new ArrayList<>();
```

Dans le screenshot ci-dessus, la classe **User** représente un utilisateur stocké dans la base.

Requête MongoDB pour trouver un utilisateur par ID (UserRepository.java) :

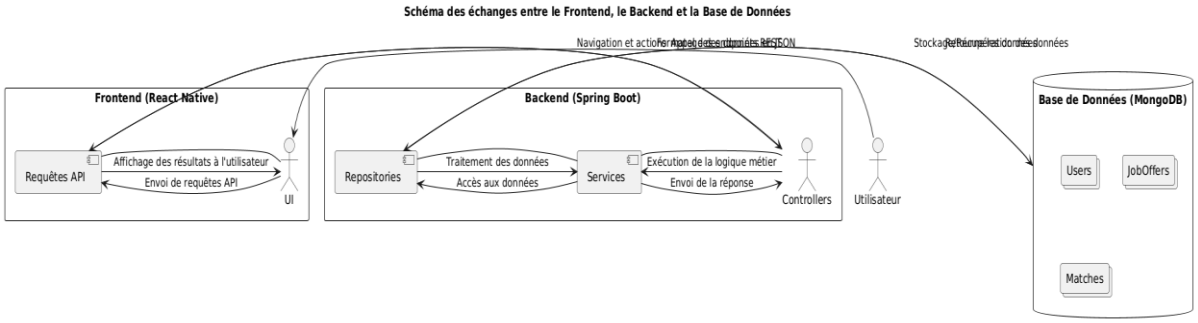
```java
@Repository  6 usages  unknown *
public interface UserRepository extends MongoRepository<User, String> {

    Optional<User> findById(String id);  new *
```

Cette méthode permet de récupérer un utilisateur à partir de son **ID**.

Les échanges entre les couches

1. **Un utilisateur se connecte sur le frontend**
    a. Son mot de passe est envoyé au backend via une requête **POST**
    b. Le backend valide ses identifiants et génère un **JWT**
    c. Ce token est stocké dans le **local storage** du frontend
2. **L'utilisateur consulte les offres**
    a. Le frontend envoie une requête **GET** au backend pour récupérer les offres
    b. Le backend interroge MongoDB et retourne la liste des offres
    c. Le frontend affiche les offres sous forme de **cartes swipables**
3. **Un candidat aime une offre**
    a. Le frontend envoie une requête **POST** pour enregistrer un **like**
    b. Le backend vérifie si l'offre a déjà été likée par l'entreprise
    c. Si oui, un **match** est enregistré dans la base

Schéma des échanges frontend-backend-database :



Avantages et Inconvéneints

Avantages

- **Modularité** : Chaque couche a un rôle précis, ce qui facilite la maintenance
- **Scalabilité** : Le backend peut évoluer sans impacter le frontend
- **Sécurité** : JWT assure une **authentification sécurisée**
- **Optimisation des requêtes** : MongoDB stocke les **relations directement dans les documents**

Inconvénients

- **Complexité accrue** : Plusieurs technologies sont utilisées (React Native, Spring Boot, MongoDB)
- **Gestion des relations** : MongoDB ne supporte pas les **jointures natives**, il faut structurer les requêtes pour éviter les lenteurs

4. Appréciation personnelle

4.1. Rapport avec les cours

Quels cours ont aidé

4.2. Difficultés rencontrées

Problèmes techniques

Résolution

4.3. Suggestions et améliorations

Si je devais refaire, que changerais je ?

Idées d'améliorations pour l'avenir

5. Conclusion

5.1. Bilan du projet

Ai-je atteint les objectifs ?

5.2. Analyse des résultats obtenus

Retour sur performance de l'application

5.3. Impact du projet sur le marché du travail