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.

L'environnement de développement a été configuré avec les outils standards adaptés au projet : JDK 17, IntelliJ IDEA pour le backend Spring Boot, et Expo CLI associé à Node.js pour le frontend React Native Web. Le backend a été initialisé via Spring Initializr avec les dépendances nécessaires (Spring Web, Spring Data MongoDB), tandis que la gestion de la base de données locale a été assurée par MongoDB Community Edition avant le passage sur MongoDB Atlas pour la production.

Des outils comme Postman ont été utilisés pour tester les endpoints REST, tandis que MongoDB Compass a facilité la visualisation des données. Axios a été intégré au frontend pour assurer la communication avec l'API.

Les outils IntelliJ IDEA, Android Studio et Postman ont été configurés dès le début afin d'assurer un environnement stable pour le développement et les tests.

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

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

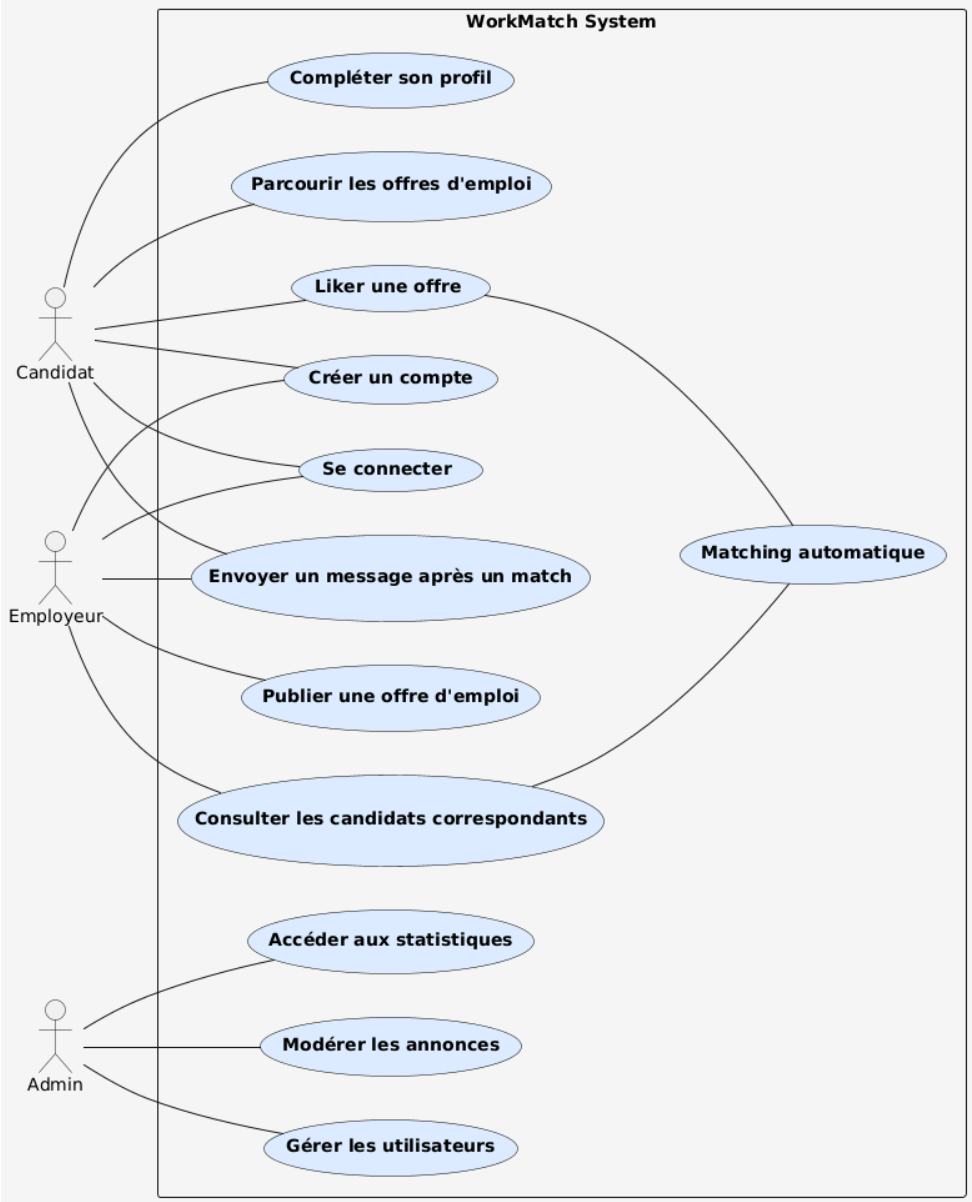
Le modèle de données repose sur plusieurs relations essentielles entre les collections :

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.

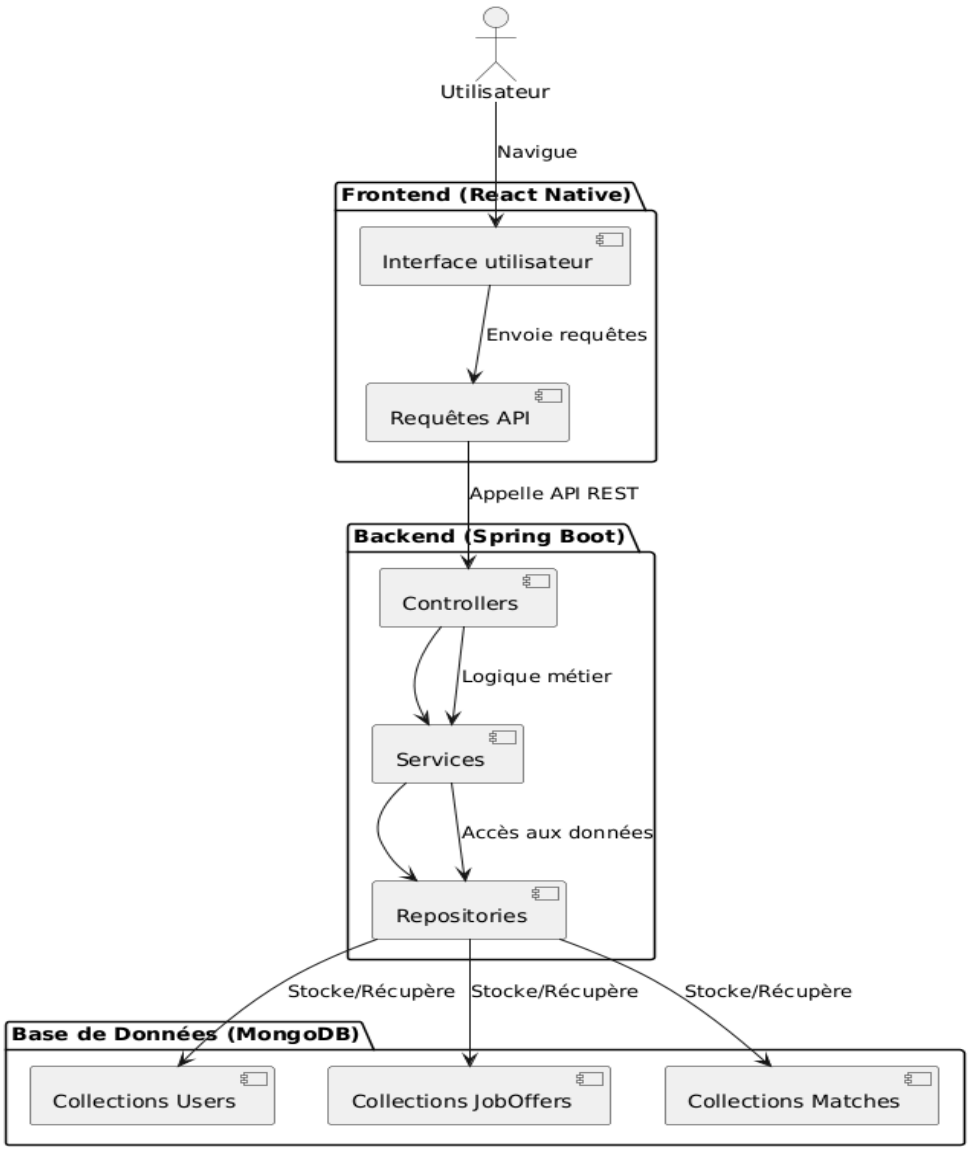
2.5. Architecture de l’application

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

La structure suit une architecture classique en couches avec séparation des modèles, repositories, services et contrôleurs, assurant une bonne maintenabilité du code.

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.

Le stockage local des tokens JWT a été géré via **AsyncStorage**, permettant de conserver les informations de session utilisateur de manière sécurisée et d'assurer une reconnexion automatique sans nécessiter une nouvelle authentification à chaque ouverture de l'application.

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 :

```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.

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) :

```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) :

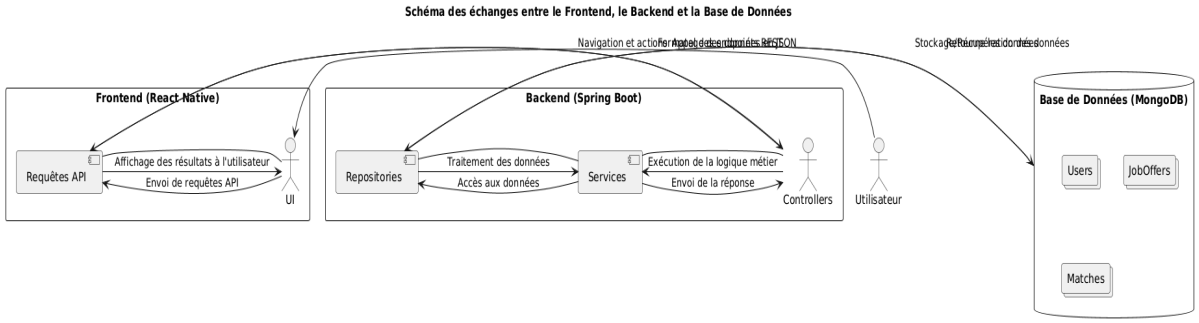
```java
@Repository  6 usages  unknown *
public interface UserRepository extends MongoRepository<User, String> {

    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

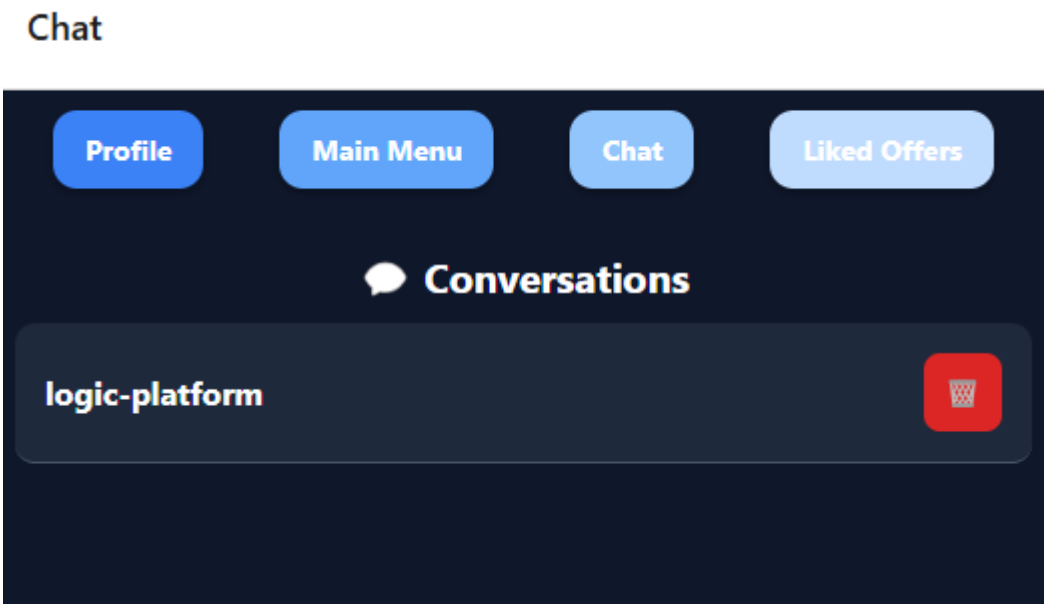
- **Complexité accrue** : Plusieurs technologies sont utilisées (React Native, Spring Boot, MongoDB)
- **Gestion des relations** : MongoDB ne supporte pas les **jointures natives**, il faut structurer les requêtes pour éviter les lenteurs

Schémas de l'interface utilisateur

Pour que l'application soit facile à utiliser, j'ai essayé de faire une interface simple et intuitive. J'ai utilisé des composants de base avec un design assez moderne, mais sans trop de complexité.

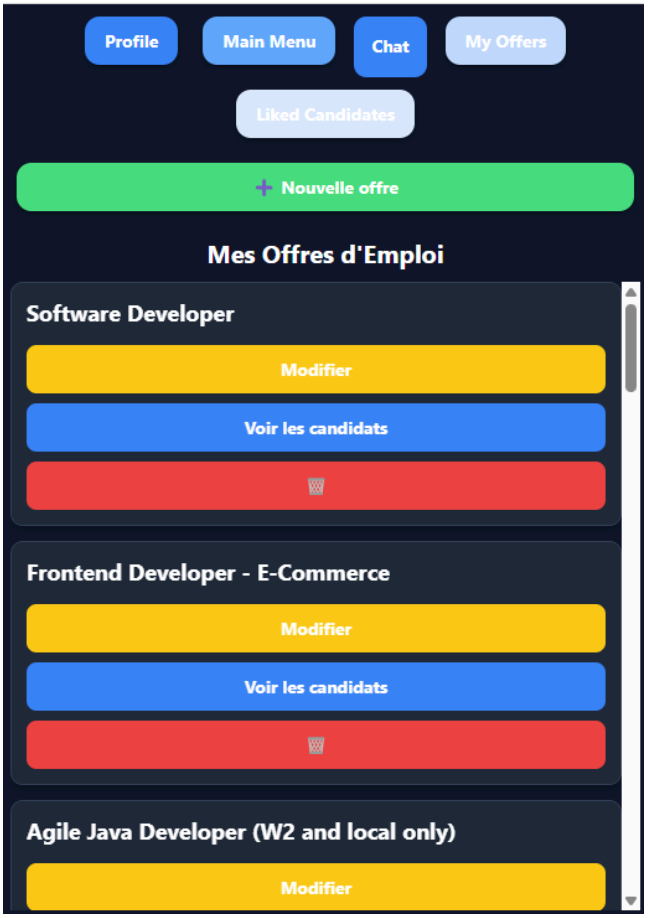
L'utilisateur qui cherche un emploi arrive directement sur une page avec des cartes d'offres d'emploi qu'il peut swiper à droite pour liker ou à gauche pour passer. Côté entreprise, l'interface permet de voir des profils de candidats et de faire la même chose.

Page de chat après un match :



Page de gestion des offres pour les entreprises :

My Offers



2.6. Contraintes de l'environnement

Contraintes techniques

Lors de la conception de WorkMatch, plusieurs choix techniques ont été guidés par la nature du projet et les objectifs à atteindre. Il était important d'opter pour des technologies qui permettent à la fois rapidité de développement, flexibilité et compatibilité avec les besoins spécifiques d'une application moderne.

- **Spring Boot** : J'ai choisi Spring Boot pour le backend car il offre une structure robuste et modulaire permettant de développer des API REST sécurisées et performantes. Il facilite l'intégration de composants comme la sécurité (Spring Security), la gestion des WebSockets pour le chat en temps réel, et permet une bonne gestion des dépendances avec Maven. De plus, sa communauté est très active, ce qui garantit un bon support en cas de problème.
- **MongoDB** : La base de données NoSQL s'est imposée naturellement pour ce projet. Avec des entités comme les offres d'emploi, les utilisateurs, les compétences et les

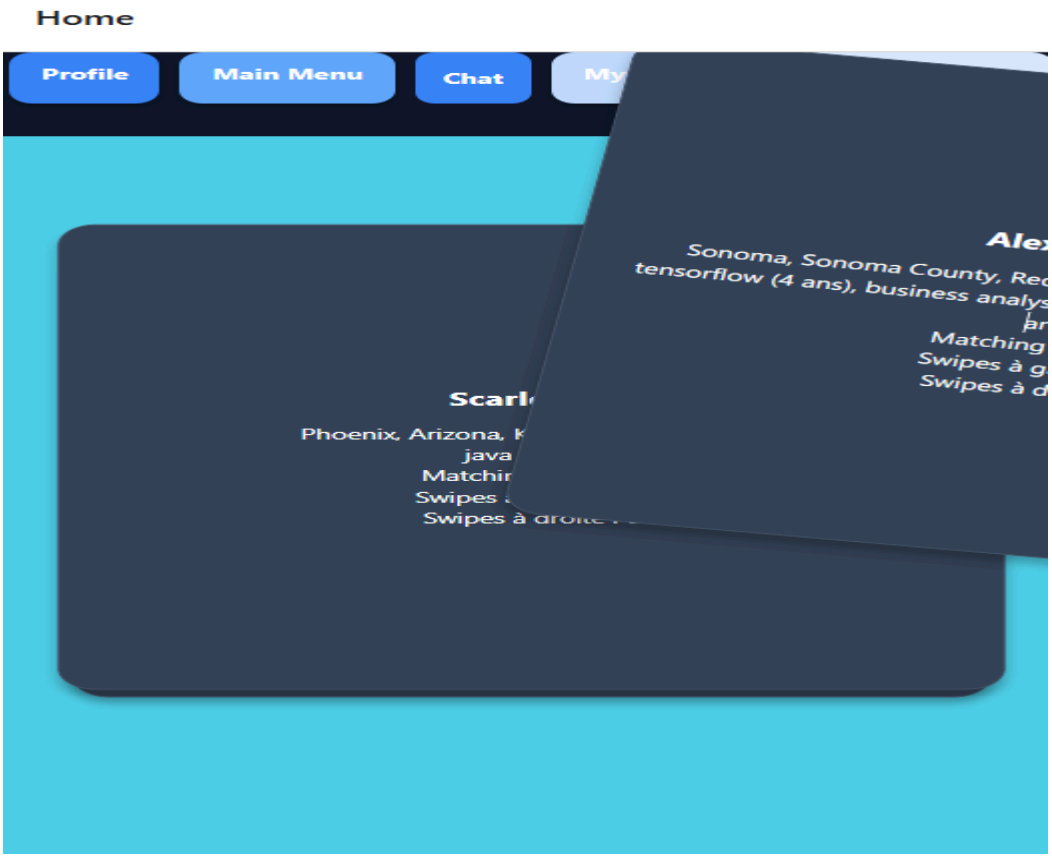
préférences, un schéma flexible était idéal. MongoDB permet de stocker des documents JSON dynamiques, ce qui simplifie la gestion des données sans avoir à modifier un schéma strict comme dans une base SQL à chaque évolution du projet.

- **React Native Web** : Pour le frontend, le choix s'est porté sur React Native Web car il permet de développer une interface responsive et moderne, tout en restant compatible avec une future adaptation mobile si besoin. Grâce à ses composants réutilisables et sa gestion efficace des états, il a été possible de créer une expérience utilisateur fluide, notamment pour le système de swipe des offres.
- **Gestion du temps réel avec WebSocket** : L'intégration du chat en temps réel était une contrainte importante. J'ai utilisé **SockJS** et **STOMP** avec Spring Boot pour établir une communication bidirectionnelle fiable entre le serveur et les clients. Ce système permet d'envoyer instantanément des notifications lorsqu'un match est détecté ou lorsqu'un message est reçu.

Ces choix techniques ont permis d'assurer que l'application soit évolutive, facilement maintenable et capable de répondre aux besoins des utilisateurs, tant du côté employeur que du côté candidat.

L'environnement de développement a été soigneusement configuré avec JDK 17, IntelliJ IDEA pour le backend, Expo CLI pour le frontend.

Exemple du **système de swipe** dans l'interface React Native Web :

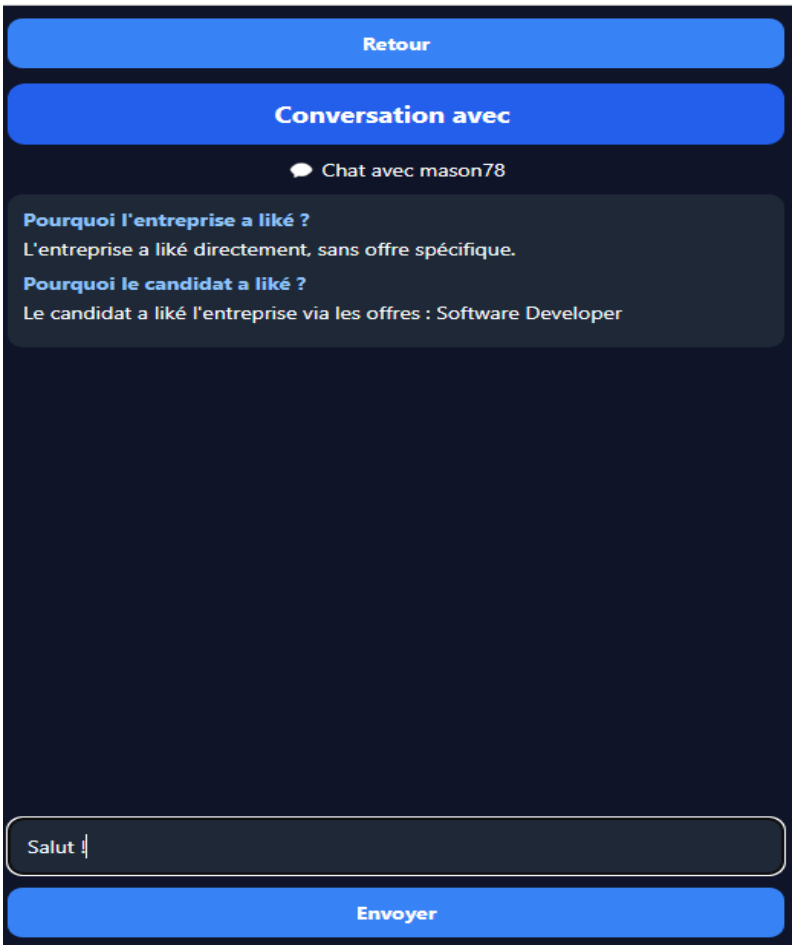


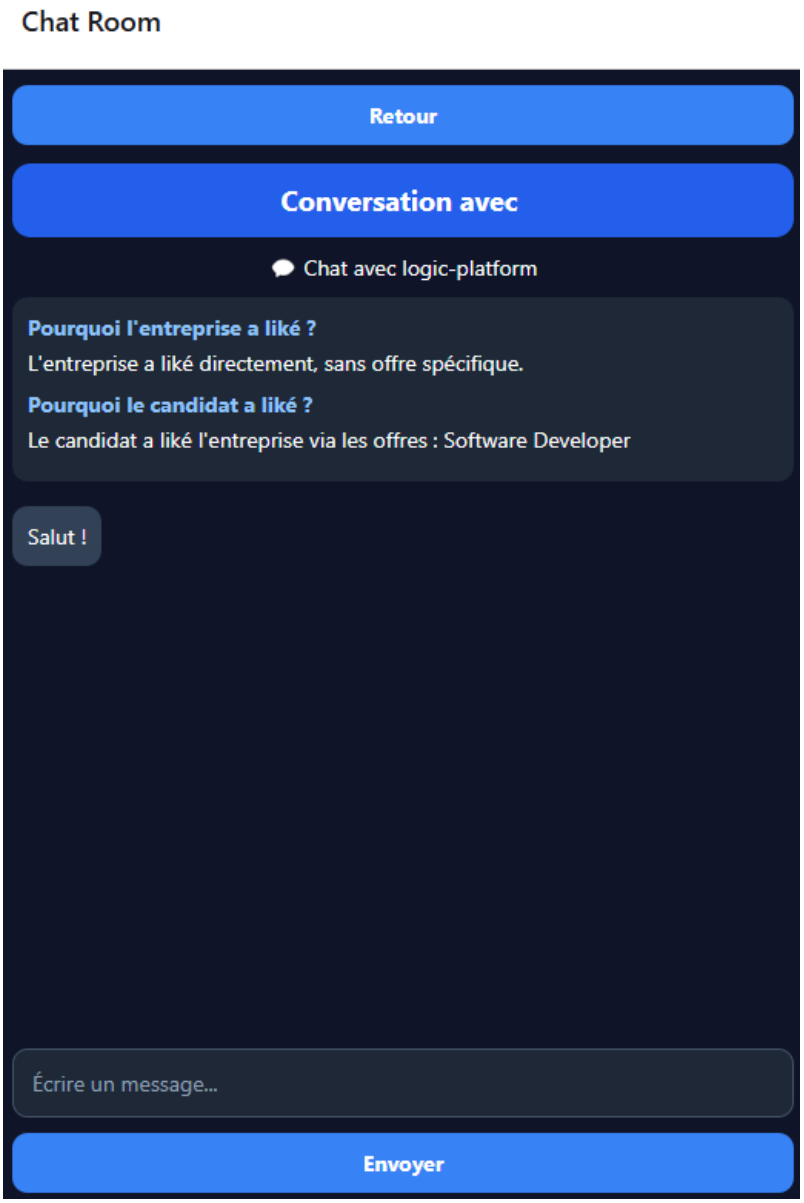
Extrait de configuration WebSocket côté backend :

```java
@Configuration  no usages  unknown
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Autowired  no usages
    private JwtUtil jwtUtil;

    @Override  no usages  unknown
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/ws")
                .setAllowedOriginPatterns("*")
                .withSockJS();

    }

    @Override  no usages  unknown
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/topic");
        registry.setApplicationDestinationPrefixes("/app");
    }
}
```

Capture de chat temps réel utilisation WebSocket :

Chat Room


Retour
Conversation avec
Chat avec mason78
Pourquoi l'entreprise a liké ?
L'entreprise a liké directement, sans offre spécifique.
Pourquoi le candidat a liké ?
Le candidat a liké l'entreprise via les offres : Software Developer
Salut !
Envoyer



Au cours du développement frontend, des conflits récurrents avec certaines dépendances ont conduit à migrer la gestion des packages de **npm** vers **yarn**, offrant une meilleure stabilité et une résolution plus efficace des erreurs liées aux modules React Native.

Contraintes de sécurité

La sécurité a été une partie importante du développement de WorkMatch, notamment pour protéger les données des utilisateurs et sécuriser l'accès aux fonctionnalités sensibles. J'ai mis en place une authentification basée sur les **JWT (JSON Web Token)** afin de garantir que seules les personnes autorisées puissent accéder aux routes privées de l'application.

Les mots de passe des utilisateurs sont hachés avec l'algorithme **BCrypt** avant d'être enregistrés dans la base de données, ce qui évite tout stockage de mots de passe en clair. Cela permet de renforcer la protection en cas de fuite de données.

Pour assurer un bon niveau de sécurité, plusieurs bonnes pratiques ont été suivies :

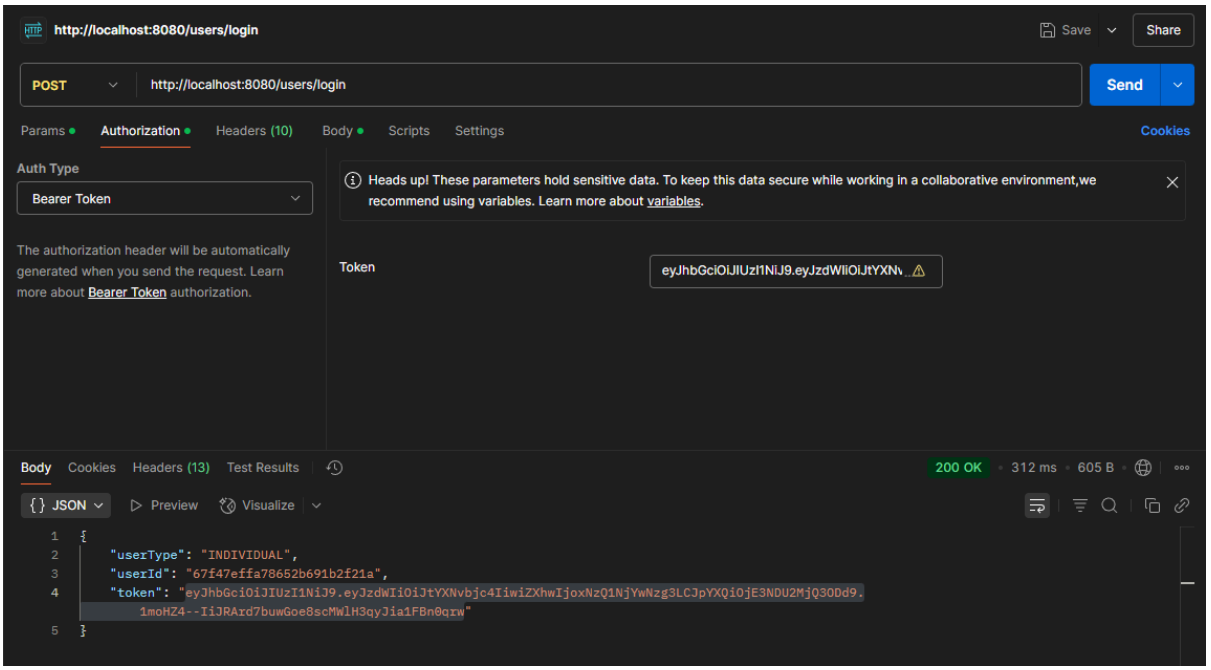
- Les mots de passe sont systématiquement hachés avant toute sauvegarde.
- Chaque requête vers une route protégée vérifie la validité du token JWT.
- La configuration **CORS** a été définie pour limiter les origines autorisées à communiquer avec l'API, évitant ainsi des accès non autorisés depuis des domaines externes.

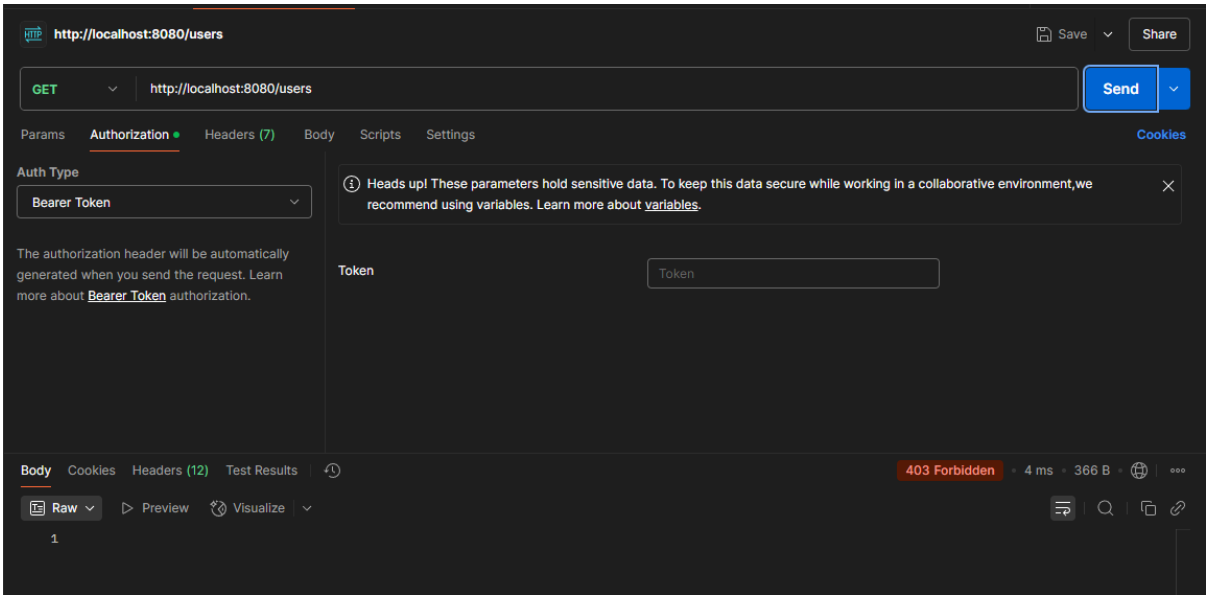
Ces mesures permettent de sécuriser l'authentification, la gestion des sessions et les échanges entre le frontend et le backend.

Extrait code de génération du token :

```
public String generateToken(String username) {  no usages  unknown
    Map<String, Object> claims = new HashMap<>();
    return Jwts.builder()
            .setClaims(claims)
            .setSubject(username)
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10))
            .signWith(SignatureAlgorithm.HS256, secret)
            .compact();
}
```

Des captures d’écran Postman accès à route protégée avec et sans fournir de token :





Contraintes de performance

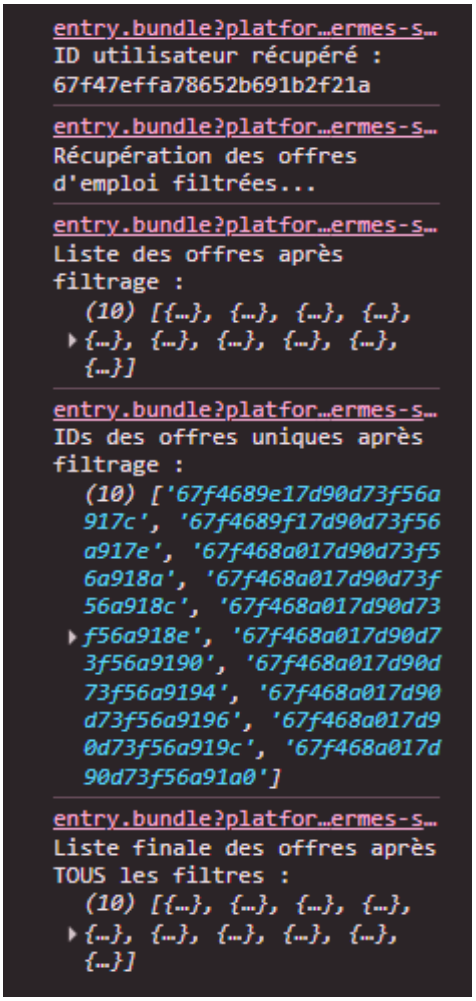
L'une des contraintes était de faire en sorte que l'application ne rame pas, surtout avec le système de swipe et les données venant de l'API Adzuna. J'ai dû limiter le nombre d'offres chargées d'un coup avec de la pagination.

Le backend devait aussi répondre rapidement, même si je l'avais hébergé sur une plateforme gratuite (Render), qui peut parfois être lente au démarrage.

J'ai donc optimisé en :

- Chargeant les offres 10 par 10.
- Réduisant les appels inutiles.
- Faisant attention à la taille des données échangées.

Capture d'écran du console.log montrant le chargement par 10 :



2.7. Démarche et choix conceptuels

Dès le début du projet, j'ai voulu créer une application simple mais efficace, en me concentrant sur l'essentiel : mettre en relation des candidats et des entreprises facilement. Pour ça, j'ai dû faire plusieurs choix techniques et stratégiques.

Je ne voulais pas réinventer la roue, donc j'ai préféré utiliser des technologies modernes, connues pour leur efficacité et leur rapidité de mise en place. L'idée était de construire une application fonctionnelle sans perdre trop de temps sur des détails complexes.

J'ai aussi cherché à rendre l'application **flexible** et **évolutive**. Par exemple, j'ai choisi une base de données NoSQL pour pouvoir adapter facilement la structure des données si besoin (comme ajouter des compétences, des préférences, etc.).

Enfin, je voulais que l'application soit **accessible** depuis n'importe quel appareil, c'est pourquoi j'ai utilisé une technologie compatible web et mobile.

L'approche générale a été :

1. **Analyser les besoins** : Qu'est-ce qu'un utilisateur (candidat ou entreprise) attend d'une telle application ?
2. **Choisir des outils adaptés** : Privilégier des frameworks rapides à mettre en place et bien documentés.
3. **Avancer étape par étape** : D'abord les fonctionnalités de base (authentification, gestion des offres), puis le matching et enfin le chat.
4. **Tester régulièrement** : Vérifier que chaque partie fonctionne avant d'enchaîner sur la suivante.

Pourquoi utiliser l'API Adzuna ?

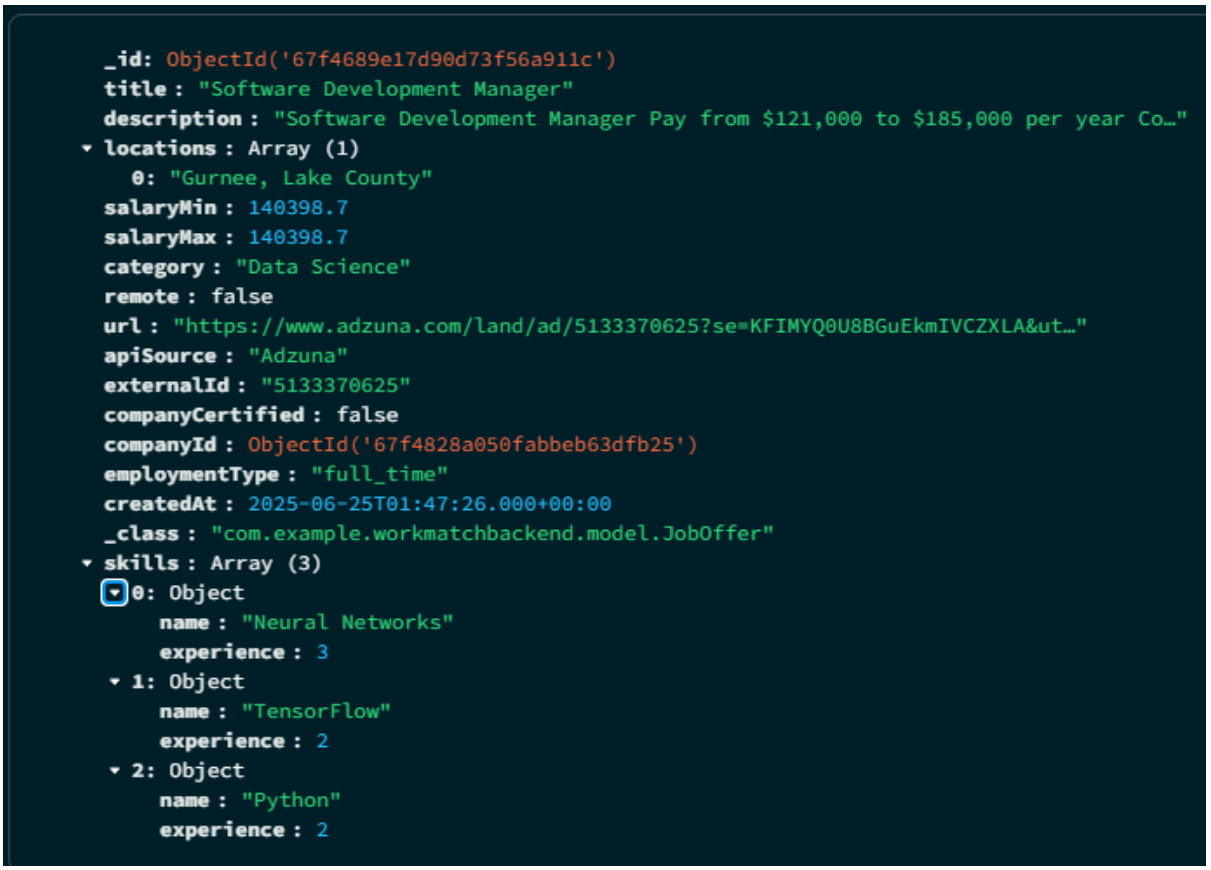
J'ai décidé d'intégrer l'API Adzuna pour récupérer des offres d'emploi réelles et enrichir mon application. Le but était de ne pas me limiter à des données fictives, mais d'offrir un contenu crédible et varié pour le système de matching.

Les raisons principales de ce choix :

- **Gain de temps** : Plutôt que d'inventer manuellement des centaines d'offres, l'API Adzuna m'a permis d'en importer automatiquement.
- **Données à jour** : Adzuna fournit des offres actualisées, ce qui donne plus de réalisme à l'application.
- **Filtrage possible** : J'ai pu cibler les offres du domaine informatique, mais aussi sélectionner les pays et les zones géographiques grâce aux paramètres flexibles de l'API.
- **Exigence du projet** : Le cahier des charges demandait d'intégrer une API externe pour travailler avec des données réelles, et Adzuna correspondait parfaitement aux besoins de WorkMatch.

L'intégration s'est faite via des appels HTTP depuis le backend. Ensuite, j'ai stocké les offres récupérées dans ma base MongoDB pour les réutiliser dans le processus de matching.

MongoDB offre importée exemple :



Extrait de code montrant l'appel à l'API Adzuna avec le filtrage :

```
String apiUrl = "https://api.adzuna.com/v1/api/jobs/" + country + "/search/" + page
        + "?app_id=b50d337a&app_key=7a9d8272a034e629a9f62ae0adb917ba"
        + "&what=" + keyword
        + "&results_per_page=50"
        + "&content-type=application/json";

try {
    ResponseEntity<JsonNode> response = restTemplate.getForEntity(apiUrl, JsonNode.class);
    JsonNode jobs = response.getBody().get("results");
```

```
if (!(title.contains("developer") || title.contains("engineer") || title.contains("software") || title.contains("it")
        || category.contains("tech") || category.contains("software") || category.contains("developer")
        || category.contains("data") || category.contains("it") || category.contains("engineering")
        || category.contains("cloud") || category.contains("cyber"))) {
    continue;
}
```

```
jobOfferRepository.save(jobOffer);
```

Pourquoi React pour le frontend ?

Pour développer l'interface utilisateur de WorkMatch, j'ai choisi d'utiliser **React Native Web**. Ce choix s'explique par plusieurs raisons pratiques et techniques.

Tout d'abord, **React** est une bibliothèque très populaire et bien documentée. Cela m'a permis de trouver facilement des ressources et des solutions lorsque j'avais des problèmes. De plus, React est basé sur un système de composants réutilisables, ce qui m'a aidé à structurer mon code de manière plus claire et organisée.

J'ai opté pour **React Native Web** afin de rendre l'application accessible directement depuis un navigateur. Ce framework permet de créer une interface mobile tout en la rendant compatible avec le web, ce qui correspondait parfaitement à mon objectif : développer rapidement une application responsive sans avoir à créer deux versions différentes.

Une autre raison importante est la facilité d'intégration avec mon backend en **Spring Boot**. Grâce à des librairies comme **Axios**, j'ai pu connecter mon frontend aux différentes API de manière simple et efficace.

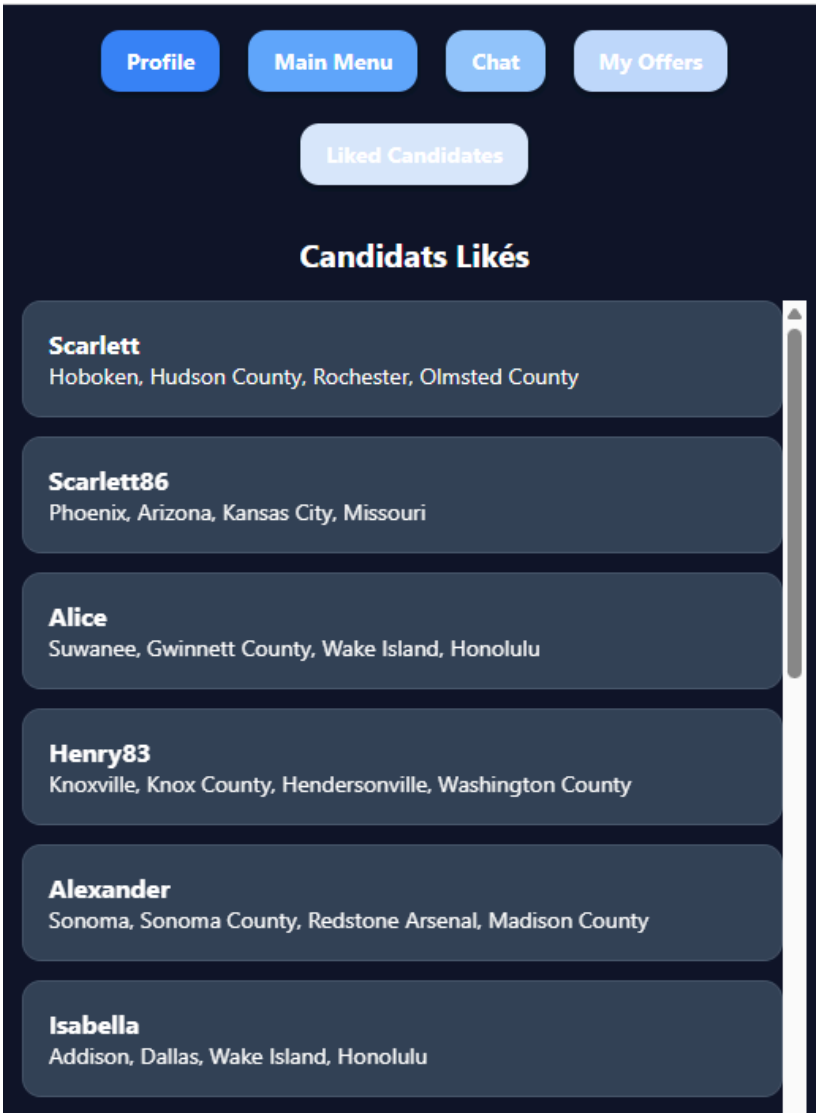
Enfin, React offre une bonne gestion de l'état avec des outils comme useState et useEffect, ce qui m'a permis de gérer dynamiquement les offres d'emploi, les swipes ou encore les notifications en temps réel.

Extrait de code récupération des offres avec Axios :

```
useEffect(() => {
    const fetchData = async () => {
      const type = await AsyncStorage.getItem('userType');
      setUserType(type);

      try {
        const token = await AsyncStorage.getItem("userToken");
        const companyId = await AsyncStorage.getItem("userId");

        const response = await axios.get(`${BASE_URL}/likes/liked-candidates/${companyId}`, {
          headers: { Authorization: `Bearer ${token}` }
        });

        console.log("Candidats likés :", response.data);
        setCandidates(response.data);
      } catch (error) {
        console.error("Erreur candidats likés :", error);
      } finally {
        setIsLoading(false);
      }
    };
fetchData();
}, []);
```

Screenshot du résultat :

Liked Candidates



2.8. Étude technique

Analyse du besoin

Le besoin principal était de créer une application qui facilite la mise en relation entre des chercheurs d'emploi et des entreprises via un système moderne de matching. L'idée n'était pas juste de lister des offres d'emploi, mais de proposer un fonctionnement inspiré des applications de swipe, pour rendre l'expérience plus interactive et rapide.

Les utilisateurs devaient pouvoir :

- Créer un compte (entreprise ou candidat).
- Pour les candidats : swiper des offres d'emploi correspondant à leurs préférences.
- Pour les entreprises : swiper des profils de candidats.
- Obtenir un **match** lorsque les deux parties montrent de l'intérêt.
- Discuter via un système de messagerie intégré après le match.

Les entreprises avaient aussi besoin d'un espace pour publier leurs propres offres d'emploi. Pour enrichir l'application, j'ai ajouté l'intégration de l'API Adzuna afin d'importer des offres réelles et d'avoir une base de données plus complète.

Schéma de l’algorithme de matching

Le système de matching fonctionne ainsi :

1. Un utilisateur ou une entreprise swipe vers la droite = un like est enregistré.
2. Lors d’un swipe, le backend vérifie si l’autre partie a déjà liké.
3. Si oui = création d’un **match** et ouverture d'une conversation.
4. Une notification est envoyée en temps réel grâce au WebSocket.

Les règles de matching prennent en compte :

- Le type d'utilisateur.
- L'ID des offres pour les candidats.
- L'ID des candidats pour les entreprises.

Extrait de code montrant la vérification d’un match suite à un swipe :

```
public void checkAndCreateMatch(String swiperId, String swipedId, String companyId) {
    boolean isMutualLike = false;
    String individualUserId = null;
    String companyUserId = null;

    if (companyId == null || companyId.isEmpty()) {
        List<Like> likesByJobSeeker = likeRepository.findAllBySwiperId(swipedId);
        isMutualLike = likesByJobSeeker.stream()
                .anyMatch(like -> swiperId.equals(like.getCompanyId()));

        if (isMutualLike) {
            individualUserId = swipedId;
            companyUserId = swiperId;
        }
    } else {
        Optional<Like> companyLike = likeRepository.findBySwiperIdAndSwipedId(companyId, swiperId);
        if (companyLike.isPresent()) {
            isMutualLike = true;
            individualUserId = swiperId;
            companyUserId = companyId;
        }
    }

    if (isMutualLike) {
        boolean matchExists = matchRepository.existsByIndividualUserIdAndCompanyUserId(individualUserId, companyUserId) ||
                matchRepository.existsByIndividualUserIdAndCompanyUserId(companyUserId, individualUserId);

        if (!matchExists) {
            Match match = new Match(individualUserId, companyUserId);
            matchRepository.save(match);
            System.out.println("Match créé entre " + individualUserId + " et " + companyUserId);
        }

        boolean conversationExists = conversationRepository.existsByUser1IdAndUser2Id(individualUserId, companyUserId) ||
                conversationRepository.existsByUser1IdAndUser2Id(companyUserId, individualUserId);
```

```
            conversationRepository.save(new Conversation(individualUserId, companyUserId));
            System.out.println("Conversation créée entre " + individualUserId + " et " + companyUserId);
        }

        Map<String, String> notifToIndividual = new HashMap<>();
        notifToIndividual.put("type", "match");
        notifToIndividual.put("message", "Nouveau match !");
        notifToIndividual.put("withUserId", companyUserId);
        notifToIndividual.put("conversationId", individualUserId + "_" + companyUserId);
        notifToIndividual.put("senderId", companyUserId); // L'expéditeur est la company

        Map<String, String> notifToCompany = new HashMap<>();
        notifToCompany.put("type", "match");
        notifToCompany.put("message", "Nouveau match !");
        notifToCompany.put("withUserId", individualUserId);
        notifToCompany.put("conversationId", individualUserId + "_" + companyUserId);
        notifToCompany.put("senderId", individualUserId); // L'expéditeur est l'individu

        messagingTemplate.convertAndSend("/topic/notifications/" + individualUserId, notifToIndividual);
        messagingTemplate.convertAndSend("/topic/notifications/" + companyUserId, notifToCompany);

    } else {
        System.out.println("Aucun like mutuel détecté entre " + swiperId + " et " + swipedId);
    }
}
```

Pour le détail du fonctionnement du système de matching et du tri des candidats, se référer à la section 3.2.3.

3. Réalisation (Développement & Implémentation)

3.1. Structure des Données

3.1.1. Organisation des fichiers – Backend (Spring Boot)

L'architecture du dossier workmatch-backend suit une structure claire, typique d'un projet Spring Boot en couches, favorisant la séparation des responsabilités (pattern MVC étendu).

Structure principale (/src/main/java/com/example/workmatchbackend)

Ce répertoire contient l'ensemble du code source métier de l'application, organisé comme suit :

config

Ce package contient tous les fichiers de configuration essentiels au bon fonctionnement du backend :

- SecurityConfig.java : configuration Spring Security + JWT.
- WebSocketConfig.java : activation et configuration du WebSocket.
- CorsConfig.java : gestion fine du CORS pour les autorisations frontend.
- AppConfig.java, RestTemplateConfig.java : configurations diverses liées aux beans ou aux appels externes (comme Adzuna via AdzunaConfig.java).

controller

Contient tous les contrôleurs REST qui exposent les routes API :

- Exemple : JobOfferController.java pour les routes d'offres d'emploi, UserController.java pour l'inscription et la connexion, ChatController.java et MessageController.java pour la messagerie.
- Chaque contrôleur appelle les services métier correspondants.

service

Contient la logique métier de chaque fonctionnalité :

- Exemple : JobOfferService.java, MatchService.java, MessageService.java, SwipeService.java...
- Chaque service est injecté dans un ou plusieurs contrôleurs et utilise les repository pour manipuler les données.

repository

Couches DAO utilisant Spring Data MongoDB :

- Exemple : UserRepository, JobOfferRepository, MessageRepository.
- Tous ces repositories étendent MongoRepository<> et permettent les opérations CRUD sans écrire de requêtes complexes.

model

Contient les entités (modèles de données) qui correspondent aux documents MongoDB :

- Exemple : User.java, JobOffer.java, JobSearcher.java, Conversation.java, Message.java, etc.
- Chaque classe est annotée avec @Document (MongoDB) et possède les champs requis, comme @Id.

filter

- JwtRequestFilter.java : filtre qui intercepte toutes les requêtes pour vérifier le token JWT et authentifier l'utilisateur.

util

- Contient des classes utilitaires comme JwtUtil.java (génération/validation de tokens).

Fichiers de configuration racine

- Dockerfile : pour le déploiement sur Render.
- application.properties : contient les configurations d'environnement (Mongo URI, port, etc.).
- pom.xml : gestion des dépendances via Maven.

resources

- Contient les fichiers de configuration applicative (comme application.properties).

3.1.2.Organisation des fichiers – Frontend (React)

Le frontend est structuré selon une logique modulaire claire avec une séparation par pages, composants et utilitaires.

Structure principale (/src/main/java/com/example/workmatchfrontend)

app

Cœur de l'application côté Expo Router :

- index.tsx : point d'entrée
- config/ : centralise les configurations globales (ex : navigation)
- navigations/MainNavigator.js : contient la logique de routing (navigation entre les pages)

pages

Organisé par sections fonctionnelles :

- auth/ : pages de connexion et d'inscription (SignInPage.tsx, SignUpPage.tsx)
- chat/ : interface de discussion temps réel (ChatRoom.js, ChatPage.js)
- homepages/ : vues principales des utilisateurs (CompanyHomePage.js, IndividualHomePage.js)
- liked/ : pages liées aux favoris (LikedOffersPage.js, LikedCandidateDetailsPage.js, etc.)
- offers/ : gestion des offres (CreateOfferPage.js, EditOfferPage.js, JobOfferDetails.js)
- onboarding/ : configuration initiale du profil (CompanyOnboardingPage.js, JobSeekerOnboardingPage.js)
- profile/ : gestion des préférences utilisateur
- settings/ : paramètres de l'application

components

Composants réutilisables :

- JobSwiper.js : logique de swiping pour les offres

- TabBarIcon.tsx, ThemedText.tsx : éléments visuels partagés

constants

- api.js : point central pour les appels Axios (API backend)
- Colors.ts : constantes de couleur pour un design cohérent

hooks

- Hooks personnalisés pour la gestion des thèmes (useThemeColor.ts, useColorScheme.ts)

assets

- Contient les images (react-logo.png, etc.) et les polices (fonts/)

Fichiers de configuration

- .env : contient les variables d'environnement (URL backend)
- app.config.js, babel.config.js, expo-router.config.js : fichiers nécessaires à l'environnement Expo
- package.json : dépendances NPM utilisées

3.1.3.Modélisation des Tables

Afin de gérer efficacement les données utilisateurs, offres d'emploi, interactions et messagerie instantanée, une base de données non relationnelle **MongoDB** a été choisie.
Elle a été sélectionnée pour sa souplesse, son adaptabilité aux données dynamiques, et ses performances élevées lors des requêtes rapides en temps réel.

Voici les structures principales des modèles utilisés :

Modèle : User (utilisateurs individuels et entreprises)

- **_id** : ObjectId (généré automatiquement par MongoDB)
- **username** : String, requis, unique
- **email** : String, requis, unique
- **password** : String, crypté avec l'algorithme BCrypt, requis
- **userType** : Enum (INDIVIDUAL, COMPANY), requis

- **preferredCategories** : Array de String, facultatif

Modèle : JobOffer (offres d'emploi publiées par les entreprises)

- **_id** : ObjectId (automatique)
- **title** : String, requis
- **description** : String, requis
- **salaryMin** et **salaryMax** : Number, requis
- **skills** : Array de String
- **category** : String (ex : "Software Developer")
- **companyId** : ObjectId (référence vers l'entreprise)
- **remote** : Boolean (télétravail ou non)
- **locations** : Array de String
- **employmentType** : Enum (full_time, part_time, internship, freelance)

Modèle : SwipedCard (enregistrement des swipes)

- **_id** : ObjectId
- **swiperId** : ObjectId (celui qui swiped)

- **swipedId** : ObjectId (celui qui est swipé)
- **isFromRedirection** : Boolean (si le swipe vient d'une redirection)
- **direction** : String ("right" ou "left")

```
_id: ObjectId('6807e78dec476d0e535f3411')
swiperId : "67f47effa78652b691b2f215"
swipedId : "67f4689e17d90d73f56a911c"
direction : "right"
jobOfferId : ""
isFromRedirection : false
_class : "com.example.workmatchbackend.model.SwipedCard"
```

Modèle : Match (confirmation de match)

- **_id** : ObjectId
- **individualUserId** : ObjectId du candidat
- **companyUserId** : ObjectId de l'entreprise

```
_id: ObjectId('680bc3ede13551291115058b')
individualUserId : "67f47effa78652b691b2f21a"
companyUserId : "67f4828a050fabbeb63dfb2d"
_class : "com.example.workmatchbackend.model.Match"
```

Modèles : Conversation et Message (messagerie instantanée)

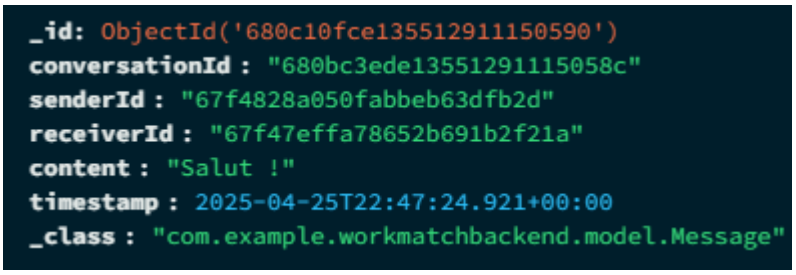
Conversation :

- **_id** : ObjectId
- **user1Id** : ObjectId
- **user2Id** : ObjectId

```
_id: ObjectId('680bc3ede13551291115058c')
user1Id : "67f47effa78652b691b2f21a"
user2Id : "67f4828a050fabbeb63dfb2d"
```

Message :

- **_id** : ObjectId
- **conversationId** : ObjectId de la conversation
- **senderId** : ObjectId de l'émetteur
- **receiverId** : ObjectId du récepteur
- **content** : texte du message
- **timestamp** : date d'envoi



3.2. Fonctionnalités Principales

3.2.1. Authentification des Utilisateurs

L'authentification est un élément central de WorkMatch pour sécuriser l'accès aux fonctionnalités. Le système repose sur l'utilisation de JWT (JSON Web Tokens) combiné à Spring Security. Lorsqu'un utilisateur, qu'il soit chercheur d'emploi ou entreprise, se connecte via la page de connexion, ses identifiants (email et mot de passe) sont transmis au backend par une requête POST. Le backend vérifie alors les informations reçues : l'email est recherché dans la base de données et le mot de passe est comparé après décryptage avec l'algorithme BCrypt.

```java
@PostMapping("/login")
public ResponseEntity<?> loginUser(@RequestBody Map<String, String> user) {
    String username = user.get("username");
    String password = user.get("password");
    if (username == null || username.trim().isEmpty()) {
        return ResponseEntity.badRequest().body("Le nom d'utilisateur est requis.");
    }

    if (password == null || password.trim().isEmpty()) {
        return ResponseEntity.badRequest().body("Le mot de passe est requis.");
    }

    if (username.length() > 30) {
        return ResponseEntity.badRequest().body("Le nom d'utilisateur est trop long.");
    }

    if (password.length() > 50) {
        return ResponseEntity.badRequest().body("Le mot de passe est trop long.");
    }

    User existingUser = userRepository.findByUsername(username);
    if (existingUser != null && passwordEncoder.matches(password, existingUser.getPassword())) {
        String token = jwtUtil.generateToken(username);
        Map<String, String> response = new HashMap<>();
        response.put("token", token);
        response.put("userType", existingUser.getUserType().toString());
        response.put("userId", existingUser.getId());
        return ResponseEntity.ok(response);
    } else {
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid credentials");
    }
}
```

Si les identifiants sont valides, un token JWT est généré côté serveur et renvoyé au frontend. Ce token est ensuite stocké localement dans AsyncStorage côté React Native Web. Grâce à ce stockage local, chaque future requête envoyée vers une route protégée inclut automatiquement le token dans l'en-tête Authorization, ce qui permet au serveur d'authentifier l'utilisateur sans devoir ressaisir ses identifiants.

Pour garantir la sécurité et la fiabilité des échanges entre le frontend et le backend, j'ai mis en place une architecture combinant Spring Security et JWT. La classe JwtUtil assure la gestion complète du token : elle génère le token signé avec l'algorithme HS256, extrait les informations (comme le username ou la date d'expiration) et valide la cohérence du token, en vérifiant par exemple son expiration et la correspondance avec l'utilisateur attendu.

```
public String generateToken(String username) {  no usages   unknown
    Map<String, Object> claims = new HashMap<>();
    return Jwts.builder()
            .setClaims(claims)
            .setSubject(username)
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10))
            .signWith(SignatureAlgorithm.HS256, secret)
            .compact();
}
```

Chaque requête entrante est interceptée par une classe spécifique, JwtRequestFilter. Ce filtre joue un rôle clé : il analyse systématiquement l'en-tête Authorization pour vérifier la présence d'un token "Bearer", extrait et valide le token via JwtUtil. Si le token est correct, l'utilisateur est authentifié dans le contexte de sécurité Spring. En cas de problème (token manquant, invalide ou expiré), une gestion d'exception simple empêche la propagation de l'erreur sans bloquer l'application.

```
if (authorizationHeader == null || !authorizationHeader.startsWith("Bearer ")) {
    System.out.println("JWT Token !");
    filterChain.doFilter(request, response);
    return;
}
```

La configuration globale de Spring Security, dans la classe SecurityConfig, renforce la robustesse de ce système. Les règles appliquées sont les suivantes : désactivation de la protection CSRF (puisque l'application utilise une API REST sans cookies), mise en place d'une politique de session stateless grâce à JWT, définition claire des routes publiques pour le login et l'inscription, et application prioritaire du JwtRequestFilter avant toute autre authentification standard.

```
http
        .csrf(csrf -> csrf.disable())
        .sessionManagement(sess -> sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(auth -> auth
                .requestMatchers("/auth/**", "/users/login", "/ws/**").permitAll()
                .anyRequest().authenticated()
        )
        .headers(headers -> headers.frameOptions(frameOptions -> frameOptions.disable()))
        .addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);
```

Ainsi, tout l'environnement WorkMatch bénéficie d'une authentification sécurisée, moderne et adaptée aux contraintes d'une application mobile et web.

3.2.2.Création et Gestion des Offres d’Emploi

La gestion des offres d'emploi est une fonctionnalité essentielle de WorkMatch.
Elle permet aux entreprises de publier, modifier et supprimer leurs annonces via des interfaces dédiées et sécurisées.

Création d’une offre :
Les entreprises peuvent créer une offre via une requête **POST** vers l’API.
L’offre doit contenir des champs obligatoires tels que :

- le titre,
- la description,
- la fourchette salariale,
- les compétences requises,
- la localisation,
- et le type de contrat.

```
await axios.post(`${BASE_URL}/joboffers`, newOffer, {
  headers: {
    Authorization: token,
    'Content-Type': 'application/json',
  },
});
```

Modification d’une offre :
Une fois publiée, l’entreprise peut modifier son offre via une requête **PUT**.
Des contrôles de validation sont effectués pour éviter les incohérences, par exemple vérifier que le salaire minimum est bien inférieur au salaire maximum.

Suppression d’une offre :
Les entreprises peuvent supprimer leurs annonces via une requête **DELETE**.
Lors de la suppression, un traitement est effectué pour effacer également toutes les interactions liées à cette offre (likes, swipes, matchs).

```java
@PostMapping
public ResponseEntity<JobOffer> createJobOffer(@RequestBody JobOffer jobOffer) {
    logger.info("Creating a new job offer.");
    if (jobOffer.getTitle() == null || jobOffer.getTitle().replaceAll("\\s", "").length() < 7) {
        return ResponseEntity.badRequest().body(null);
    }
    if (jobOffer.getDescription() == null || jobOffer.getDescription().replaceAll("\\s", "").length() < 20) {
        return ResponseEntity.badRequest().body(null);
    }
    if (jobOffer.getSalaryMin() >= jobOffer.getSalaryMax()) {
        return ResponseEntity.badRequest().body(null);
    }
    if (jobOffer.getEmploymentType() == null || jobOffer.getEmploymentType().trim().isEmpty()) {
        return ResponseEntity.badRequest().body(null);
    }
    if (jobOffer.getLocations() == null || jobOffer.getLocations().isEmpty()) {
        return ResponseEntity.badRequest().body(null);
    }
    if (jobOffer.getSkills() == null || jobOffer.getSkills().isEmpty()) {
        return ResponseEntity.badRequest().body(null);
    }
    if (jobOffer.getCreatedAt() == null) {
        jobOffer.setCreatedAt(LocalDate.now());
    }
    JobOffer savedJobOffer = jobOfferService.saveJobOffer(jobOffer);
    return ResponseEntity.status(HttpStatus.CREATED).body(savedJobOffer);
}
```

```java
@DeleteMapping("/{id}")  no usages  unknown *
public ResponseEntity<?> deleteJobOffer(@PathVariable String id) {
    logger.info("Deleting job offer and its dependencies for ID: {}", id);

    JobOffer offer = jobOfferRepository.findById(id).orElse(null);
    if (offer == null) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Offre introuvable");
    }

    ObjectId companyId = offer.getCompanyId();

    long count = jobOfferRepository.countByCompanyId(companyId);

    if (count <= 1) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Impossible de supprimer la dernière offre.");
    }

    jobOfferService.deleteJobOfferAndDependencies(id);
    return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
}
```

```java
@PutMapping("/{id}")
public ResponseEntity<?> updateJobOffer(@PathVariable String id, @RequestBody JobOffer updatedOffer) {
    Optional<JobOffer> existingOfferOpt = jobOfferService.getJobOfferById(id);

    if (existingOfferOpt.isEmpty()) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Offre non trouvée.");
    }
    String title = updatedOffer.getTitle() != null ? updatedOffer.getTitle().replaceAll("\\s+", "") : "";
    if (title.length() < 7) {
        return ResponseEntity.badRequest().body("Le titre doit contenir au moins 7 caractères (hors espaces).");
    }

    String description = updatedOffer.getDescription() != null ? updatedOffer.getDescription().replaceAll("\\s+", "") : "";
    if (description.length() < 20) {
        return ResponseEntity.badRequest().body("La description doit contenir au moins 20 caractères (hors espaces).");
    }

    if (updatedOffer.getSalaryMin() >= updatedOffer.getSalaryMax()) {
        return ResponseEntity.badRequest().body("Le salaire minimum doit être inférieur au salaire maximum.");
    }

    List<String> validTypes = List.of("full_time", "part_time", "internship", "freelance");
    if (updatedOffer.getEmploymentType() == null || !validTypes.contains(updatedOffer.getEmploymentType())) {
        return ResponseEntity.badRequest().body("Le type de contrat est invalide.");
    }

    if (updatedOffer.getLocations() == null || updatedOffer.getLocations().isEmpty()) {
        return ResponseEntity.badRequest().body("Veuillez sélectionner au moins une ville.");
    }

    if (updatedOffer.getSkills() == null || updatedOffer.getSkills().isEmpty()) {
        return ResponseEntity.badRequest().body("Veuillez sélectionner au moins une compétence.");
    }
```

Validation métier lors de la création d'une offre

Avant de sauvegarder une offre d'emploi, plusieurs règles de validation métier sont directement intégrées dans le contrôleur afin de garantir la cohérence et la qualité des données enregistrées.

Les principales validations effectuées sont :

- Vérification de la présence et de la longueur minimale du **titre** (au moins 7 caractères après suppression des espaces inutiles).
- Validation de la **description**, qui doit contenir au moins 20 caractères pertinents.
- Contrôle logique sur les **salaires** : le salaire minimum doit obligatoirement être inférieur au salaire maximum.
- Vérification que des champs essentiels comme le **type de contrat** (employmentType), les **localisations** (locations) et les **compétences requises** (skills) ne sont ni vides ni null.
- Ajout automatique de la **date de création** (createdAt) si elle n'est pas fournie dans la requête.

```
if (jobOffer.getTitle() == null || jobOffer.getTitle().replaceAll("\\s", "").length() < 7) {
    return ResponseEntity.badRequest().body(null);
}
if (jobOffer.getDescription() == null || jobOffer.getDescription().replaceAll("\\s", "").length() < 20) {
    return ResponseEntity.badRequest().body(null);
}
if (jobOffer.getSalaryMin() >= jobOffer.getSalaryMax()) {
    return ResponseEntity.badRequest().body(null);
}
if (jobOffer.getEmploymentType() == null || jobOffer.getEmploymentType().trim().isEmpty()) {
    return ResponseEntity.badRequest().body(null);
}
if (jobOffer.getLocations() == null || jobOffer.getLocations().isEmpty()) {
    return ResponseEntity.badRequest().body(null);
}
if (jobOffer.getSkills() == null || jobOffer.getSkills().isEmpty()) {
    return ResponseEntity.badRequest().body(null);
}
if (jobOffer.getCreatedAt() == null) {
    jobOffer.setCreatedAt(LocalDate.now());
}
```

Intégration Frontend (React Native Web)

Côté frontend, un formulaire dynamique a été développé pour permettre aux entreprises de saisir facilement toutes les informations nécessaires lors de la création d'une offre. Des composants interactifs comme des dropdowns sont utilisés pour sélectionner les localisations et les compétences. De plus, des boutons d'ajustement permettent de définir rapidement la fourchette salariale minimum et maximum. Ce système rend la saisie fluide, intuitive et limite les erreurs côté utilisateur.

Suppression en cascade des données liées à une offre

Lorsqu'une entreprise décide de supprimer une offre d'emploi, il ne s'agit pas uniquement d'effacer le document JobOffer de la base de données. Cette offre est souvent liée à de nombreuses interactions telles que des swipes effectués par des candidats, des likes

enregistrés, des matches créés à partir d'intérêts réciproques, ainsi que des conversations et des messages associés à ces matches.

Afin d’assurer l’intégrité et la cohérence des données, un mécanisme de suppression en cascade a été mis en place. Concrètement, lors de la suppression d'une offre, tous les swipes où l’ID de l’offre apparaît sont automatiquement supprimés, ainsi que tous les likes associés à cette offre. Les matches qui concernent exclusivement cette offre sont également supprimés. Enfin, si des conversations avaient été ouvertes suite à ces matches, elles sont elles aussi effacées, de même que tous les messages correspondants.

Cette approche rigoureuse garantit que l’ensemble des données associées à une offre est proprement supprimé, évitant ainsi la persistance de données orphelines ou incohérentes dans la base.

Avantages de cette approche

L'effacement en cascade présente plusieurs bénéfices majeurs. D'une part, il assure la cohérence des données : aucune référence à une offre supprimée ne subsiste. D'autre part, il offre une expérience utilisateur optimale en évitant aux candidats et aux entreprises de tomber sur des éléments devenus inexistants. Enfin, il contribue à maintenir une base de données propre et performante, en réduisant le volume de données inutiles stockées.

3.2.3.Algorithme de Matching

L'algorithme de matching est au cœur de l'application WorkMatch. Il permet d'établir un lien entre un candidat et une entreprise lorsque les deux manifestent un intérêt réciproque via le système de swiping.

Du côté des candidats, le swipe vers la droite signifie un "like" sur une offre d’emploi. Du côté des entreprises, le swipe peut être effectué sur des profils de chercheurs d’emploi à partir de deux pages distinctes : CompanyHomePage, qui propose un balayage général de profils, et CompanyRedirectedPage, qui cible spécifiquement les candidats potentiels pour une offre donnée.

Chaque interaction est stockée dans la collection SwipedCard, qui enregistre l’ID du swipant, l’ID de la cible, la direction du swipe (droite ou gauche) et l’origine (redirigée ou non).

À chaque nouveau like, le backend vérifie s’il existe déjà un swipe réciproque enregistré dans la base de données. En cas de correspondance, un document Match est automatiquement généré. Cela déclenche également la création d’une Conversation entre les deux parties, et une notification est envoyée en temps réel via WebSocket à l’entreprise et au candidat concernés.

Extrait de code backend : détection d’un match côté entreprise

Le code ci-dessous montre comment le backend vérifie si un candidat a déjà liké une offre appartenant à une entreprise. Si c’est le cas, un Match est créé, une Conversation est générée, et une notification est envoyée à chaque partie via WebSocket (/topic/notifications/{userId}).

```
public boolean checkAndCreateMatchAfterCompanyLike(String companyUserId, String candidateUserId) {
    List<Like> candidateLikes = likeRepository.findAllBySwiperId(candidateUserId);

    boolean hasLikedCompany = candidateLikes.stream()
            .anyMatch(like -> like.getCompanyId().equals(companyUserId));

    if (hasLikedCompany) {
        boolean matchExists = matchRepository.existsByIndividualUserIdAndCompanyUserId(candidateUserId, companyUserId);
        if (!matchExists) {
            Match match = new Match(candidateUserId, companyUserId);
            matchRepository.save(match);
            System.out.println("Nouveau match créé entre " + candidateUserId + " et " + companyUserId);

            boolean convExists = conversationRepository.existsByUser1IdAndUser2Id(candidateUserId, companyUserId)
                    || conversationRepository.existsByUser1IdAndUser2Id(companyUserId, candidateUserId);

            if (!convExists) {
                conversationRepository.save(new Conversation(candidateUserId, companyUserId));
                System.out.println("Conversation créée !");
            }

            return true;
        }
    }

    System.out.println("Pas encore de like du candidat vers l'entreprise.");
    return false;
}
```

```
messagingTemplate.convertAndSend("/topic/notifications/" + individualUserId, notifToIndividual);
messagingTemplate.convertAndSend("/topic/notifications/" + companyUserId, notifToCompany);
```


```
messagingTemplate.convertAndSend("/topic/notifications/" + individualUserId, notifToIndividual);
messagingTemplate.convertAndSend("/topic/notifications/" + companyUserId, notifToCompany);
```


La gestion des offres d'emploi est une fonctionnalité essentielle de WorkMatch. Elle permet aux entreprises de publier, modifier et supprimer leurs annonces via des interfaces dédiées et sécurisées.

Lorsqu'une entreprise crée une offre, un formulaire dynamique en React Native Web permet de saisir le titre, la description, la fourchette salariale, les compétences (via des dropdowns), les localisations et le type de contrat.

Une fois l’offre remplie, une requête HTTP POST est envoyée vers l’API backend sécurisée via un token JWT (voir extrait de code ci-dessous).

Avant la sauvegarde, des règles métier sont appliquées pour garantir la cohérence des données :

Le titre doit faire minimum 7 caractères, la description au moins 20, le salaire min doit être inférieur au max, et les champs clés (type de contrat, villes, compétences) doivent être remplis.

Les entreprises peuvent modifier une offre existante avec une requête PUT, les mêmes règles de validation sont appliquées pour éviter les incohérences.

Enfin, la suppression d'une offre s'accompagne d'un mécanisme de suppression en cascade. Toutes les données liées (likes, swipes, matchs, conversations, messages) sont supprimées automatiquement pour garantir l'intégrité globale.

Fonctionnalités Principales du Système de Matching

Fonctionnalité	Description
Matching pour une offre spécifique	Affiche les candidats compatibles avec une offre précise (depuis la page MyOffers).
Matching global pour une entreprise	Affiche tous les candidats triés par compatibilité moyenne avec toutes les offres de l'entreprise.
Matching pour un candidat	Affiche les offres triés par compatibilité moyenne avec toutes les offres de l'entreprise
Filtrage	Supprime les candidats déjà swipés (droite ou gauche).
Tri intelligent	Affiche en premier les candidats ayant liké une offre de l'entreprise, puis ceux avec le meilleur score.
Affichage formaté	Le score de matching est arrondi à 2 décimales dans l'interface utilisateur.

Critères du Matching et Pondérations

Le score de compatibilité est calculé selon 4 critères :

Critère	Pondération	Description
Compétences	40%	Le candidat possède les compétences requises par l'offre.
Localisation	20%	Il est situé dans la ville de l'offre (ou accepte cette ville).
Télétravail	10%	Le candidat accepte le remote si l'offre le propose.
Salaire	30%	L'offre correspond au salaire minimum du candidat.

Chaque critère est normalisé sur 100, puis multiplié par sa pondération.

```java
private double calculateMatchingScore(JobSearcher jobSearcher, JobOffer jobOffer) {
    double score = 0.0;

    //Score basé sur les compétences (40%)
    int totalSkills = jobOffer.getSkills().size();
    double skillsScore = totalSkills > 0
            ? (jobSearcher.getSkills().stream()
            .filter(skill -> jobOffer.getSkills().stream()
                    .anyMatch(reqSkill -> reqSkill.getName().equalsIgnoreCase(skill.getName()) &&
                            skill.getExperience() >= reqSkill.getExperience()))
            .count() / (double) totalSkills)
            : 0.0;
    score += skillsScore * 40;

    //Score basé sur la localisation (20%)
    double locationScore = jobSearcher.getLocations().stream()
            .anyMatch(loc -> jobOffer.getLocations().contains(loc)) ? 1.0 : 0.0;
    score += locationScore * 20;

    //Score basé sur remote (10%)
    double remoteScore = (jobSearcher.isRemote() == jobOffer.isRemote()) ? 1.0 : 0.0;
    score += remoteScore * 10;

    //Score basé sur le salaire (30%)
    double salaryScore = 1.0 - (Math.abs(jobSearcher.getSalaryMin() - jobOffer.getSalaryMin()) / 5000.0);
    salaryScore = Math.max(0, Math.min(1, salaryScore)); // Entre 0 et 1
    score += salaryScore * 30;

    return score;
}
```

Matching Global pour une Entreprise (CompanyHomePage)

Lorsqu'une entreprise arrive sur sa page d'accueil (CompanyHomePage), elle voit une liste de tous les candidats classés par score **moyen** sur **l'ensemble de ses offres**.

Étapes du process :

1. Récupération de toutes les offres de l'entreprise via jobOfferRepository.findByCompanyId.
2. Récupération de tous les jobseekers.
3. Pour chaque jobseeker :
    a. Calcul du score pour **chaque** offre.
    b. Moyenne des scores : matchingScore.
4. Tri des candidats :
    a. En premier : ceux qui ont déjà liké une offre.
    b. Ensuite : ceux avec le meilleur score.

```
for (JobSearcher js : jobSearchers) {
    if (swipedIds.contains(js.getId())) {
        continue;
    }

    double totalScore = 0.0;
    for (JobOffer offer : companyOffers) {
        totalScore += calculateMatchingScore(js, offer);
    }
    double averageScore = totalScore / companyOffers.size();
    js.setMatchingScore(averageScore);
}
```

Matching pour une Offre Spécifique

Quand une entreprise consulte une **offre précise**, le système calcule un score unique pour cette offre uniquement.

Code backend :

```
public List<JobSearcher> findMatchingCandidatesForSingleOffer(String jobOfferId) {  no usages  unknown *
    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());

    List<JobSearcher> matchingCandidates =
            jobSearcherRepository.findAll()
                    .stream()
                    .filter(js -> js.getSkills() != null)
                    .map(js -> {
                        double score = calculateMatchingScore(js, jobOffer);
                        js.setMatchingScore(score);

                        return js;
                    })
                    .sorted(Comparator.comparing(JobSearcher::getMatchingScore).reversed())
                    .collect(Collectors.toList());

    System.out.println("Nombre de candidats trouvés pour l'offre : " + matchingCandidates.size());

    return matchingCandidates;
}
```

Filtrage des Candidats déjà Swipés

Pour ne pas proposer plusieurs fois le même candidat à une entreprise, on filtre tous ceux qui ont déjà été likés ou rejetés.

```
List<JobSearcher> jobSearchers = jobSearcherRepository.findAll();

//Récupérer les swipes pour exclure les candidats déjà vus
List<Like> swipedCandidates = likeRepository.findBySwiperId(companyId);
Set<String> swipedIds = swipedCandidates.stream()
        .map(Like::getSwipedId)
        .collect(Collectors.toSet());

for (JobSearcher js : jobSearchers) {
    if (swipedIds.contains(js.getId())) {
        continue;
    }

    double totalScore = 0.0;
    for (JobOffer offer : companyOffers) {
        totalScore += calculateMatchingScore(js, offer);
    }
    double averageScore = totalScore / companyOffers.size();
    js.setMatchingScore(averageScore);
}

return jobSearchers.stream()
        .filter(js -> !swipedIds.contains(js.getId()))
        .sorted(Comparator.comparing(JobSearcher::getMatchingScore).reversed())
        .collect(Collectors.toList());
```

Tri Final des Résultats

Après filtrage, les résultats sont triés de façon intelligente :

1. En priorité : candidats ayant liké une offre de l'entreprise.
2. Ensuite : ceux ayant le meilleur score.

Affichage dans le Frontend (React Native)

Dans la CompanyHomePage, les candidats sont affichés avec leur score (arrondi à 2 décimales).

```
{selectedOffer ? (
    <Text style={styles.cardText}>Score de matching (offre sélectionnée) :
        {jobSearcher.matchingScore !== undefined ? jobSearcher.matchingScore.toFixed(2) + "%" : "N/A"}
    </Text>
) : (
    <Text style={styles.cardText}>Score de matching (toutes offres) :
        {jobSearcher.matchingScore !== undefined ? jobSearcher.matchingScore.toFixed(2) + "%" : "N/A"}
    </Text>
)}
```

3.2.4.Système de Messagerie

Une fois qu'un match est confirmé entre un candidat et une entreprise, WorkMatch ouvre automatiquement un canal de communication via un système de messagerie instantanée intégré.

La technologie utilisée repose sur **WebSocket**, combinée à **STOMP** et **SockJS**, pour garantir des échanges en temps réel. Le backend est configuré avec **Spring Boot**, tandis que le frontend, développé avec **React Native Web**, permet l'abonnement dynamique aux topics WebSocket afin de recevoir les messages en direct.

La gestion du projet s'est appuyée sur **Git** et **Bitbucket**, assurant un bon suivi de version et une intégration continue vers les environnements de déploiement.

Fonctionnement général du système :

Lorsqu'un match est validé, une **Conversation** est automatiquement générée côté backend. Les deux utilisateurs concernés peuvent ensuite échanger des messages à travers cette conversation. Chaque message est sauvegardé dans la collection **Message** de **MongoDB**. En parallèle, une **notification en temps réel** est envoyée via WebSocket à l'utilisateur destinataire, assurant une réception instantanée du message.

Configuration backend du WebSocket

Le backend définit un endpoint principal pour la connexion WebSocket ainsi qu'un **broker simple** pour la distribution des messages à chaque abonné.

Extrait de la configuration WebSocket (WebSocketConfig.java)
Le snippet ci-dessous montre la définition de l'endpoint /ws et du broker "/topic" utilisé pour les échanges temps réel :

```java
@Configuration  no usages  unknown
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Autowired  no usages
    private JwtUtil jwtUtil;

    @Override  no usages  unknown
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/ws")
                .setAllowedOriginPatterns("*")
                .withSockJS();

    }

    @Override  no usages  unknown
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/topic");
        registry.setApplicationDestinationPrefixes("/app");
    }
}
```

Abonnement aux notifications côté frontend (React Native Web)
Lorsque l'utilisateur est connecté, son identifiant est utilisé pour s'abonner au topic WebSocket personnalisé. Cela permet d'afficher en temps réel les notifications de nouveaux messages non lus.

```javascript
stomp.subscribe(`/topic/messages/${userId}`, (message) => {
  const msg = JSON.parse(message.body);
  console.log('Nouveau message reçu (notification) :', msg);
  setUnreadCount((prev) => prev + 1);
});
```

Stockage des messages
Chaque message envoyé entre un candidat et une entreprise est persisté dans la base de données MongoDB. Le document contient plusieurs champs essentiels : l'identifiant de la conversation (conversationId), les identifiants de l'émetteur et du destinataire (senderId, receiverId), le contenu du message (content) ainsi que la date et l'heure d'envoi (timestamp). Ce format structuré permet de retrouver l'historique des échanges à tout moment.

```
_id: ObjectId('680c10fce135512911150590')
conversationId : "680bc3ede13551291115058c"
senderId : "67f4828a050fabbeb63dfb2d"
receiverId : "67f47effa78652b691b2f21a"
content : "Salut !"
timestamp : 2025-04-25T22:47:24.921+00:00
_class : "com.example.workmatchbackend.model.Message"
```


Envoi et réception des messages

Dans le contrôleur ChatController.java, chaque message envoyé par un utilisateur est intercepté via l'endpoint /app/send/{conversationId}. Le message est ensuite diffusé en temps réel à tous les utilisateurs abonnés au topic associé à cette conversation.

Extrait du contrôleur ChatController (enregistrement du message)
Ce code illustre comment un message est reçu via une requête POST classique (endpoint /send), validé, puis sauvegardé dans MongoDB :

```
@PostMapping("/send")
public ResponseEntity<Message> sendMessage(@RequestBody Message messageDetails) {
    System.out.println("Message reçu : " + messageDetails);

    if (messageDetails.getSenderId() == null || messageDetails.getReceiverId() == null) {
        System.out.println("Erreur : senderId ou receiverId est null");
        return ResponseEntity.badRequest().build();
    }

    Message message = new Message(
            messageDetails.getConversationId(),
            messageDetails.getSenderId(),
            messageDetails.getReceiverId(),
            messageDetails.getContent(),
            Instant.now()
    );

    Message savedMessage = messageRepository.save(message);
    System.out.println("Message sauvegardé : " + savedMessage);
    return ResponseEntity.ok(savedMessage);
}
```

Déploiement et Hébergement

Le projet a été versionné via Git et hébergé sur Bitbucket, permettant un suivi efficace des modifications et une intégration simplifiée avec les plateformes de déploiement Render et Netlify. Afin de rendre l'application WorkMatch accessible, le backend et le frontend ont été déployés sur des plateformes cloud adaptées aux projets web. Le backend repose principalement sur les starters Spring Boot pour le web et MongoDB. Les dépendances essentielles ont été gérées via Maven, garantissant une intégration rapide des outils nécessaires au développement de l'API REST.

3.2.5.Backend (Spring Boot) - Render

Du côté backend, après avoir structuré le projet Spring Boot avec les répertoires classiques controller, service, repository et model, j'ai commencé par définir l'entité **User** avec les champs nécessaires à l'authentification. La création de l'interface **UserRepository**, étendant MongoRepository, s'est déroulée sans encombre, mais lors de l'implémentation du contrôleur utilisateur (**UserController**), j'ai rencontré plusieurs erreurs liées aux imports manquants et à la gestion des méthodes CRUD.

Par exemple, la méthode registerUser posait problème lors de la sauvegarde d'un nouvel utilisateur, en raison d'une mauvaise injection du repository. Après analyse, j'ai corrigé la déclaration des dépendances en utilisant correctement l'annotation @Autowired, ce qui a permis de résoudre l'erreur.

Chaque correction a été suivie de tests via **Postman** afin de valider les endpoints créés. Ce processus m'a permis d'affiner la logique métier et d'assurer la fiabilité des opérations CRUD avant de passer à l'intégration des fonctionnalités plus complexes comme l'authentification sécurisée et la gestion des tokens JWT.

Le backend développé avec **Spring Boot** a été déployé sur la plateforme **Render**, choisie pour sa simplicité d'utilisation et sa compatibilité avec les projets Java.

Avant le déploiement sur Render et Netlify, l'application a été entièrement testée en environnement local, en utilisant IntelliJ IDEA pour le backend, MongoDB Community Edition, ainsi qu'un émulateur Android pour simuler le comportement mobile.

Étapes de déploiement du backend sur Render :

- **Création du JAR exécutable :**
  mvn clean package
- **Configuration du Dockerfile :**
  Mentionne brièvement que le Dockerfile a été optimisé pour Render (mémoire, exécution).

```
#Build de l'application avec Maven
FROM maven:3.9.4-eclipse-temurin-17 AS build
WORKDIR /app
COPY . .
RUN mvn clean install -DskipTests

#Exécution avec JDK
FROM eclipse-temurin:17-jdk
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar

# Port exposé (peut rester 8080)
EXPOSE 8080

# Commande pour lancer l'application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

- **Connexion à Render :**
  Lien GitHub, config des variables (ex: MONGO_URI).
- **Gestion des ports :**
  Port 8080 exposé automatiquement.
- **Statelessness assurée :**
  Authentification gérée par JWT.

3.2.6.Frontend (React Native Web) - Netlify

Lors de la mise en place du frontend avec React Native Web, l'une des premières étapes a consisté à créer les pages de connexion et d'inscription. J'ai donc développé les composants SignInPage.tsx et SignUpPage.tsx, en utilisant la bibliothèque react-navigation afin de gérer la navigation entre ces différentes pages. Très rapidement, je me suis retrouvé confronté à une erreur liée à un conteneur imbriqué, provoquée par une mauvaise organisation du NavigationContainer et du MainNavigator. Après plusieurs essais, j'ai compris que la structure de mes imports et l'emplacement des balises de navigation n'étaient pas correctement définis. J'ai donc réorganisé l'arborescence de mes fichiers et ajusté les imports dans app.tsx et index.tsx, ce qui a permis de résoudre le problème.

Une autre difficulté rencontrée concernait l'intégration de la bibliothèque Axios. En essayant d'effectuer mes premières requêtes HTTP vers le backend, une erreur m'indiquait que le module Axios était introuvable. Après vérification, j'ai constaté que la dépendance n'avait

pas été correctement installée. Une simple réinstallation via la commande npm install axios a permis de régler le problème et de poursuivre le développement.

Des phases régulières de tests et de débogage ont été nécessaires pour valider l'affichage des composants et la bonne gestion de la navigation entre les différentes pages. Ces étapes ont permis d'assurer une interface fluide et fonctionnelle avant d'intégrer les échanges avec le backend.

L'intégration d'Axios a nécessité une réinstallation suite à une erreur de dépendance manquante. L'architecture globale du projet frontend a également été corrigée pour résoudre un bug lié à l'imbrication de composants de navigation.

Enfin, pour Git, quelques ajustements ont été faits lors de l'initialisation des dépôts (sans entrer dans les commandes précises), ce qui a facilité le suivi des évolutions du projet grâce à une gestion de version claire.

Le développement du frontend a débuté par la configuration de la structure de base en utilisant React Native Web. La navigation entre les différentes pages a été assurée grâce à la bibliothèque react-navigation combinée avec createStackNavigator.

Le fichier principal index.tsx a été conçu pour initialiser l'application et rediriger vers le MainNavigator, responsable de la gestion des routes. Le MainNavigator permet de gérer la navigation entre les pages de connexion (SignInPage) et d'inscription (SignUpPage).

```tsx
import React from 'react';
import 'react-native-gesture-handler';
import MainNavigator from "../navigations/MainNavigator"


export default function Main() {
  return <MainNavigator />;
}
```

```
function MainStackNavigator() {
  const [initialRoute, setInitialRoute] = React.useState('SignIn');

  React.useEffect(() => {
    const determineInitialRoute = async () => {
      const userType = await AsyncStorage.getItem('userType');
      const isLoggedIn = await AsyncStorage.getItem('userToken');

      if (isLoggedIn) {
        if (userType === 'INDIVIDUAL') {
          setInitialRoute('IndividualHome');
        } else if (userType === 'COMPANY') {
          setInitialRoute('CompanyHome');
        }
      } else {
        setInitialRoute('SignIn');
      }
    };

    determineInitialRoute();
  }, []);
```

Durant cette phase, des erreurs liées à une mauvaise organisation du NavigationContainer ont été rencontrées, provoquant des conflits de conteneurs imbriqués. La correction a consisté à repositionner correctement les composants de navigation et à ajuster les imports dans les bons fichiers (app.tsx, index.tsx).

Ensuite, l'intégration de la bibliothèque Axios a permis de connecter le frontend avec l'API backend. Des tests réguliers ont été réalisés afin de s'assurer du bon fonctionnement des requêtes HTTP. Pour cela, des outils comme **Postman** (test des endpoints REST) et **MongoDB Compass** (vérification visuelle des documents) ont été utilisés.

Cette structure de base du frontend, combinée aux outils de test, a constitué une fondation solide pour le développement des fonctionnalités plus complexes de l'application WorkMatch.

Le frontend, développé avec React Native Web, a été déployé sur **Netlify**, une solution simple et efficace pour héberger des applications web statiques.

Durant le développement, des conflits de versions entre certaines dépendances React Native ont été rencontrés, notamment avec npm. Pour contourner ces problèmes, la commande npm

install --legacy-peer-deps a été utilisée temporairement. Finalement, un passage à yarn a permis de stabiliser la gestion des packages et d'éviter ces incompatibilités récurrentes.

Définition de l'URL du backend Render dans les variables frontend pour les appels API :

```
# Pour distant
EXPO_PUBLIC_BACKEND_URL=https://projet-workmatch.onrender.com
```

3.3. Problèmes Rencontrés et Solutions

Analyse des Problèmes Techniques et Solutions Apportées

Durant la phase d'intégration et de validation, plusieurs problèmes techniques ont été identifiés, nécessitant des ajustements ciblés tant sur le backend que sur le frontend :

Problème 1 :
Les utilisateurs de type COMPANY avaient accès à l'ensemble des offres d'emploi, ce qui allait à l'encontre des règles métier définies.
Analyse : Ce problème était dû à l'absence de contrôle du userType dans les endpoints.
Solution : Un contrôle systématique a été ajouté, renvoyant une erreur 403 Forbidden en cas d'accès non autorisé.

Problème 2 :
Des erreurs récurrentes survenaient dans les appels API du frontend, principalement liées à l'oubli d'inclure le token JWT dans les en-têtes HTTP.
Solution : Mise en place d'un intercepteur global Axios pour garantir l'ajout automatique du token à chaque requête.

Problème 3 :
Le backend rencontrait des erreurs de compilation dues à des imports manquants et à des incohérences dans les signatures de méthodes.
Solution : Une revue complète des contrôleurs et services a été effectuée, accompagnée d'une vérification des dépendances Maven et du nettoyage des avertissements dans l'IDE.

3.3.1.Gestion des JWT

Lorsqu'un token JWT est invalide ou expiré, le backend retourne une réponse HTTP 403 Forbidden.
Pour gérer cela côté frontend (React Native Web), un intercepteur Axios a été mis en place. Il permet de détecter automatiquement ces erreurs et de rediriger l'utilisateur vers la page de connexion, tout en supprimant le token stocké localement dans AsyncStorage.

```
axios.interceptors.request.use(
  async config => {
    const token = await AsyncStorage.getItem('userToken');
    if (token) {
      config.headers.Authorization = `Bearer ${token}`;
    }
    return config;
  },
  error => Promise.reject(error)
);
```

Pendant l'implémentation de l'authentification, plusieurs problèmes ont été rencontrés liés à la gestion des tokens JWT, notamment :

Problèmes rencontrés :

- Erreurs fréquentes "JWT malformed" ou réponses 403 Forbidden lors des requêtes authentifiées.
- Mauvaise gestion de l'en-tête Authorization, avec ajout incorrect du préfixe "Bearer".
- Expiration du token non prise en compte côté frontend.

Solutions mises en place :

- Mise en place d'un intercepteur Axios côté frontend pour automatiser l'ajout du token dans l'en-tête Authorization.
- Vérification conditionnelle pour éviter le double préfixe "Bearer" :

```
const token = await AsyncStorage.getItem('userToken');
const bearerToken = token.startsWith('Bearer ') ? token : `Bearer ${token}`;
```

- Utilisation de l'outil jwt.io pour analyser et valider les tokens générés.
- Ajout de logs de contrôle temporaire pour vérifier la transmission correcte des tokens, puis suppression de tous les logs sensibles avant passage en production afin de renforcer la confidentialité des données.

L'ensemble des routes du backend a été testé manuellement à l'aide de **Postman**. Ces tests ont permis de vérifier :

- La bonne gestion de l'authentification via JWT (envoi du token dans l'en-tête).
- La création, modification et suppression des utilisateurs et des offres.
- La gestion correcte des erreurs (codes HTTP 400, 403, 404, 201…).
- Le respect des règles métier lors de la création d'une offre (validation des champs obligatoires, cohérence des salaires, etc.).

- La suppression en cascade des données liées à une offre ou à un utilisateur.

Tests WebSocket :

Des tests spécifiques ont été réalisés pour valider :

- La connexion au serveur WebSocket.
- La réception en temps réel des notifications de match et des messages.
- La stabilité des échanges, y compris la reconnexion automatique en cas de déconnexion.

3.3.2.Problèmes avec ObjectId de MongoDB

Lors du développement, des erreurs sont apparues à cause de la mauvaise gestion du type ObjectId dans les entités MongoDB.
Certaines routes échouaient ou retournaient null, car des chaînes de caractères étaient utilisées au lieu de véritables objets ObjectId.
Pour corriger cela, tous les champs liés aux identifiants (userId, companyId, jobOfferId, etc.) ont été explicitement typés en ObjectId, à la fois dans les entités Java et lors des traitements backend. Cette correction a permis de fiabiliser les relations entre les entités (User, JobOffer, Match, etc.), et d'éviter les erreurs de conversion à l'exécution.

Vérification des interfaces :

Chaque fonctionnalité a été testée manuellement sur la version web de l'application afin de s'assurer :

- Du bon affichage des composants (swiper, formulaires, listes de conversations).
- De la réactivité de l'interface et du respect des règles UX.
- De la gestion des erreurs côté utilisateur (ex : affichage de messages en cas de champ manquant dans un formulaire).

Tests des interactions avec le backend :

- Vérification que toutes les requêtes Axios intègrent correctement le token JWT.
- Tests des différentes actions utilisateurs : swipes, likes, envoi de messages, création d'offres.
- Contrôle du bon fonctionnement des notifications en temps réel (pastilles rouges, nouveaux messages, alertes de match).

3.3.3.Erreurs CORS

Pour permettre la communication entre le frontend déployé (Netlify) et le backend Spring Boot, une configuration spécifique CORS a été ajoutée.
La classe CorsConfig autorise uniquement les origines de confiance (localhost:8081 et https://workmatchtfe.netlify.app), limitant ainsi les risques d'attaques Cross-Origin.
Grâce à cette configuration, les erreurs de type CORS policy ont été éliminées et les échanges frontend-backend fonctionnent de manière sécurisée.

```
registry.addMapping("/**")
        .allowedOriginPatterns("http://localhost:8081", "https://workmatchtfe.netlify.app")
        .allowedMethods("*")
        .allowedHeaders("*")
        .allowCredentials(true);
```

Afin de résoudre les erreurs CORS survenues lors des tests locaux et en production (après déploiement sur Netlify), une configuration spécifique a été ajoutée dans la classe CorsConfig. Celle-ci définit précisément les origines autorisées et les autorisations de méthodes, tout en respectant les normes de sécurité imposées par l'utilisation du paramètre allowCredentials(true).
Cette configuration autorise uniquement les origines nécessaires, évitant les failles potentielles et assurant une communication fluide entre le frontend hébergé sur Netlify et le backend déployé sur Render. Grâce à cela, toutes les requêtes API sont désormais acceptées sans blocage CORS, que ce soit en environnement local ou en production.

3.4. Tests et Validation Technique

Validation de l'Authentification et des Endpoints Sécurisés

L'authentification basée sur Spring Security et les JWT a été rigoureusement testée en environnement local afin de garantir le bon fonctionnement des mécanismes de sécurité attendus.

Résultats des Tests avec Postman

Les scénarios suivants ont été réalisés pour valider la gestion des accès aux endpoints publics et privés :

Test	Résultat attendu	Résultat obtenu
Accès à un endpoint public sans token	200 OK	Conforme
Accès à un endpoint privé sans token	403 Forbidden	Conforme
Accès à un endpoint privé avec un token invalide	403 Forbidden	Conforme
Accès à un endpoint privé avec un token valide	200 OK	Conforme

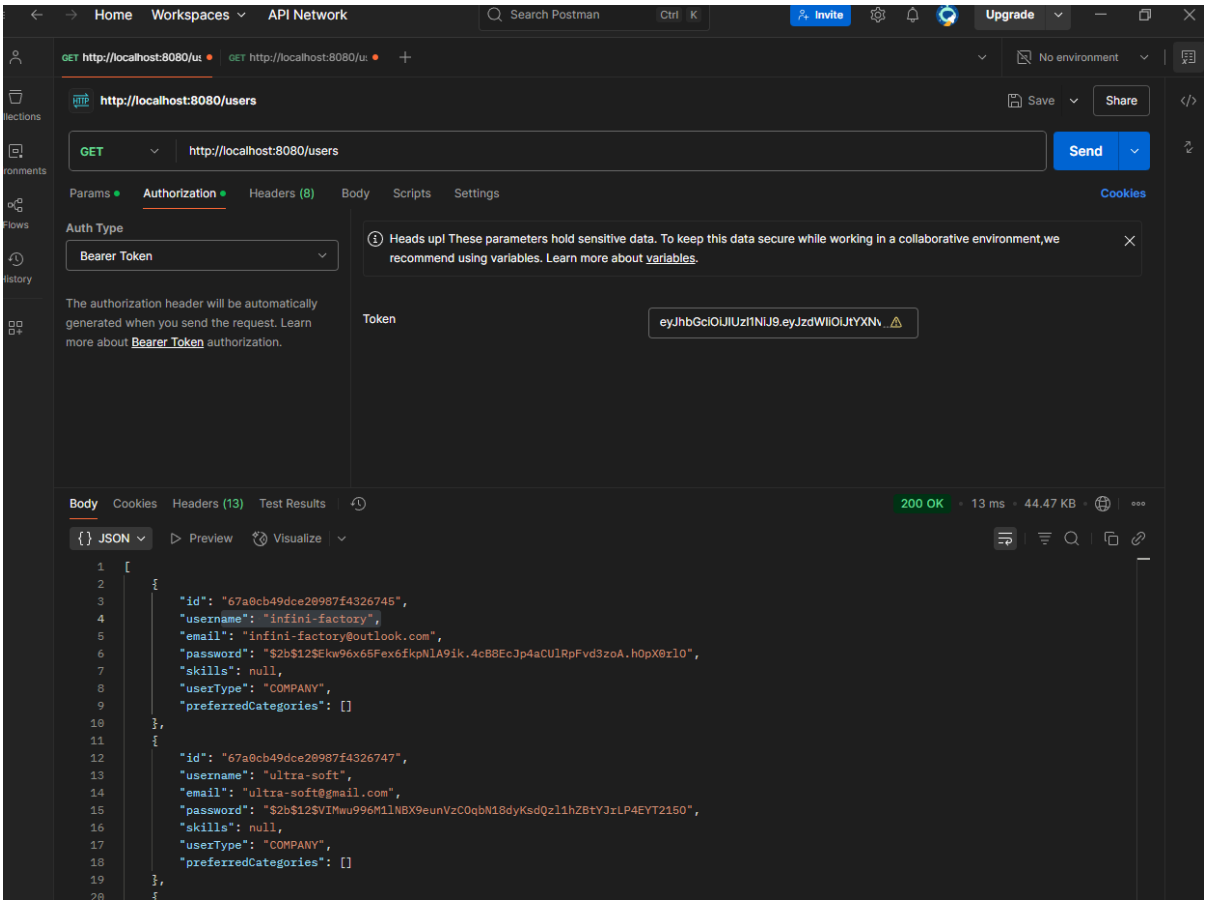
Chaque test a confirmé que :

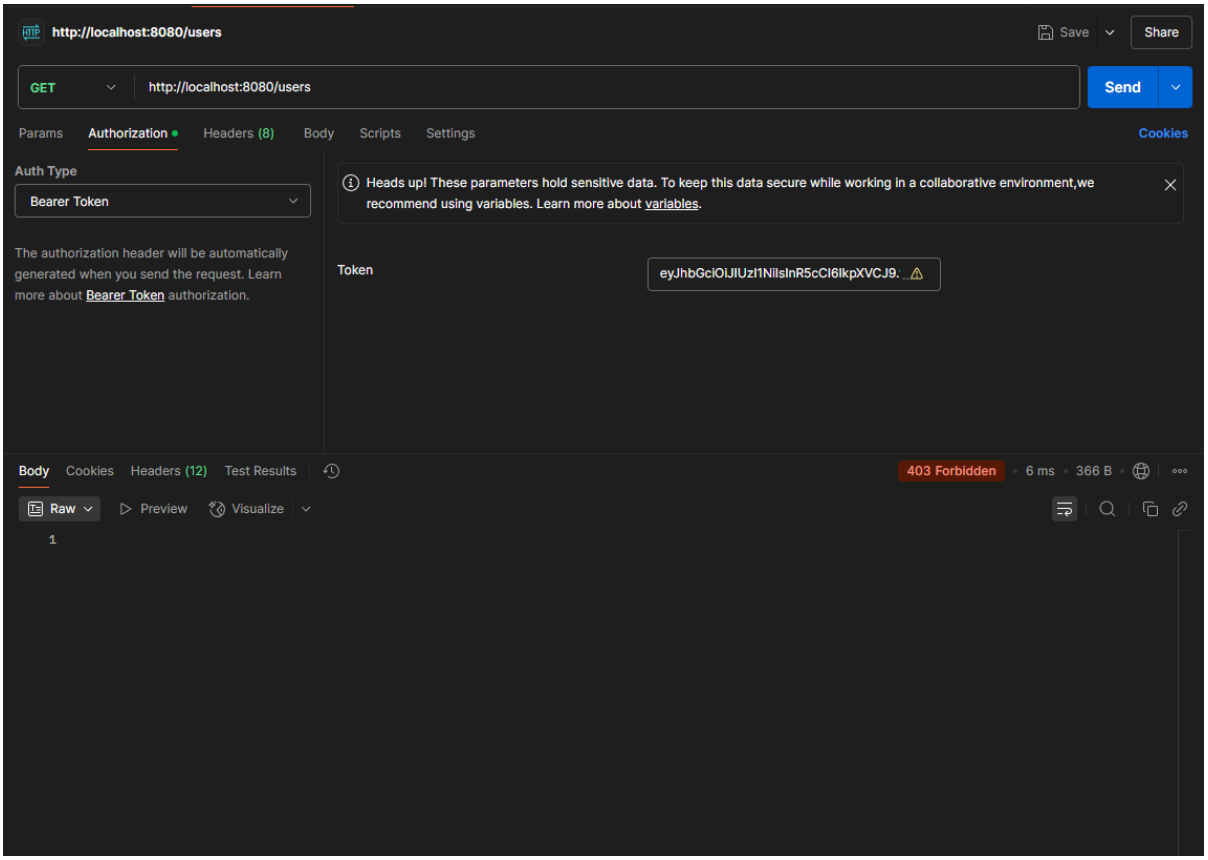
- Les endpoints publics restent accessibles sans authentification.
- Les endpoints protégés refusent l'accès sans token ou avec un token incorrect.
- Un token JWT valide permet d'accéder aux ressources sécurisées, avec un retour attendu de 200 OK.

Exemple de Validation avec Postman

Des captures d'écran ont été réalisées pour illustrer ces tests, notamment :

- La réception correcte du 403 Forbidden lors de l'utilisation d'un token invalide.
- La réussite de la connexion via POST /users/login, retournant un 200 OK avec le token JWT généré.





3.4.1.Résultats des Tests

Les tests ont permis de s'assurer que l'application est stable, cohérente et conforme aux exigences fonctionnelles définies. Quelques améliorations ont été identifiées (notamment l'automatisation de certains tests), mais l'ensemble des fonctionnalités principales est validé et opérationnel.

Avant le passage en production, toutes les traces de logs contenant des informations sensibles, telles que les tokens JWT ou les identifiants utilisateurs, ont été soigneusement supprimées. Cette étape a permis de garantir la confidentialité des données et de respecter les bonnes pratiques en matière de sécurité applicative.

Préparation, Insertion et Homogénéisation des Données

Afin de simuler un environnement complet pour les tests et démonstrations de l'application WorkMatch, un important travail d'initialisation des données a été effectué sur MongoDB.

Tout d'abord, 200 utilisateurs ont été générés, dont 130 de type INDIVIDUAL (chercheurs d'emploi) et 70 de type COMPANY (entreprises). Les mots de passe de tous les utilisateurs ont été hashés avec l'algorithme bcrypt pour respecter les bonnes pratiques de sécurité.

Chaque utilisateur de type INDIVIDUAL a été enrichi d'un document jobSearcher, incluant des compétences techniques, des préférences salariales et des localisations réalistes. Ces données ont été générées de manière semi-aléatoire, mais toujours cohérente avec le marché ciblé.

Plus de 1000 offres d'emploi ont ensuite été récupérées depuis l'API Adzuna, en filtrant uniquement celles issues du domaine informatique (IT) aux USA. Chaque offre a été liée à une entreprise existante via le champ companyId, afin de garantir des relations cohérentes entre les entités.

Pour garantir l'intégrité des données, des scripts de nettoyage ont été appliqués. Ils ont permis d'éliminer les doublons, de supprimer les champs inutiles ou obsolètes, et de vérifier la cohérence des relations (comme la correspondance entre les offres et les entreprises).

Les localisations des chercheurs d'emploi ont été attribuées de manière réaliste : une majorité provient directement des villes associées aux offres d'emploi existantes, tandis qu'une minorité a été ajoutée pour diversifier les profils géographiquement.

Un algorithme de catégorisation simple a été mis en place pour classer automatiquement les offres dans des catégories telles que "Software Developer", "Data Science" ou "DevOps", en se basant sur les mots-clés présents dans le titre ou la description de l'offre.

Enfin, les compétences techniques ont été générées automatiquement en fonction du type de poste. Par exemple, une offre pour un "Java Developer" est automatiquement associée à des compétences telles que Java, Spring ou Hibernate. Du côté des chercheurs d'emploi, une répartition intelligente a été mise en œuvre afin de simuler des profils variés, augmentant ainsi la pertinence des tests de matching.

3.4.2.Développement des Fonctionnalités Avancées

Modification et Optimisation des Fonctionnalités Utilisateur

Dans le cadre de l'amélioration continue de l'application, plusieurs fonctionnalités avancées ont été développées afin d'optimiser l'expérience utilisateur :

- **Édition des Offres d'Emploi :**
  Une page dédiée (EditOfferPage) a été mise en place, permettant aux entreprises de modifier leurs offres existantes. Tous les champs essentiels (titre, description, type de contrat, télétravail, salaire, villes et compétences) peuvent être mis à jour via un appel sécurisé au backend.
- **Gestion des Messages Non Lus :**
  Pour garantir une expérience fluide dans la messagerie, un système de réinitialisation des messages non lus a été intégré dès qu'un utilisateur quitte une conversation.

- **Scroll Automatique dans le Chat :**
  Un scroll automatique a été configuré afin d'afficher en permanence le dernier message lors d'une discussion active, améliorant la lisibilité et l'interaction en temps réel.

4. Appréciation Personnelle

4.1. Rapport avec les Cours

La réalisation de ce projet a été l'occasion concrète de mettre en pratique les connaissances acquises tout au long de mon bachelier. Bien que les cours m'aient apporté des bases solides en développement backend avec Spring Boot, en gestion de bases de données ou encore en création d'interfaces, j'ai rapidement constaté que la mise en œuvre d'une application complète comme WorkMatch nécessitait d'aller bien au-delà de ce qui avait été vu en classe.

4.2. Difficultés Rencontrées

L'une des principales difficultés que j'ai rencontrées concernait justement cette communication entre les différentes couches de l'application. La gestion des erreurs CORS, par exemple, s'est révélée bien plus complexe que je ne l'imaginais.
De même, la configuration de Spring Security combinée aux tokens JWT a nécessité de nombreuses recherches et ajustements avant d'obtenir un système à la fois sécurisé et fonctionnel.
La mise en place du temps réel via WebSocket a également été un défi, notamment pour assurer la stabilité des connexions et la gestion des notifications en cas de déconnexions intempestives.
Ces obstacles, bien que parfois décourageants, ont été formateurs et m'ont permis de développer des réflexes essentiels en matière de résolution de problèmes et d'autonomie technique.

4.3. Suggestions et Améliorations

4.3.1. Utilité pour la Vie Professionnelle

Avec du recul, je mesure à quel point ce projet m’a permis de comprendre la réalité du développement d'une application moderne, bien au-delà des exercices théoriques.
Il m’a appris à anticiper les problèmes liés aux environnements de déploiement, à mieux structurer mon code pour faciliter la maintenance, et surtout à toujours garder en tête l'expérience utilisateur finale.
Je suis convaincu que cette expérience me sera précieuse dans ma vie professionnelle, car elle

m’a offert une vision globale du cycle de développement, de la conception à la mise en production.

4.3.2.Ce Que Je Changerais

Si je devais refaire un projet similaire, je prendrais davantage le temps de documenter chaque étape technique et d'intégrer des outils de tests automatisés dès le début afin de gagner en efficacité sur le long terme.
Ce projet m’a également conforté dans l’idée que la curiosité, la capacité d’adaptation et la persévérance sont des qualités indispensables pour évoluer dans le domaine du développement informatique.

4.3.3.Idées d’Améliorations Futures

Premièrement, l’implémentation d’un **système de matching basé sur l’intelligence artificielle** (machine learning) permettrait d’affiner la pertinence des propositions. En analysant les comportements des utilisateurs (offres likées, temps passé sur une annonce, historiques de matching), un algorithme pourrait progressivement ajuster les suggestions d’offres ou de candidats en fonction de leurs préférences réelles, même implicites.

Deuxièmement, une fonctionnalité de **chat assisté par IA** pourrait être ajoutée pour faciliter la prise de contact entre les entreprises et les candidats. Ce chatbot pourrait proposer des messages automatiques, répondre aux questions fréquentes ou aider à planifier un entretien, réduisant ainsi les temps d’attente et améliorant la réactivité.

Troisièmement, l’ajout de **filtres avancés et personnalisables** pour les candidats et les recruteurs (niveau d’études, disponibilités, langues parlées, technologies spécifiques, etc.) permettrait une recherche encore plus ciblée et précise.

Du côté technique, il serait judicieux de mettre en place une **infrastructure de tests automatisés (unitaires et end-to-end)** pour accélérer les phases de déploiement et garantir la stabilité de l’application. De même, la **migration vers une architecture microservices** pourrait permettre de mieux gérer la scalabilité et de faciliter la maintenance à long terme.

Enfin, pour une ouverture à l’international, l’ajout d’une **gestion multilingue** de l’interface et des données (offres traduites automatiquement, filtres linguistiques, etc.) pourrait élargir considérablement l’audience et permettre à WorkMatch d’être utilisé dans différents pays.

5. Conclusion

5.1. Bilan du Projet

La réalisation de WorkMatch m’a permis de mener à bien un projet complet, depuis sa conception jusqu’à son déploiement, en passant par le développement de ses fonctionnalités clés. Dès le départ, les choix techniques se sont orientés vers des technologies modernes et adaptées aux exigences d’une application de matching et de messagerie en temps réel. Le choix de **Spring Boot** pour le backend s’est révélé judicieux grâce à sa robustesse et sa flexibilité. Pour le frontend, l’utilisation de **React Native Web** a offert l'avantage de développer une interface réactive, accessible sur navigateur, tout en conservant une logique adaptée aux applications mobiles.

Tout au long du projet, ces choix technologiques ont démontré leur pertinence, même si certaines limitations ou complexités ont nécessité des ajustements. La gestion de la sécurité avec JWT, la mise en place d’un système de notifications et de messagerie en temps réel, ou encore la gestion des échanges entre services frontend et backend ont été autant de défis que j'ai su relever progressivement. Le résultat obtenu est une application fonctionnelle, respectant les objectifs initiaux : permettre aux entreprises et aux candidats de se connecter facilement via un système de matching intelligent, enrichi par une messagerie fluide et instantanée.

5.2. Bilan des choix effectués et du résultat obtenu

Ce projet a également été l'occasion de développer des compétences transversales, telles que l’organisation du travail, la gestion des imprévus techniques, et l'adaptation constante face aux problématiques de déploiement. Si certaines parties auraient pu être optimisées davantage avec plus de temps, notamment l'automatisation des tests ou l’amélioration de l’interface utilisateur, le bilan global reste très positif. WorkMatch est aujourd’hui une application stable, déployée, et prête à être utilisée dans un contexte réel. Cette expérience représente une étape importante dans mon parcours, tant sur le plan technique que personnel, et m’a donné des bases solides pour aborder de futurs projets professionnels avec davantage de maîtrise et de confiance.

5.3. Utilisation de l’IA dans le projet

L’intégration de l’Intelligence Artificielle (IA), notamment via des outils comme ChatGPT, a grandement facilité la réalisation de ce projet, tant sur le plan technique que rédactionnel.

Loin de se limiter à une simple assistance ponctuelle, l'IA a été un partenaire précieux à différentes étapes, tout en respectant une démarche critique et autonome.

1. Support Technique et Optimisation du Code

L'IA m'a permis de surmonter des blocages techniques en fournissant des explications claires et des exemples contextualisés. Par exemple :

- Configuration de Spring Security avec JWT : ChatGPT a clarifié le fonctionnement des filtres d'authentification et la gestion des tokens, en proposant des extraits de code adaptés à mon architecture. Cela a accéléré la mise en place de ces fonctionnalités sans sacrifier la compréhension.

- Scripts Python pour la gestion de la base de données : J'ai utilisé l'IA pour générer des scripts automatisés (nettoyage de données, migrations), que j'ai ensuite ajustés pour correspondre à mes besoins spécifiques.

- Reformulation et optimisation de code : Dans certains cas, l'IA a suggéré des refactorisations pour rendre le code plus lisible (par exemple, simplifier des conditions complexes ou améliorer la structure d'une fonction).

2. Structuration et Rédaction

L'IA a également joué un rôle dans l'amélioration de la qualité rédactionnelle :

- Clarté et fluidité : Certaines explications techniques ont été reformulées pour être plus accessibles, notamment dans la documentation ou les commentaires de code. Par exemple, des descriptions trop jargonnantes ont été simplifiées sans perdre en précision.

- Corrections grammaticales et syntaxiques : Les outils d'IA ont détecté des fautes subtiles (accords, tournures maladroites) et proposé des alternatives, ce qui a réduit le temps de relecture.

- Organisation des idées : Pour des sections complexes (comme l'architecture backend ou les workflows), l'IA a aidé à structurer le contenu en suggérant des enchaînements logiques entre les paragraphes.

3. Limites et Approche Critique

Bien que très utile, l'IA n'a jamais remplacé ma réflexion :

- Adaptation nécessaire : Les suggestions (code ou texte) devaient systématiquement être vérifiées, testées, et parfois ajustées. Par exemple, certaines solutions génériques ne tenaient pas compte de contraintes spécifiques (compatibilité avec des librairies, performances).

- Vérification croisée : Pour les concepts techniques, je me suis appuyé sur des sources complémentaires (documentations officielles, forums) pour valider les informations fournies par l'IA.

4. Gain de Temps et Apprentissage

L'IA a été un accélérateur, mais aussi un outil d'apprentissage :

- En expliquant des erreurs ou en proposant des alternatives, elle m'a aidé à approfondir ma compréhension (par exemple, sur la gestion des CORS ou les bonnes pratiques REST).

- Le temps gagné sur les tâches répétitives (corrections, génération de snippets) m'a permis de me concentrer sur des défis plus complexes.

Conclusion : Un Outil Responsable

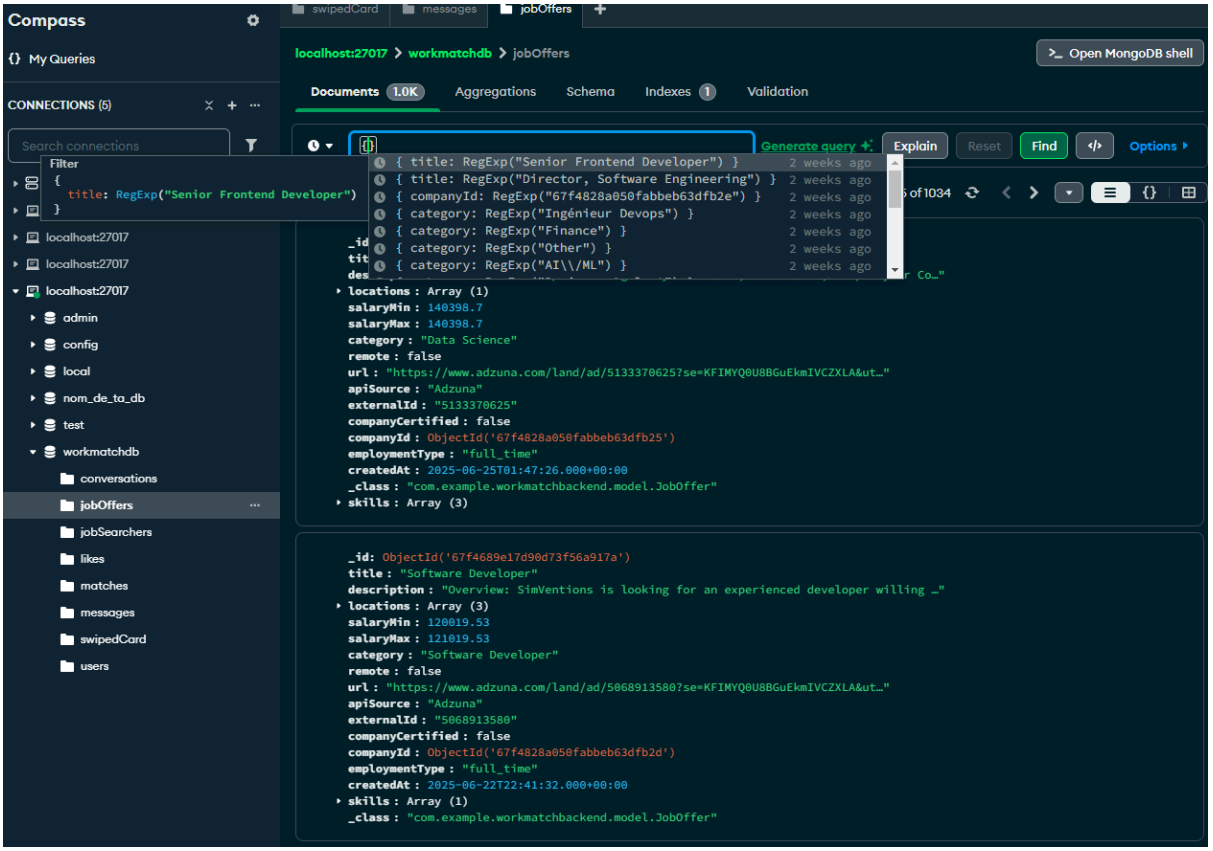
L'IA a été un levier efficace pour ce projet, à condition de l'utiliser avec discernement. Son vrai apport réside dans sa capacité à débloquer des situations, à suggérer des pistes, et à optimiser certaines tâches. Cependant, la maîtrise du sujet et la rigueur technique restent indispensables : chaque ligne de code ou phrase reformulée a été relue, testée, et validée. Cette expérience renforce ma conviction que l'IA doit être un compagnon de travail, et non une solution miracle – une approche que j'entends poursuivre dans mes futurs projets professionnels.

6. Bibliographie

- **Spring Boot Documentation Officielle**
  https://spring.io/projects/spring-boot
- **Spring Security Reference**
  https://docs.spring.io/spring-security/reference/index.html
- **MongoDB Documentation**
  https://www.mongodb.com/docs/
- **MongoDB Atlas – Cloud Database Service**
  https://www.mongodb.com/cloud/atlas
- **React Native Documentation**
  https://reactnative.dev/docs/getting-started
- **React Native Web Documentation**
  https://necolas.github.io/react-native-web/docs/
- **Axios – Documentation Officielle**
  https://axios-http.com/docs/intro
- **STOMP over WebSocket Protocol**
  https://stomp.github.io/stomp-specification-1.2.html
- **SockJS Documentation**
  https://github.com/sockjs/sockjs-client
- **Render – Cloud Hosting for Developers**
  https://render.com/docs
- **Netlify Documentation**
  https://docs.netlify.com/
- **JWT (JSON Web Token) Introduction**
  https://jwt.io/introduction
- **Postman – API Platform**
  https://www.postman.com/
- **Expo pour React Native Web**
  https://docs.expo.dev/workflow/web/
- **ChatGPT – OpenAI**
  https://openai.com/chatgpt

7. Annexes

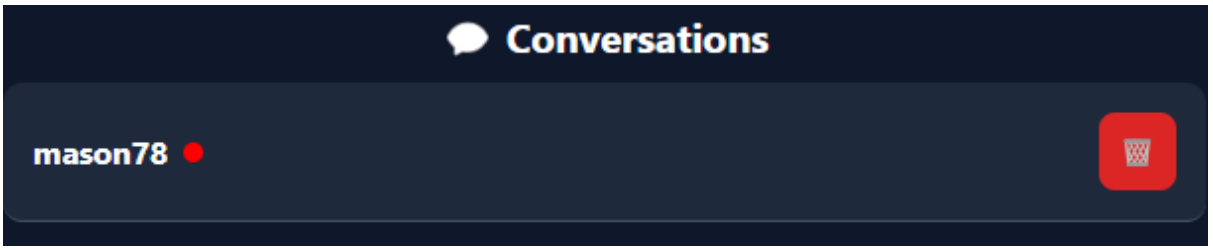
Rechercher spécifique sur MongoDB et interface :



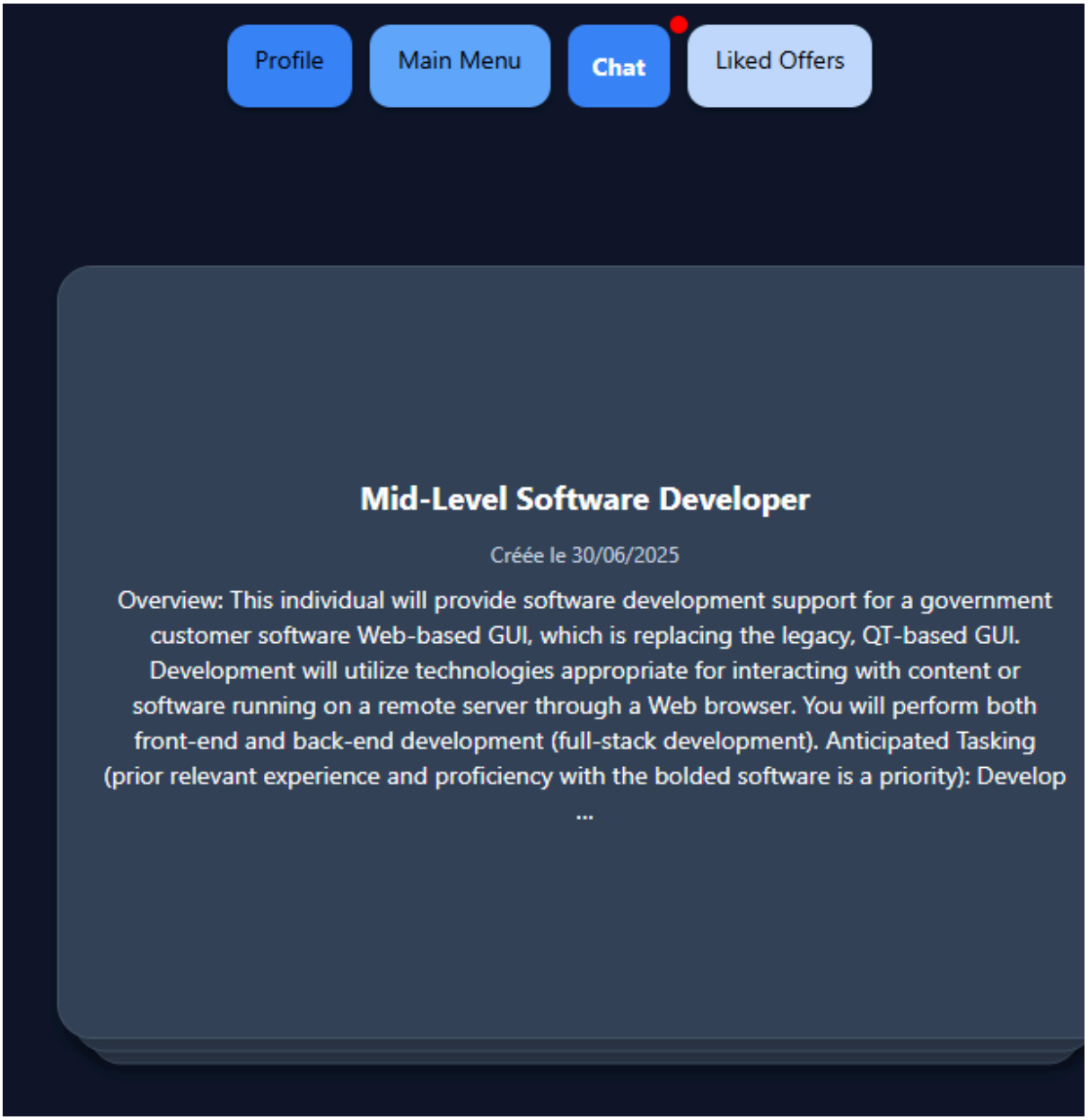
Exemple script python généré avec l'IA :

```python
from pymongo import MongoClient
from bson import ObjectId

# Connexion à MongoDB
client = MongoClient("mongodb://localhost:27017/")
db = client["workmatchdb"]
users_collection = db["users"]

# Liste des ObjectId des users à supprimer
user_ids_to_delete = [
    "67f8653d4086291a97495857",  # synapse-zone
    "67f8688b4086291a9749585c",  # global-systems
    "67f86d9d4086291a9749586f",  # digital-group
    "67f870754086291a97495878",  # cyber-works
    "67f876524086291a9749587d",  # byte-hub
    "67f8806a4086291a97495881",  # byte-zone
    "67f882e84086291a97495892",  # code-institute
    "67f883f94086291a97495893",  # global-partners
    "67f8840e4086291a97495894",  # cyber-group
    "67f8848c4086291a97495896",  # byte-ventures
]

# Suppression des utilisateurs
result = users_collection.delete_many({"_id": {"$in": [ObjectId(uid) for uid in user_ids_to_delete]}})

print(f"✅ Utilisateurs supprimés : {result.deleted_count}")
```

Réception notif pour message dans chatpage :



Réception notif pour match ou message hors chatpage :



Page myOffers pour un COMPANY :

My Offers