

Dossier de projet professionnel



SKILLLY

Réalisation de l'application Mobile SKILLLY

Présenté par Thomas Spinec

SOMMAIRE

INTRODUCTION.....	4
Présentation personnelle.....	5
Présentation du projet.....	5
Contexte.....	5
Objectif.....	5
Cibles.....	5
Fonctionnalités clés.....	5
Technologies utilisées.....	6
Compétences Couvertes par le projet.....	8
Analyse de l'existant.....	9
Contexte du marché.....	9
Pourquoi SKILLY ?.....	10
Les utilisateurs du projet.....	10
Fonctionnalités attendues.....	11
Organisation du projet.....	13
Gestion de projet.....	13
Méthodologie.....	13
Création des tickets.....	14
Collaboration renforcée.....	14
Conception de l'application :	15
Design System & Conception UX/UI de l'application.....	16
1 - MoodBoard d'inspiration :	16
2 - Design System :	17
3 - Maquettes UX et parcours utilisateurs :	20
Conception de la base de données.....	22
1. Modèle Conceptuel de Données (MCD).....	23
2. Modèle Logique de Données (MLD).....	24
3. Modèle Physique de Données (MPD).....	25
Développement du backend de l'application.....	26
Organisation générale.....	26
Arborescence du projet.....	27
Fonctionnement de l'API.....	27
Middleware.....	28
Routage.....	28
Contrôleurs (Controllers).....	28
Services.....	28
Modèles (Models).....	29
Sécurité.....	29
Gestions des erreurs.....	30

Documentation de l'API : SWAGGER.....	33
Développement du frontend.....	36
Arborescence et navigation.....	36
Pages & composants.....	37
Intégration de l'API.....	38
Sécurisation des appels avec AXIOS.....	38
Tanstack.....	39
Socket.....	40
Messagerie instantanée avec WebSocket (Gorilla).....	40
Architecture générale.....	40
Fonctionnement simplifié.....	41
Sécurité et performances.....	42
Stockage des messages.....	42
Diffusion globale.....	42
Intégration côté frontend.....	43
Les tests.....	43
Tests automatisés.....	43
Architecture générale.....	44
Cahiers de recette.....	52
Déploiement et Démarche DevOps.....	55
Conteneurisation avec Docker.....	55
Veille technologique & résolution de problème.....	57
Exemple de veille : incompatibilité entre react-native-deck-swiper et Expo SDK 53.....	58
Conclusion.....	59

INTRODUCTION

SKILLY est une application mobile de mise en relation entre développeurs et recruteurs du secteur tech, conçue pour simplifier et accélérer le processus de recrutement. Le projet est né d'un constat terrain : les plateformes actuelles sont souvent impersonnelles, peu ciblées, et frustrantes pour les deux parties.

D'un côté, les candidats multiplient les candidatures sans retour ; de l'autre, les recruteurs reçoivent des profils qui ne correspondent pas à leurs besoins.

Nous avons imaginé SKILLY comme une application de **matching intelligent** et de communication directe, à la manière d'un Tinder professionnel, mais orienté vers le recrutement technique.

L'application propose une interface de **swipe** pour permettre aux candidats de découvrir des offres adaptées à leur profil, et aux recruteurs de sélectionner des profils pertinents. Lorsqu'un match a lieu, les utilisateurs peuvent immédiatement **échanger via une messagerie instantanée**, sécurisée et en temps réel, intégrée dans l'application.

Sur le plan technique, SKILLY repose sur un **frontend développé en React Native** pour assurer une compatibilité iOS et Android.

Le **backend est développé en Go**, un langage choisi pour sa performance, sa simplicité et sa capacité à gérer efficacement la concurrence. L'ensemble est connecté à une **base de données PostgreSQL**, garantissant la fiabilité et la structuration des données.

Les données de messagerie sont stockées dans une base **MongoDB**, et les échanges temps réel sont assurés via **WebSocket**, grâce à la bibliothèque Gorilla.

Côté frontend, nous avons intégré **TanStack Query** pour la gestion des requêtes et du cache, et **Axios** pour intercepter les appels sortants et injecter automatiquement le token JWT. Le backend expose une API REST sécurisée, documentée via **Swagger**, ce qui a permis une collaboration fluide entre les couches front et back.

L'ensemble du projet est conteneurisé avec **Docker**, et une intégration continue est en place via **GitHub Actions**, facilitant le travail en équipe et le contrôle qualité.

Plus qu'un simple projet pédagogique, SKILLY est une application fonctionnelle et évolutive, qui répond à un vrai besoin et qui nous a permis de mettre en pratique l'ensemble des compétences acquises tout au long de notre parcours.

Présentation personnelle

Je m'appelle Thomas Spinec, j'ai 28 ans. J'ai découvert la programmation en Licence et Master de biologie marine, dans lesquels j'utilisais *R* et *python*. Cela m'a beaucoup plu, et j'ai décidé de m'orienter vers ce domaine. Je me suis donc inscrit à La Plateforme_, dans le parcours Start, dans lequel j'ai obtenu mon titre de développeur web et web mobile. Je suis aujourd'hui en bachelor 3, afin de passer le titre de concepteur développeur d'application, en alternance.

Présentation du projet

Contexte

Dans un contexte où les plateformes de recrutement traditionnelles peinent à offrir des résultats pertinent, rapide et interactif, SKILLY a été conçue comme une application mobile de matching entre développeurs et recruteurs, inspirée du modèle de SWIPE type TINDER.

Elle vise à simplifier, fluidifier, et moderniser le processus de recrutement dans le secteur de la TECH.

Objectif

Le but principal de SKILLY est de réinventer l'expérience de recrutement en offrant une interface intuitive, rapide et mobile, tant pour les développeurs que pour les recruteurs. L'application propose une mise en relation directe entre profils qualifiés et offres pertinentes via un système de matching intelligent et interactif.

Cibles

- Candidats développeurs (FrontEnd, BackEnd, FullStack, Mobile...)
- Recruteurs (StartUps, PME, Grandes entreprises de la TECH)

Fonctionnalités clés

- Création de profils (candidats et recruteurs)
- Matching basé sur les compétences, la localisation, l'expérience et les préférences

- SWIPE pour " liker" ou "passer" , des offres ou profils
- Messagerie instantanée en cas de match mutuel
- Gestion des candidatures (suivi, statut, historique)
- Création et publication d'offres coté recruteurs
- Ajout de CV, Diplômes, Certificats, compétences
- Gestion des entreprises, offres et profils, via interface dédiée

Technologies utilisées

Le choix des technologies pour SKILLY a été guidé par des contraintes techniques, des besoins de performance, ainsi que par les compétences de l'équipe.

FrontEnd - React Native:

Nous avons choisi React Native pour le développement de l'application mobile, principalement pour les raisons suivantes:

- L'équipe disposait déjà d'une bonne maîtrise de Javascript, ce qui a permis une montée en productivité rapide
- React Native permet de développer une seule base de code pour Android et IOS, ce qui optimise le temps de développement
- Il est facile à déployer grâce à Expo, un outil qui facilite l'émulation, le partage et la publication de l'application sans configuration complexe (Expo est aujourd'hui la solution officielle pour lancer React Native)
- L'écosystème riche de composants et la documentation claire en font un excellent choix pour prototyper et livrer rapidement un MVP

BackEnd - GoLang:

Pour la partie serveur, nous avons opté pour GoLang (Go), un langage open source connu pour sa performance et sa fiabilité :

- Rapide à exécuter et peu gourmand en ressources, Go est particulièrement adapté aux systèmes distribués et aux services BackEnd scalables. De plus, il est au moins 5 fois plus rapide que des langages comme php (cette différence peut varier selon ce qui est comparé).
- Il s'agit d'un choix écoresponsable, car ses faibles besoins en CPU et mémoire, permettent de réduire l'impact énergétique des infrastructures cloud
- Son typage fort, ses performances proches du C, et son support natif de la concurrence (via Goroutines) en font un excellent candidat pour des applications temps réel comme la nôtre.

Pour structurer et renforcer notre BackEnd, nous avons intégré les bibliothèques suivantes :

- Gin : Un framework Web minimaliste et performant, utilisé pour construire notre API REST. Gin permet de gérer facilement les routes, middleware et les réponses HTTP. Il est connu pour sa rapidité d'exécution grâce à une architecture légère ce qui en fait un excellent choix pour un backend nécessitant de la scalabilité sans complexité excessive.
- Gorm: C'est un ORM (Object Relational Mapping) pour Go. Il simplifie les interactions avec notre base de données relationnelles (SQL) en nous permettant de manipuler les données sous formes d'objet Go, tout en générant automatiquement les requêtes SQL nécessaires. Gorm améliore la lisibilité du code et réduit les erreurs dans la gestion des données.
- Gorilla WebSocket: cette bibliothèque nous a permis d'intégrer un système de messagerie en temps réel entre les candidats et les recruteurs. Gorilla WebSocket gère la communication bi-directionnelle persistante entre le client et le serveur, ce qui est essentiel pour les échanges instantanés et fluide dans une application de type "matching" comme SKILLLY.

Base de données - SQL et NoSQL:

- Nous avons utilisé une base relationnelle SQL pour gérer les données structurées de l'application (profils, offres, candidatures, compétences ...), ce qui garantit l'intégrité des données et facilite les relations complexes
- Pour la messagerie en temps réel (rooms et messages), nous avons fait le choix du NoSQL, plus souple et plus performant pour stocker des échanges asynchrones, faiblement structuré et en grande quantité

Environnement de développement:

Bien que l'équipe ait travaillé sur des postes sous MacOS, Linux et Windows, cet aspect était peu important du fait du choix de React Native, qui permet un développement multiplateforme indépendant du système d'exploitation.

En revanche, pour la partie BackEnd développée en Go, nous avons mis en place une conteneurisation avec Docker. Cette approche présente plusieurs avantages:

- Elle garantit un environnement identique pour tous les développeurs, quelle que soit leur machine
- Elle facilite les déploiements en production en éliminant les erreurs liées à des différences d'environnements
- Elle s'inscrit dans une démarche DevOps moderne et professionnelle, avec une architecture plus fiable, évolutive et isolée
- Elle permet d'anticiper des usages en intégration continue (CI/CD) pour les prochaines étapes du projet

Cette décision technique renforce la stabilité du BackEnd et améliore la collaboration au sein de l'équipe de développement.

Compétences Couvertes par le projet

Activité 1 - Développer une application sécurisée :

- Installer et configurer l'environnement de travail en fonction du projet
- Développer des interfaces utilisateurs
- Développer des composants métier
- Contribuer à la gestion d'un projet informatique

Activité 2 - Concevoir et développer une application sécurisée organisée en couches:

- Analyser les besoins et maquetter une application
- Définir l'architecture logicielle
- Concevoir et mettre en place une base de données relationnelle
- Développer des composants d'accès aux données SQL et NoSQL

Activité 3 - Préparer le déploiement d'une application sécurisée :

- Préparer et exécuter les plans de tests
- Préparer et documenter le déploiement
- Contribuer à la mise en production dans une démarche DevOps

Compétences transversales mobilisées:

- Communiquer en français et en anglais (Documentation technique, interface utilisateur)
- Mettre en oeuvre une démarche de résolution de problème (débug, choix technologiques, gestion d'erreur)

- Apprendre en continu (veille sur les composants, sécurité, évolutions technologiques)

Analyse de l'existant

Contexte du marché

Le secteur du recrutement en ligne est dominé par de grandes plateformes telles que Indeed, Monster, Jora, Linkedin ... Ces outils répondent aux besoins de mise en relation entre entreprises et candidats, mais il reste souvent basé sur des modèles traditionnels: formulaires de recherche, candidatures par email ou portail web, échange par mail, sans personnalisation avancée ni expérience utilisateur fluide.

Limite des solutions existantes:

Coté Candidats:

- Résultats de recherche peu pertinent: les suggestions sont souvent générique ou ne prennent pas réellement en compte les compétences ou préférences du candidat
- Offres obsolètes ou incomplètes: certaines annonces ne sont plus valides ou manquent d'informations essentielles (salaire, mission précise, type de contrat...)
- Manque de feedback: les candidats restent souvent sans réponse après leur candidatures, ce qui génère de la frustration et un sentiment d'opacité
- Parcours utilisateur fastidieux: répétition des mêmes informations sur chaque plateforme, absence de fluidité ou de centralisation des candidatures

Coté Recruteurs:

- Manque de ciblage: les offres sont vues par des profils non pertinents, rendant le tri long et inefficace
- Visibilité limitée des offres: Sans options sponsorisées, les annonces peinent à se démarquer
- Interface complexe : certains outils (comme Monster) sont jugés lourd, peu intuitif et difficilement personnalisable
- Peu d'interaction direct avec les candidats: les échanges sont lent, rarement centralisé et souvent unilatéraux

Synthèse des besoins non couverts:

Les retours d'expériences montrent des attentes claire:

- Pour les chercheurs d'emploi: trouver rapidement des offres adaptés à leur profil et obtenir un retour, même négatif
- Pour les recruteurs: accéder directement à des profils qualifiés, faciliter le tri, centraliser les candidatures et réduire les délais d'embauche

Pourquoi SKILLY ?

Face à ces constats, SKILLY apporte des réponses concrètes:

- Matching intelligent basé sur les compétences, préférences, disponibilité et type de contrat
- Candidatures en un swipe: UX inspiré des apps de rencontre pour un engagement rapide et instinctif
- Messagerie intégrée en temps réel: pour faciliter les échanges dès qu'un match est établi
- Réponse automatique aux candidats non retenus: Pour améliorer la transparence du processus
- Application mobile multi plateforme, moderne, rapide, intuitive, pensée pour les développeurs et les recruteurs

Les utilisateurs du projet

L'application SKILLY s'adresse à deux types d'utilisateurs bien distincts, avec des attentes spécifiques: les développeurs en recherche d'opportunités professionnelles, et les recruteurs en quête de profils qualifiés dans le domaine tech.

Du côté des candidats, l'objectif est de simplifier le processus de recherche d'emploi, souvent long, répétitif et frustrant sur les plateformes traditionnelles. Les développeurs veulent pouvoir trouver des offres rapidement, sans avoir à remplir dix fois le même formulaire ou envoyer des candidatures à l'aveugle. Ils souhaitent également être visibles sur les compétences réelles, recevoir des retours clairs, et pouvoir échanger facilement avec les entreprises intéressées.

Du côté des recruteurs, les enjeux sont inversés mais complémentaires: ils doivent trier efficacement les candidatures, cibler les bon profils, gagner du temps, et rendre le processus de recrutement plus fluide. Beaucoup d'outils actuels sont complexes, mal adaptés aux petites structures, et n'offrent pas de solution simple pour entrer en contact avec des profils pertinents.

C'est pourquoi SKILLYY propose une expérience scindée en deux parcours utilisateurs, chacun avec une interface et des fonctionnalités spécifiques:

- Pour les développeurs: création du profil, matching intelligent, swipe, suivi des candidatures, messagerie en cas de match.
- Pour les recruteurs: création d'offres, tri des candidats, gestion centralisée des candidatures, messagerie directe, réponses automatiques

Cette approche permet de garantir une expérience optimisée et ciblée pour chaque rôle, tout en facilitant la mise en relation rapide et pertinente entre les deux parties

Fonctionnalités attendues

Le projet SKILLYY repose sur une volonté forte: révolutionner le processus de recrutement tech en le rendant plus rapide, intuitif et interactif. Pour cela, l'application embarque un ensemble de fonctionnalités pensées pour répondre aux besoins concrets des deux profils utilisateurs (développeurs & recruteurs), tout en s'adaptant aux usages mobiles modernes.

L'application est donc organisée autour de deux parcours distincts, chacun avec des propres fonctionnalités:

Pour les candidats développeurs :

- Création de profil personnalisé : bio, années d'expérience, compétences, diplômes, certifications, type de contrat souhaité, localisation, disponibilité
- Import de documents: ajout de CV et justificatifs
- Matching intelligent: système de suggestion d'offres basé sur les compétences, préférences et disponibilités du candidat
- Swipe sur les offres : interaction rapide et intuitive (type TINDER) pour aimer ou passer une offre
- Messagerie en temps réel : échange direct avec un recruteur en cas de match mutuel
- Suivi des candidatures : statut visible (en attente, accepté, refusé) regroupé dans un espace dédié
- Réception des notifications pour les nouveaux matchs, candidatures ou messages
- Gestion autonome du profil: modification des infos, suppression du compte, mise à jour des documents

Pour les recruteurs :

- Création et gestion d'offres d'emploi : titre, description, missions, compétences recherchées, type de contrat, salaire, localisation, etc ...

- Sélection de profils via système de matching en fonction des critères définis dans l'offre.
- Consultation des profils candidats : visualisation de leurs compétences, expériences, CV, certificats
- Messagerie intégrée : discussion directe avec les candidats ayant matché
- Réponses automatiques aux autres candidats non retenus (fonction configurable)
- Tableau de bord de suivi des candidatures : tri par statut (en attente, accepté, refusé), actions rapides (répondre, modifier statut)
- Gestion de l'entreprise : informations publiques de la société, SIRET, taille, secteur, logo, site web

Fonctionnalités transverses (tous utilisateurs) :

- Authentification sécurisée (email/mot de passe ou OAuth via Google, LinkedIn, GitHub)
- Interface fluide et mobile-friendly grâce à React Native + Expo
- Stockage des données sécurisé avec BackEnd en Go, conteneurisé via Docker
- Performance optimisée : architecture légère, API REST, websocket pour la messagerie
- Accès multiplateforme : disponible sur Android ou iOS

Il est important de noter que toutes les fonctionnalités présentées ci-dessus ne sont pas encore implémentées à ce stade du projet (nous avons un POC, ou Proof of Concept, actuellement). Certaines d'entre elles sont en cours de développement ou prévues pour une version future (V1 par exemple), dans une logique d'évolution produit. C'est notamment le cas :

- de l'import de documents (CV, justificatifs, ...), qui n'a pas été priorisé dans ce POC afin de concentrer les efforts sur le cœur fonctionnel
- de la réception des notifications pour les nouveaux matchs, candidatures ou messages. Cette fonctionnalité est en cours d'ajout, il y a déjà le badge dans la navigation pour les messages, mais il n'y a pas encore de notification. La raison est la même que pour l'import de document.
- de la gestion autonome du profil, qui est en partie en place pour les candidats, qui peuvent ajouter ou supprimer des compétences/certifications. Cependant la modification des autres éléments du profil (recruteur ou candidat) n'a pas encore été implémentée (côté front seulement, le CRUD est bien en place dans le back) toujours pour se concentrer sur les fonctionnalités cœur
- de la consultation du profil des candidats, les informations principales sont sur les cartes à swipper, mais il n'y a pas encore de composants d'accès au profil pour un recruteur
- de la réponse automatique, c'est une fonctionnalité que l'on veut implémenter rapidement, ça sera une priorité pour la prochaine étape de développement

- de l'authentification par applications tierces (google, Linkedin, ...), qui n'a également pas été priorisé dans ce POC

Petite mention pour le matching intelligent, dont le fonctionnement précis est encore en phase de conception (pondération des critères, logique de score, architecture, performance). Cette fonctionnalité sera sans doute implémentée dans la prochaine étape de développement, mais son fonctionnement sera certainement amené à être modifié dans les versions ultérieures pour mieux répondre aux besoins.

Organisation du projet

Gestion de projet

Le développement de SKILLY s'est appuyé sur une gestion de projet rigoureuse, menée en trinôme, afin de garantir une organisation claire, une répartition équilibrée des tâches et une visibilité constante sur l'avancement du projet.

Méthodologie

Pour organiser le travail et assurer un bon suivi du déroulement tout au long du développement, nous avons utilisé **GitHub Projects** comme outil central de gestion de projet. Cela nous a permis de garder une vision claire de l'avancement, de prioriser les tâches et de collaborer efficacement.

Notre board était structuré selon une logique kanban, avec cinq colonnes représentant les différentes phases de traitement :

- **Backlog** : liste des idées tâches à venir ou non encore planifiées
- **Ready**: Tickets prêts à être développés
- **In Progress**: Tâche en cours de développement
- **In Review**: Fonctionnalités terminées, en attente de validation
- **Done**: Éléments finalisés et intégrés

La colonne "In Review", a été particulièrement utile pour tester, valider, et faire relire les fonctionnalités avant clôture.

Création des tickets

Chaque ticket était associé :

- à une fonctionnalité (Back -Login, Front - , Create a Job offer, etc ...)
- à une tâche design (Design_RecruterMyOffers, Figma - Design System)
- ou à un objectif transversal (Tests, Swagger, Userflow, etc ...)

Nous utilisions également des labels pour indiquer la nature (frontend, backend, design) ou le rôle principal impliqué.

Cette gestion via GitHub Projects nous a permis de suivre facilement l'état d'avancement des tâches. Cela nous a aussi permis de partager la vision du projet en temps réel et de rester synchronisés malgré la diversité des chantiers. Mais surtout, cela nous a permis de conserver un historique clair des décisions et livrables

Chaque membre du groupe pouvait consulter le tableau, choisir une tâche 'Ready' et passer son ticket en "In Progress", ce qui a fluidifié les échanges et réduit les conflits de version.

The screenshot shows a GitHub Project board titled 'Skillly'. The board has five columns: 'Backlog', 'Ready', 'In progress', 'In review', and 'Done'. Each column contains several items, each with a small profile picture, a ticket number, and a brief description. The 'Backlog' column has 5 items. The 'Ready' column has 0 items. The 'In progress' column has 4 items. The 'In review' column has 6 items. The 'Done' column has 21 items. At the bottom of each column, there is a '+ Add item' button.

Backlog	Ready	In progress	In review	Done
5 / 40	0	4 / 10	6 / 10	21
Estimate: 0	(Estimate: 0)	(Estimate: 0)	(Estimate: 0)	(Estimate: 0)
This item hasn't been started	This is ready to be picked up	This is actively being worked on	This item is in review	This has been completed
skillly #13 algorithme	skillly #28 Figma (Design system + Prototype)	skillly #29 Userflow	skillly #47 Swagger	
Draft closing offer (accept candidate or just close the offer)	skillly #89 TailwindConfig	skillly #58 Design_CandidateProfile	skillly #9 BACK - Login	
Draft automatic messages	skillly #80 Design_UauthRegister	skillly #56 Design_CandidateJobOffers	skillly #12 BACK - Create company	
Draft reviewing	skillly #76 Design_UauthLogin	skillly #62 Design_RecruterIndex	skillly #15 Mise en place tests	
Draft Files	skillly #64 Design_RecruterMyOffers	skillly #67 Design_RecruterApplications	skillly #27 BACK - Applications	
+ Add item	+ Add item	+ Add item	+ Add item	+ Add item

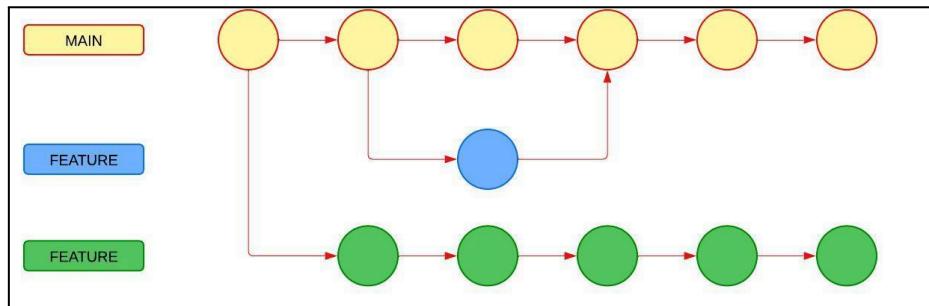
Collaboration renforcée

La création de branche directement via GitHub Project permet de faire un lien entre cette branche et l'issue GitHub, une pull-request sur la branche mettait automatiquement l'issue dans la colonne "In Review". Ce lien entre GitHub Projects, les branches de développement et les pull-requests a également facilité le suivi technique : chaque tâche était liée à une PR, validée puis fusionnée après relecture,

pour assurer une traçabilité propre entre la gestion des tâches et le code source.

Notre gestion de branches s'inspire du modèle GitFlow, simplifié pour notre contexte:

- main : branche stable et déployable
- develop : intégration des fonctionnalités validées
- feature/xxx : développements fonctionnels isolés
- hotfix/xxx : corrections urgentes (rarement utilisées)



Les branches étaient mergées dans develop via une Pull Request, après relecture et test. Une fois validée, la tâche passait automatiquement de "In Review" à "Done" dans le board GitHub.

Conception de l'application :

La conception de notre application s'est appuyé sur une démarche structurée mêlant recherche utilisateur, réflexion technique et design centré sur l'expérience mobile.

Notre objectif était double : concevoir une interface moderne et intuitive, tout en posant les bases d'une architecture logicielle claire, sécurisée et évolutive.

Démarche de conception

Nous avons commencé par une analyse de l'existant (plateforme de matching pro, apps de recrutement mobile), puis défini nos priorités:

- Réduire la friction entre offre et candidat
- Proposer une navigation intuitive
- Mettre en avant les actions clés : postuler, matcher, échanger

Sur cette base, nous avons :

- Recueil et synthèse des besoins utilisateurs, via l'analyse de l'existant et les documents de cadrage fonctionnel
- Création d'un moodboard d'inspiration afin de définir l'univers visuel de l'application (couleurs, typographies, style des composants UI, ambiance)

- Réalisation de maquettes UX/UI en s'inspirant des mobiles actuels (comportement swipe, navigation fluide, boutons accessibles)
- Définition de l'architecture logicielle multicouche à l'aide des diagrammes MCD, MLD et MPD
- Rédaction des spécifications techniques, alignées avec le référentiel du titre CDA
- Choix des technologies, frameworks et outils en fonction des contraintes de performances, de collaboration et de portabilité

Design System & Conception UX/UI de l'application

La conception de l'interface utilisateur de SKILLLY s'est inscrite dans une approche Mobile First, centrée sur l'ergonomie, la lisibilité et la simplicité d'usage.

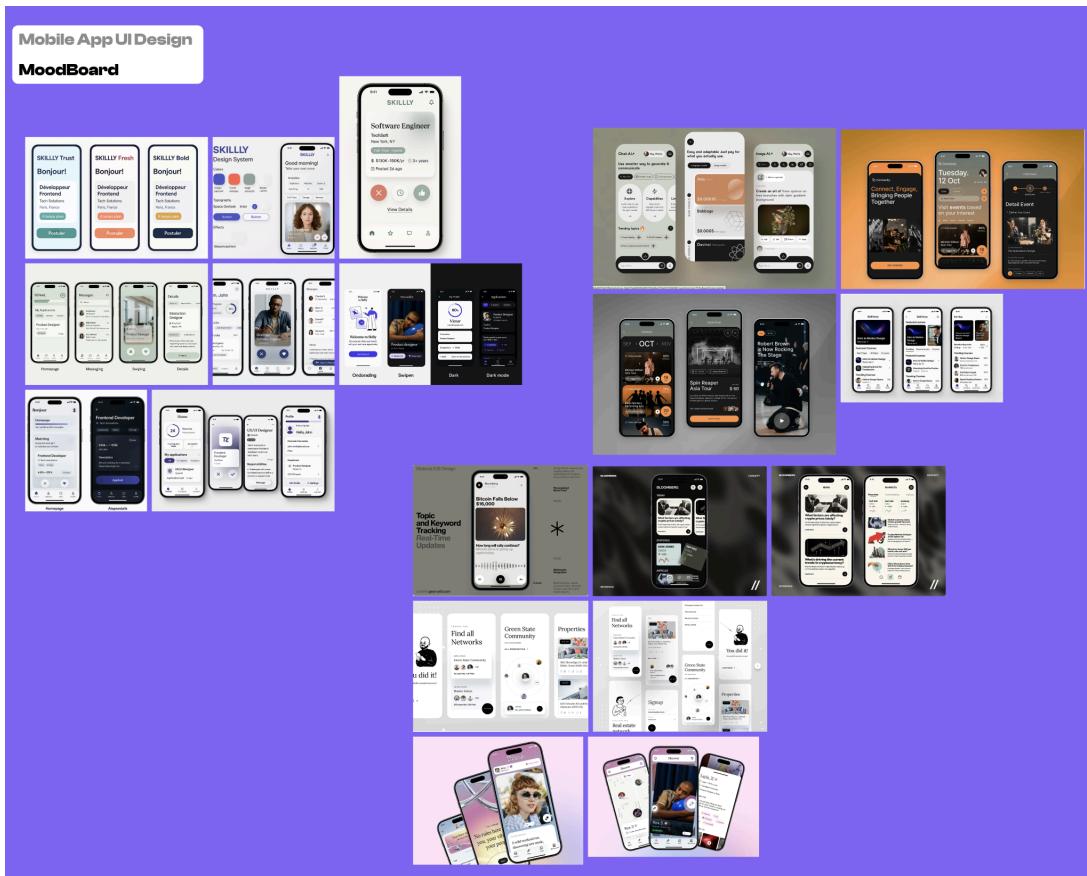
L'objectif était de créer une expérience fluide, intuitive et cohérente, aussi bien pour les développeurs que pour les recruteurs.

Cette démarche UX/UI s'est structurée autour de la mise en place d'un Design System Modulaire, garantissant la cohérence graphique, la maintenabilité et la scalabilité de l'interface.

1 - MoodBoard d'inspiration :

Avant de débuter le maquettage, un moodboard visuel a été créé pour guider la direction artistique de l'application ainsi que pour définir l'ambiance générale, en s'inspirant de :

- Applications à navigation gestuelle (Tinder, Yubo)
- Interfaces claires et dynamiques (Dribbble, Behance, LinkedIn mobile)
- Design Tech moderne orienté recrutement et matching

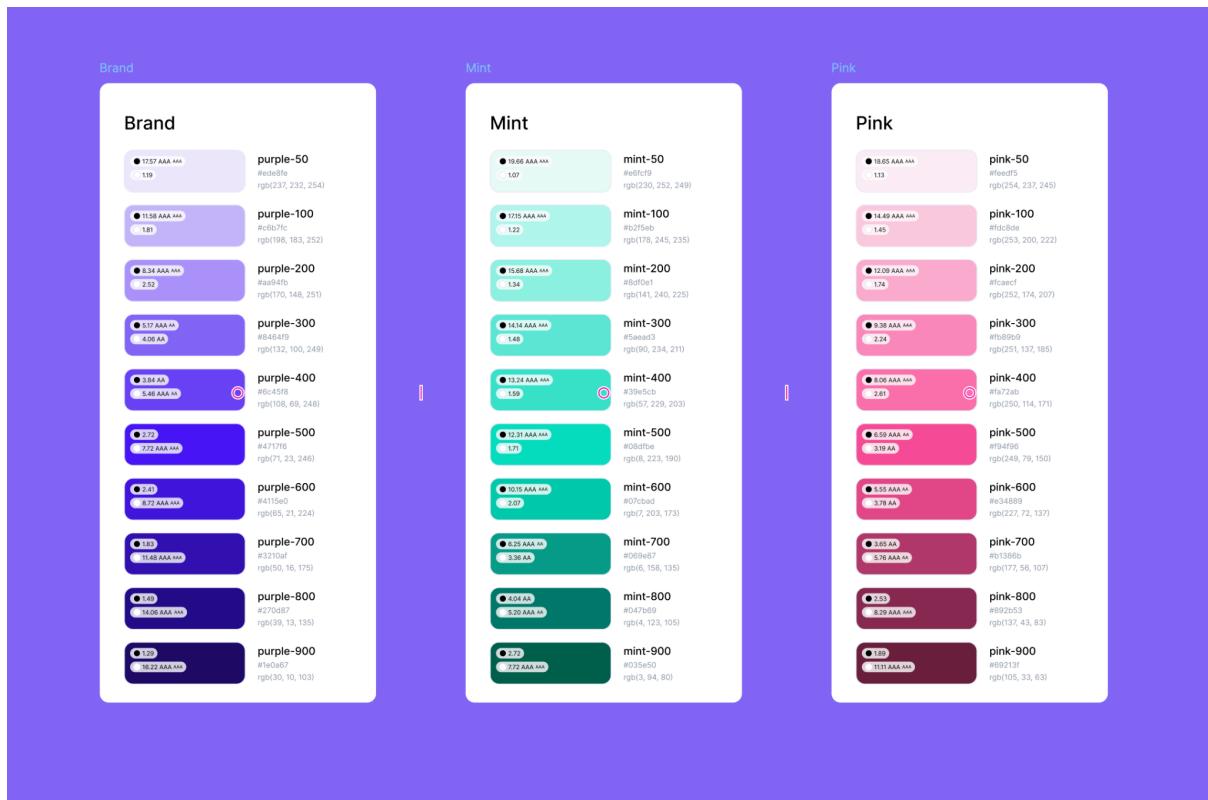


2 - Design System :

Le design system de SKILLLY repose sur quatre piliers fondamentaux : **couleurs, typographies, composants, et règles de hiérarchie visuelle.**

Palette de couleurs:

- **Indigo:** #4717F6 - Couleur dominante, présente en fond et en éléments principaux (CTA)
- **Menthe:** #08DFBE - utilisée pour les CTA secondaires et/ou boutons d'action positifs
- **Rose Pâle:** #F94F96 - utilisée pour attirer l'attention (matchs, alertes, CTA clés)
- **Blanc et gris clair:** Couleur permettant d'aérer l'interface et d'équilibrer les couleurs vives

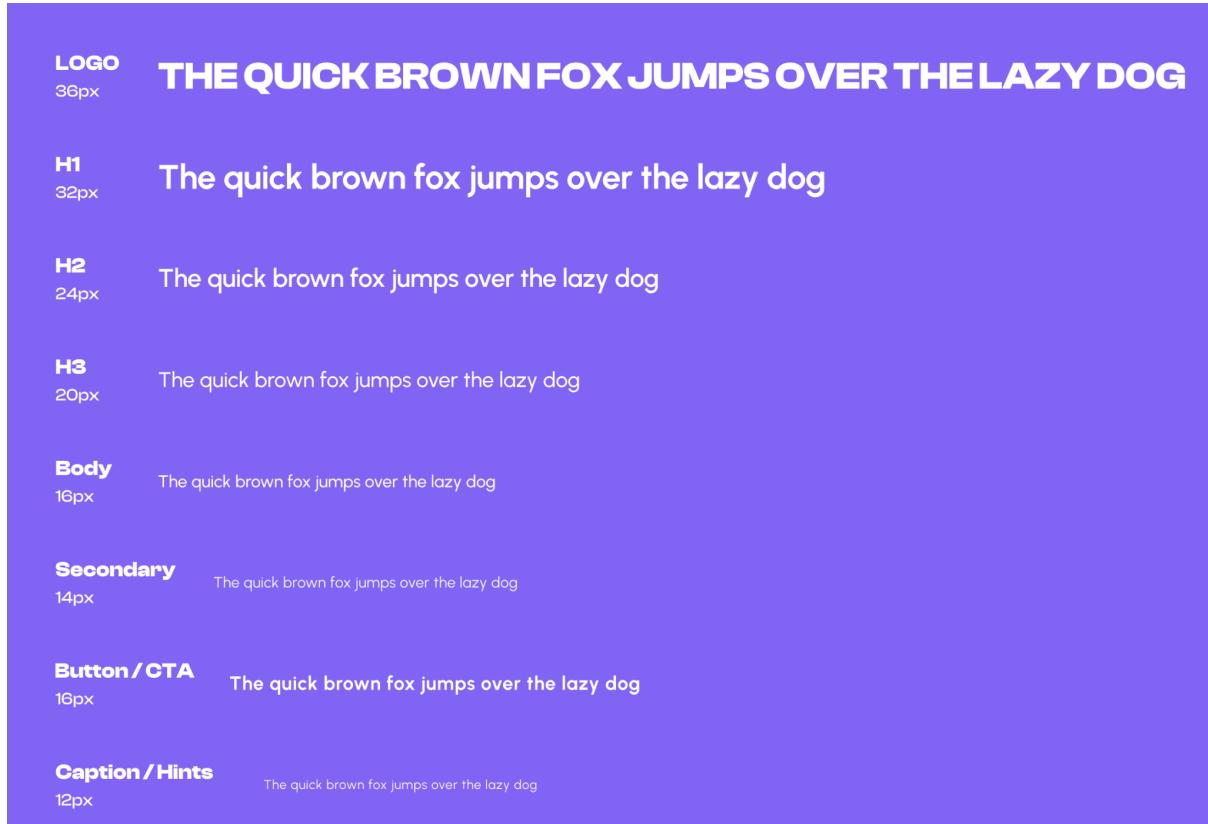


Ces teintes sont utilisées avec des dégradés subtils, notamment pour les boutons, backgrounds et overlays, apporte une touche moderne et vibrante.

Typography

Le système typographique est conçu pour assurer impact visuel, lisibilité et hiérarchie claire, mais surtout pour garder un niveau d'accessibilité élevé :

- **Typographies :**
 - Titres → Clash Display : personnalité, impact, style moderne
 - Texte courant → Urbanist : clarté, stabilité, lisibilité mobile
- **Hiérarchie typographique :**
 - H1 (écran principal) : Clash Display Bold, 36-48px
 - H2-H3 (sous-titres, titres de section) : Clash Display ou Urbanist SemiBold, 24-32px
 - Texte standard: Urbanist Regular, 16-18px
 - Boutons: Urbanist SemiBold, Majuscules
 - Informations secondaires: Urbanist Light, 12-14px

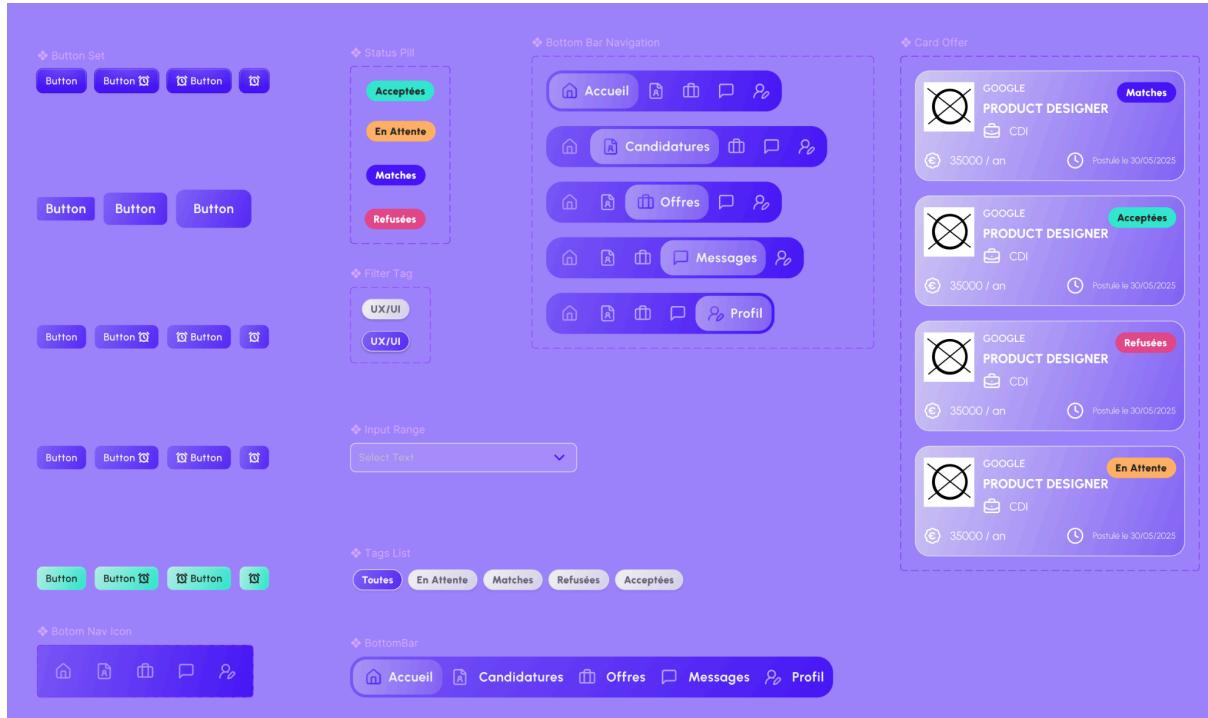


Composants UI & Réutilisabilité

Nous avons conçu une série de composants UI atomiques et réutilisables, en suivant les principes d'un design system structuré :*

- Boutons (CTA primaire / secondaire) : larges, arrondis, accessibles
- Cards : affichage des offres, profils, candidatures
- Badges de statut : (matché, en attente, nouveau message ...)
- Navigation

Ces composants suivent un système de spacing, de couleurs et de typographie unifié, ce qui permet une intégration rapide côté front et une cohérence graphique sur l'ensemble de l'app.



Grâce à cette approche nous avons posé les fondations d'une interface :

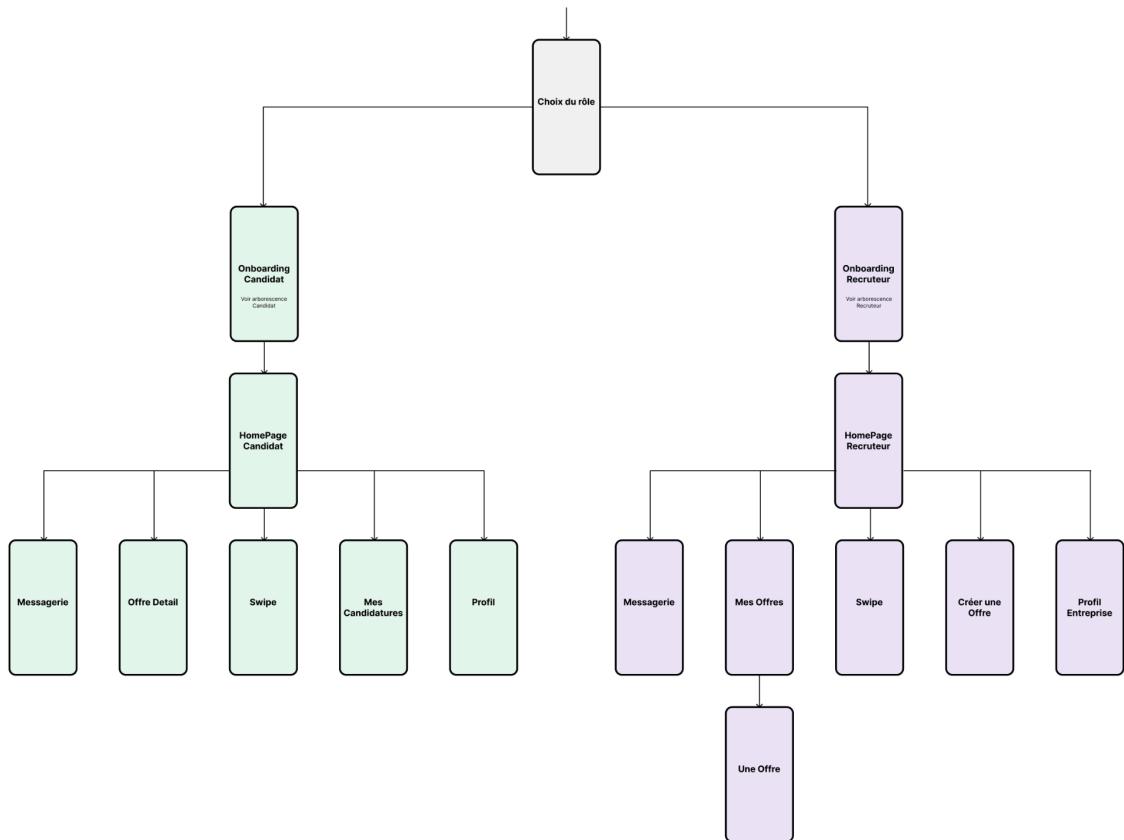
- **Évolutive** (composants réutilisables)
- **Accessible** (contraste, taille des éléments, navigation mobile)
- **Personnalisé** (selon le rôle utilisateur, tout en gardant une cohérence visuelle)

Ce design system est à la fois un outil de conception, une référence technique pour les développeurs, et un socle d'identité pour la marque SKILLY.

3 - Maquettes UX et parcours utilisateurs :

Avant de maquetter visuellement l'application, nous avons établi une arborescence complète des écrans, permettant de :

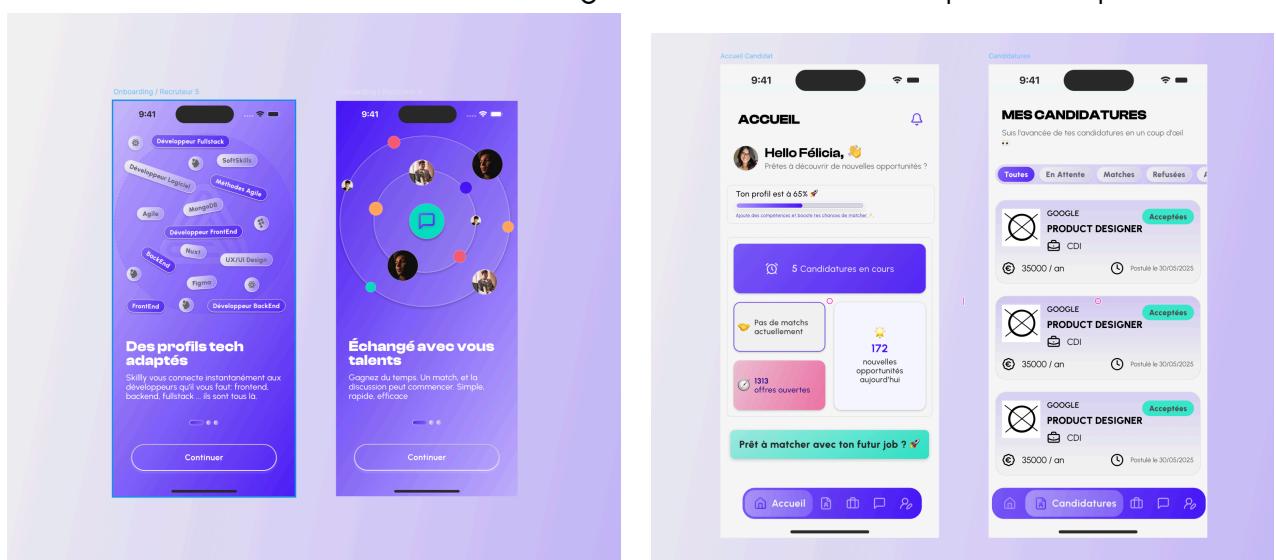
- Identifier les points communs et les spécificités selon les rôles
- Organiser la navigation de manière fluide et logique
- Prendre en compte dès le départ les écrans secondaires : onboarding, messagerie, profil, etc



Navigation et structure d'interface

L'application repose sur une structure de navigation unifiée, déclinée selon le rôle utilisateur :

- Onboarding personnalisé dès l'ouverture de l'app (choix du rôle + création de compte)
- Navigation tabulaire (tab bar) fixe en bas de l'écran
- Écran spécifiques par rôle :
 - Candidat : page de swipe, candidature, suivi des réponses
 - Recruteur : création d'offres, gestion des candidatures, profil entreprise

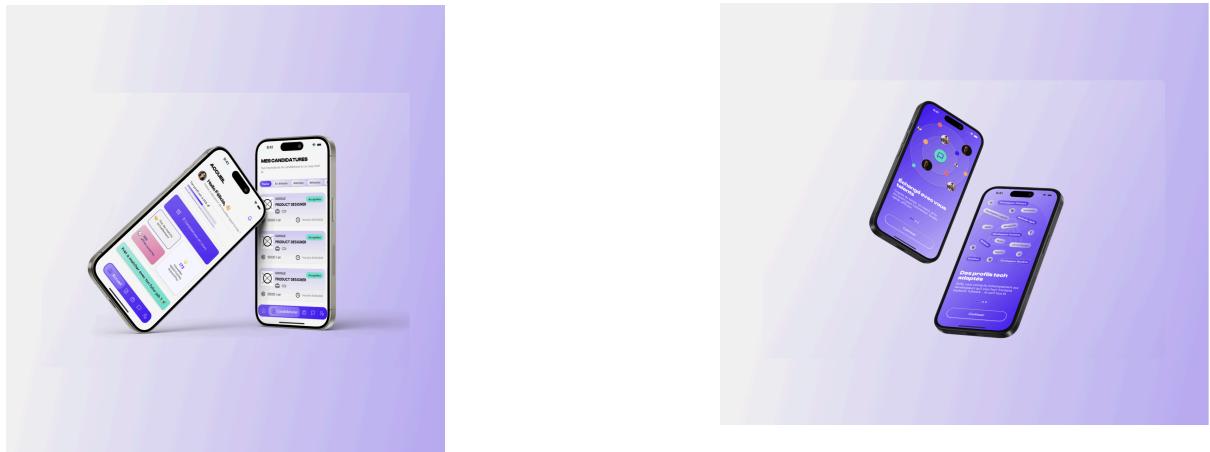


À partir de cette base UX, nous avons conçu des maquettes haute fidélité sur Figma, en appliquant :

- Les composants issus du design system
- Des espacements et des tailles optimisées pour le tactile
- Des états visuels clairs (actif, sélectionné, inactif)

Chaque écran a été conçu pour mettre en avant les actions principales :

- Coté candidat : visualiser une offre, swiper, postuler, échanger
- Coté recruteur : publier une annonce, consulter des candidatures, discuter.



Conception de la base de données

Le cœur fonctionnel de l'application SKILLLY repose sur une gestion rigoureuse de données, notamment :

- Les utilisateurs (candidats et recruteurs)
- Les offres d'emploi
- Les candidatures
- Les compétences, documents, certifications
- La messagerie

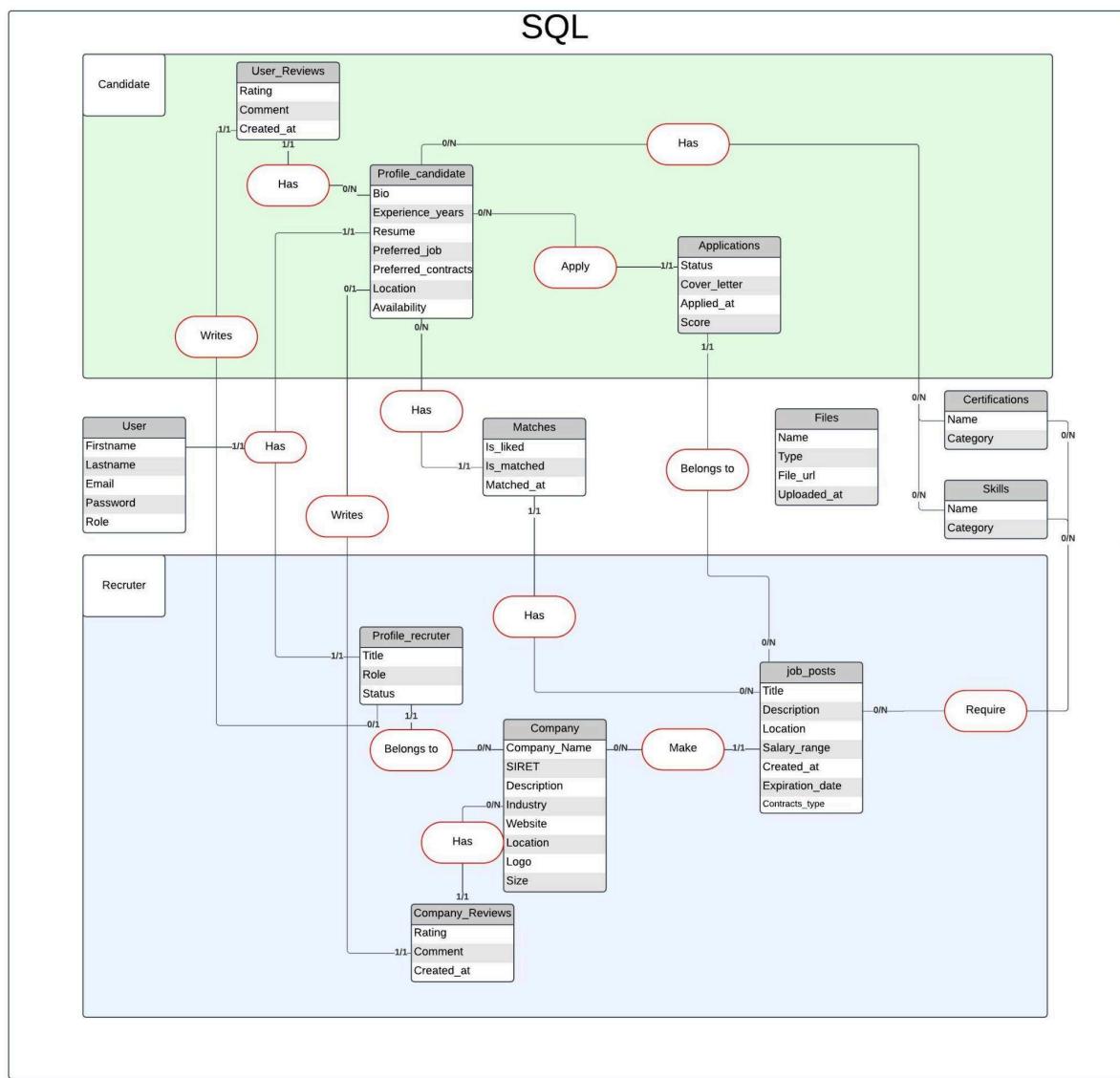
Pour structurer ces données, la méthode MERISE a été utilisée, avec une progression en 3 étapes :

- Modèle Conceptuel de Données (MCD) - représentation métier
- Modèle Logique de Données (MLD) - structuration des entités et relations

- Modèle Physique de Données (MPD) - types de données et clés

1. Modèle Conceptuel de Données (MCD)

Le MCD identifie les entités principales du système et les relations entre elles. Voici les plus importantes :



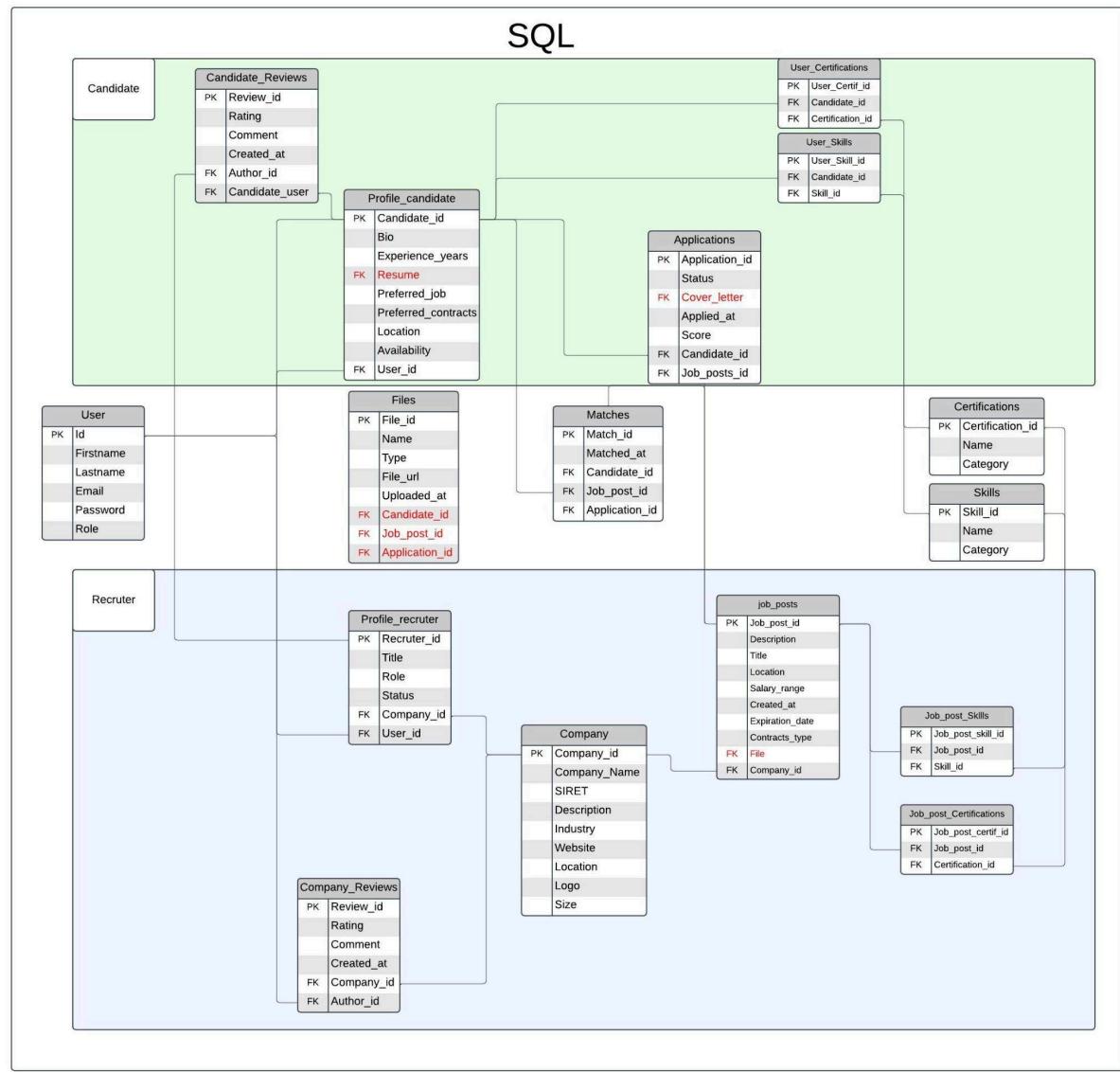
Les Relations sont :

- Un recruteur appartient à un entreprise
- Un candidat peut postuler à plusieurs offres
- Une offre appartient à une entreprise
- Une candidature est liée à un candidat et une offre

- Les messages sont liée à des utilisateurs et à des "rooms"
- Les candidats peuvent avoir plusieurs compétences et certifications (relations N/N)

2. Modèle Logique de Données (MLD)

Le MLD traduit le MCD en une structure plus formelle et proche de l'implémentation SQL. Il définit les tables, clés primaires (PK), clés étrangères (FK) et la structure des relations entre les entités.



A ce stade, les cardinalités explicites du MCD ne sont plus représentées graphiquement. Elles sont intégrées logiquement dans la structure des relations, notamment via :

- Les clés étrangères
- Les tables de liaison pour gérer les relations N/N

La base SQL a été conçue pour refléter les besoins fonctionnels suivants :

- Gestion des profils utilisateurs (candidats et recruteurs)
- Publication et gestion d'offres d'emploi
- Matching automatique entre profils et offres
- Suivi des candidatures et stockage de fichiers
- Ajout de compétences et certifications
- Evaluation des entreprises et des candidats

Structure de la base:

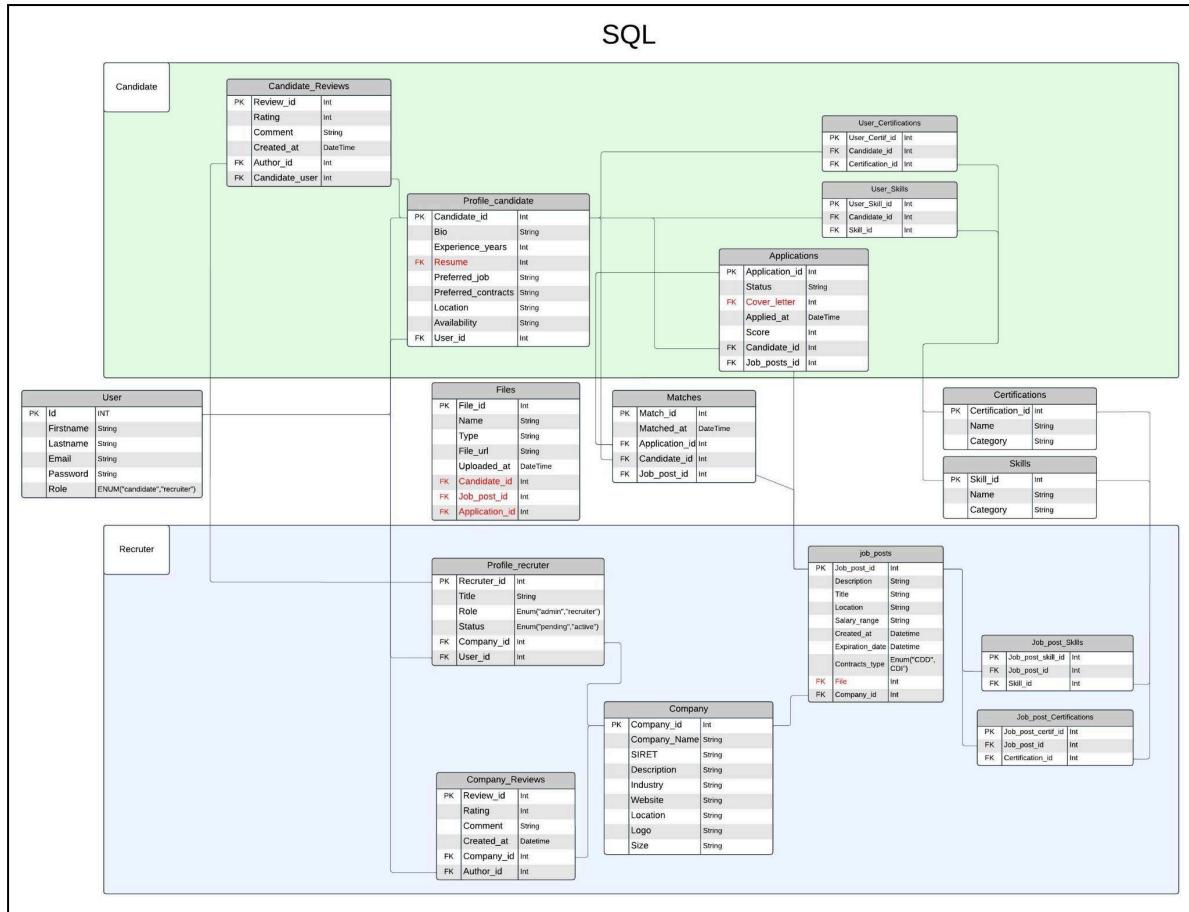
- Le schéma distingue trois grands domaines :
 - Les entités liées au candidat (en vert)
 - Les entités liées au recruteur et à l'entreprise (en bleu)
 - Les entités communes ou transverses (en blanc)
- Chaque entité métier a sa propre table principale, avec des clés étrangères bien définies
- Des tables de liaisons assurent les relations N/N entre :
 - Candidats et compétences (User_Skills)
 - Candidats et certifications (User_Certifications)
 - Offres et compétences (Job_post_Skills)
 - Offres et certifications (Job_post_Certifications)

3. Modèle Physique de Données (MPD)

Le Modèle Physique de Données (MPD) traduit le MLD en un schéma technique prêt à être implémenté dans un **Système de Gestion de Base de Données** relationnelle (ex: MySQL, PostgreSQL).

Il précise :

- Les types de données (VARCHAR, INT, DATE ...)
- Les contraintes d'intégrité (NOT NULL, UNIQUE, DEFAULT ...)
- Les clés primaires et étrangères
- Et les relations entre les tables



Cette étape permet de garantir la cohérence, la fiabilité et la performance de la base de l'environnement d'exécution réel. Ce modèle est prêt à être transformé en script SQL de création de base de données.

Développement du backend de l'application

Organisation générale

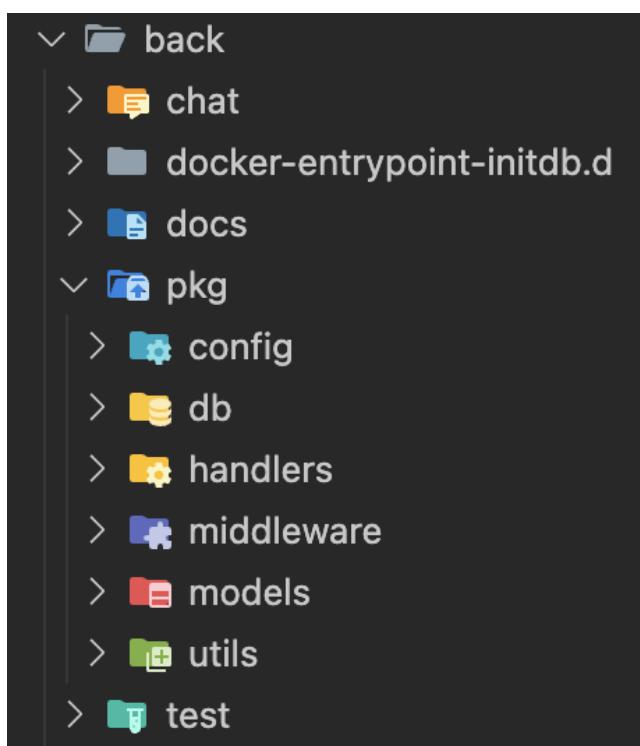
Le backend de l'application est développé en Go (Golang) et suit une architecture modulaire, facilitant la maintenance, l'évolutivité et la clarté du code. Le projet est organisé en plusieurs dossiers principaux, chacun ayant une responsabilité bien définie (gestion des routes, logique métier, accès aux données, etc.).

Arborescence du projet

L'arborescence du dossier back/ est la suivante :

- chat/ : Gestion du module de chat (messages, rooms, etc.)
- pkg/ : Contient la majorité de la logique métier, organisée par domaine (authentification, utilisateurs, offres d'emploi, etc.)
- test/ : Tests unitaires et d'intégration
- docs/ : Documentation (Swagger)
- main.go : Point d'entrée de l'application

Exemple de l'arborescence :



Fonctionnement de l'API

L'API expose des endpoints REST permettant aux clients (front-end, applications mobiles) d'interagir avec le système. Chaque ressource (utilisateur, offre d'emploi, message, etc.) dispose de ses propres routes, contrôleurs, services et modèles.

Le backend utilise des middlewares pour la gestion de l'authentification et la gestion des rôles.

Middleware

Les middlewares sont des fonctions intermédiaires exécutées avant d'atteindre les contrôleurs. Ils servent notamment à :

- Vérifier l'authentification d'un user via JWT
- Vérifier les rôles (admin, recruteur, candidat)

Exemple :

`pkg/middleware/auth.go` gère la vérification du token d'authentification.

Routage

Le routage est centralisé dans le fichier `pkg/handlers/routes.go`.

Chaque domaine (auth, utilisateur, offre, etc.) a ses propres routes, regroupées par contrôleur.

Exemple :

```
au := r.Group("/auth")
au.POST("/login", LoginHandler)
au.POST("/signup/candidate", RegisterCandidateHandler)
au.POST("/signup/recruiter", RegisterRecruiterHandler)
au.GET("/me", authMiddleware.AuthMiddleware(), GetCurrentUserHandler)
```

Contrôleurs (Controllers)

Les contrôleurs reçoivent les requêtes HTTP, extraient les données nécessaires, appellent les services appropriés et renvoient la réponse au client.

Exemple :

`pkg/handlers/user/controller.go` gère les opérations CRUD sur les utilisateurs.

Services

Les services contiennent la logique métier. Ils orchestrent les opérations complexes, appellent les repositories pour accéder aux données, et appliquent les règles de gestion.

Exemple :

`pkg/handlers/user/service.go` contient les fonctions de création, modification, suppression d'un utilisateur.

Modèles (Models)

Les modèles représentent les structures de données manipulées par l'application (utilisateur, offre, message, etc.).

Ils sont définis dans `pkg/models/` et servent à la fois pour la validation, la sérialisation et l'accès à la base de données.

Exemple :

```
16 type User struct {
17     ID      uint        `json:"id" gorm:"primaryKey"`
18     FirstName string     `json:"first_name"`
19     LastName  string     `json:"last_name"`
20     Email    string     `json:"email"`
21     Password string     `json:"--"`
22     Role     utils.RoleType `json:"role"`
23     CreatedAt time.Time   `json:"created_at"`
24     UpdatedAt time.Time   `json:"updated_at"`
25
26     ProfileCandidate *ProfileCandidate `json:"profile_candidate" gorm:"foreignKey:UserID;references:ID"`
27     ProfileRecruiter  *ProfileRecruiter `json:"profile_recruiter" gorm:"foreignKey:UserID;references:ID"`
28 }
```

Sécurité

La sécurité de SKILLLY a été pensée dès la conception, afin de garantir la confidentialité, l'intégrité et la fiabilité des données échangées entre les utilisateurs et l'API.

Plusieurs mesures ont été mises en place à différents niveaux de l'architecture :

Authentification & gestion des sessions

L'authentification repose sur l'utilisation de **JWT (Json Web Tokens)**. Lorsqu'un utilisateur se connecte, un token chiffré lui est délivré et doit être transmis dans le header `Authorization` de chaque requête vers une route protégée.

Un JWT est un jeton sécurisé, signé numériquement, contenant des informations utiles comme l'identifiant de l'utilisateur, son rôle, et une date d'expiration. Contrairement à une session classique, il permet une authentification sans état (stateless) et s'adapte parfaitement aux architectures SPA/mobile/API modernes.

Grâce à ce système :

- le serveur peut vérifier l'identité de l'utilisateur à chaque requête,
- sans avoir à maintenir de session en base,
- tout en pouvant invalider un token expiré ou altéré.

Contrôle des rôles et autorisations

Certaines routes sont accessibles uniquement selon le rôle utilisateur (admin, recruteur, candidat).

Des middlewares vérifient systématiquement le rôle contenu dans le token JWT avant d'autoriser l'accès à la ressource. Cela permet d'éviter toute élévation de privilège, en conformité avec les recommandations OWASP A01 - Broken Access Control.

Validation des données en entrée

Toutes les données utilisateur sont validées en amont via des DTO (Data Transfer Objects) et des règles strictes:

- types, longueurs, formats
- champs obligatoires
- valeurs autorisées (enums)

Cela permet de prévenir les **attaques par injection (OWASP A03 - Injection)** et les entrées malveillantes, tout en améliorant la stabilité de l'API.

Chiffrement des mots de passe

Les mots de passe utilisateurs ne sont jamais stockés en clair.

Ils sont systématiquement hachés avec un algorithme de type bcrypt, incluant un sel, avant d'être persistées en base de données. Cette méthode permet de se prémunir contre les attaques par force brute ou par vol de base de données (OWASP A02 - Cryptographic Failures).

Gestions des erreurs

La gestion des erreurs dans une API est un élément central de la robustesse et de la fiabilité d'une application. Elle permet non seulement de signaler clairement les dysfonctionnements au client, mais aussi de faciliter le débogage, d'améliorer l'expérience utilisateur, et garantir un comportement cohérent dans toutes les situations.

Dans SKILLLY, nous avons mis en place une gestion des erreurs structurée et centralisée au niveau du backend en Go (Gin). Cette approche s'appuie sur :

- L'utilisation rigoureuse des codes HTTP standardisés
- Une structure JSON unifiée pour les réponses d'erreurs
- Des logs détaillés pour le suivi côté serveur
- Des validations en amont des données utilisateur pour prévenir les erreurs en profondeur

1) Codes HTTP utilisés

Nous avons utilisé les principaux codes HTTP REST pour indiquer l'état des requêtes :

- 200 OK

Utilisé pour indiquer qu'une requête s'est déroulée avec succès

- 201 Created

Utilisé lorsqu'une requête de création de ressources s'est bien déroulée

- 204 No content

Utilisé lors de la réussite d'une requête dont la réponse ne contient pas de ressource (comme la suppression d'une ressource)

- 400 Bad Request

Utilisé lorsque les données envoyées sont invalides, manquantes ou mal formatées.

- 401 Unauthorized

Utilisé lorsque l'utilisateur n'est pas authentifié ou que le token est manquant/incorrect.

- 403 Forbidden

Utilisé lorsque l'utilisateur authentifié n'a pas les droits nécessaires (mauvais rôle)

- 404 Not Found

Utilisé lorsque la ressource demandée n'existe pas.

- 409 Conflict

Utilisé lors d'un conflit (exemple mail déjà utilisé lors de la création de compte)

- 500 Internal Server Error

Utilisé pour les erreurs inattendues côté serveur (ex: problème lors de l'accès à la base de données, transaction échouée).

2) Structure des réponses d'erreur

Toutes les erreurs renvoyées au client suivent une **structure unifiée** en JSON. Cela facilite l'affichage côté frontend et l'interprétation par les développeurs.

```
{  
  "error": "Message d'erreur explicite"  
}
```



```
1 c.JSON(200, jobPost)
```



```
1 c.JSON(201, match)
```



```
1 c.JSON(400, gin.H{"error":  
  :  
  "Invalid application ID"}  
)
```



```
1 c.JSON(401, gin.H{"error":  
  : "Unauthorized"})
```



```
1 c.JSON(403, gin.H{"error": "Forbidden"})
```

```
1 c.JSON(404, gin.H{"error":  
    "Job post not found"})
```



```
1 c.JSON(500, gin.H{"error":  
    "Failed to update application state: " +  
    err.Error()})
```

3) Validation des données (avant erreur)

Nous avons mis en place des DTO (Data Transfer Objects) qui effectuent une validation stricte des entrées utilisateur avant que la logique métier ne soit déclenchée :

- Types de données (ex: email, string, date)
- Contraintes (longueur minimale, regex, champ requis)
- Valeurs autorisées (enum pour les rôles, types de contrat, etc ...)

Cette validation précoce permet :

- D'éviter des erreurs profondes (ex: injection SQL)
- D'envoyer des messages clairs à l'utilisateur
- De limiter les traitements inutiles sur des données corrompues

Grâce à cette gestion des erreurs :

- L'API est plus prévisible pour le frontend
- Les erreurs sont cohérentes et compréhensibles
- Les utilisateurs reçoivent des retours explicites et contextualisés
- Le backend est plus stable et plus facile à maintenir

Documentation de l'API : SWAGGER

Pour faciliter l'utilisation et la compréhension de l'API de SKILLY, nous avons mis en place une documentation automatique et interactive via Swagger UI, générée à l'aide de la librairie Swaggo.

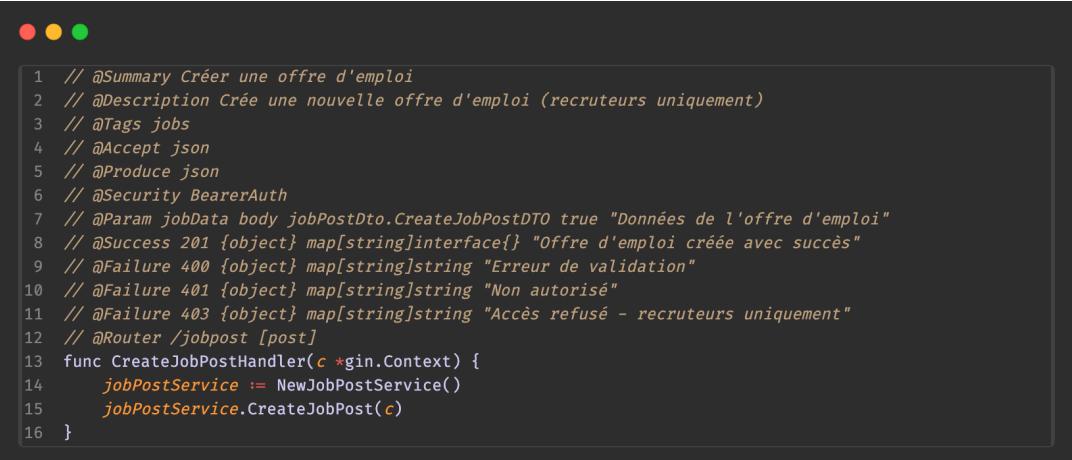
Cela permet à toute l'équipe (dev front, back, testeurs) de consulter, tester et comprendre les endpoints disponibles rapidement.

Mise en place technique

Nous avons intégré **Swaggo** dans notre projet Go avec les outils suivants :

- github.com/swaggo/swag : pour générer les fichiers à partir des annotations
- github.com/swaggo/gin-swagger : pour intégrer Swagger UI dans notre serveur Gin

Chaque route de l'API est annotée dans le code Go à l'aide de commentaires standards Swagger, par exemple:



```
● ● ●
1 // @Summary Crée une offre d'emploi
2 // @Description Crée une nouvelle offre d'emploi (recruteurs uniquement)
3 // @Tags jobs
4 // @Accept json
5 // @Produce json
6 // @Security BearerAuth
7 // @Param jobData body JobPostDTO true "Données de l'offre d'emploi"
8 // @Success 201 {object} map[string]interface{} "Offre d'emploi créée avec succès"
9 // @Failure 400 {object} map[string]string "Erreur de validation"
10 // @Failure 401 {object} map[string]string "Non autorisé"
11 // @Failure 403 {object} map[string]string "Accès refusé - recruteurs uniquement"
12 // @Router /jobpost [post]
13 func CreateJobPostHandler(c *gin.Context) {
14     jobPostService := NewJobPostService()
15     jobPostService.CreateJobPost(c)
16 }
```

La documentation est ensuite générée automatiquement via la commande :

`swag init`. Elle est accessible depuis l'URL : <http://localhost:8080/swagger/index.html>

Fonctionnalités documentées

Extrait des routes disponibles dans Swagger :

Avantages pour le projet

- Compréhension rapide des routes disponibles
- Test des routes protégées avec JWT
- Collaboration simplifiée entre développeurs front et back
- Permet de générer une doc API à livrer au client final

Exemple de documentation d'un endpoint : création d'une offre (POST/jobpost)

L'interface Swagger permet d'interagir directement avec les routes documentées de notre API. L'exemple ci-dessus illustre le endpoint POST/jobpost , utilisé par les recruteurs pour créer une nouvelle offre d'emploi.

On montre ici les données attendues dans la requête.

Sécurité

Cette route est protégée : seuls les utilisateurs authentifiés avec un rôle de recruteur peuvent l'utiliser.

L'accès se fait via l'en-tête Authorization avec un token JWT valide.

Réponse

Si la création est réussie, l'API renvoie un code 201 Created et l'objet de l'offre nouvellement créée en JSON.

Développement du frontend

Arborescence et navigation

L'arborescence de l'application SKILLY repose sur une organisation logique des écrans, pensée en fonction des deux types d'utilisateurs (candidat/recruteur). L'ensemble de la navigation a été conçu pour offrir une expérience mobile fluide, en limitant la profondeur des parcours, tout en respectant les attentes des utilisateurs.

La navigation est structurée autour d'un menu tabulaire fixe en bas de l'écran, comportant les principales sections :

- **Accueil**
- **Mes offres** (pour les recruteurs) / **Mes candidatures** (Pour les candidats)
- **Candidatures** (pour les recruteurs) / **Offres** (pour les candidats)
- **Messagerie**
- **Profil**

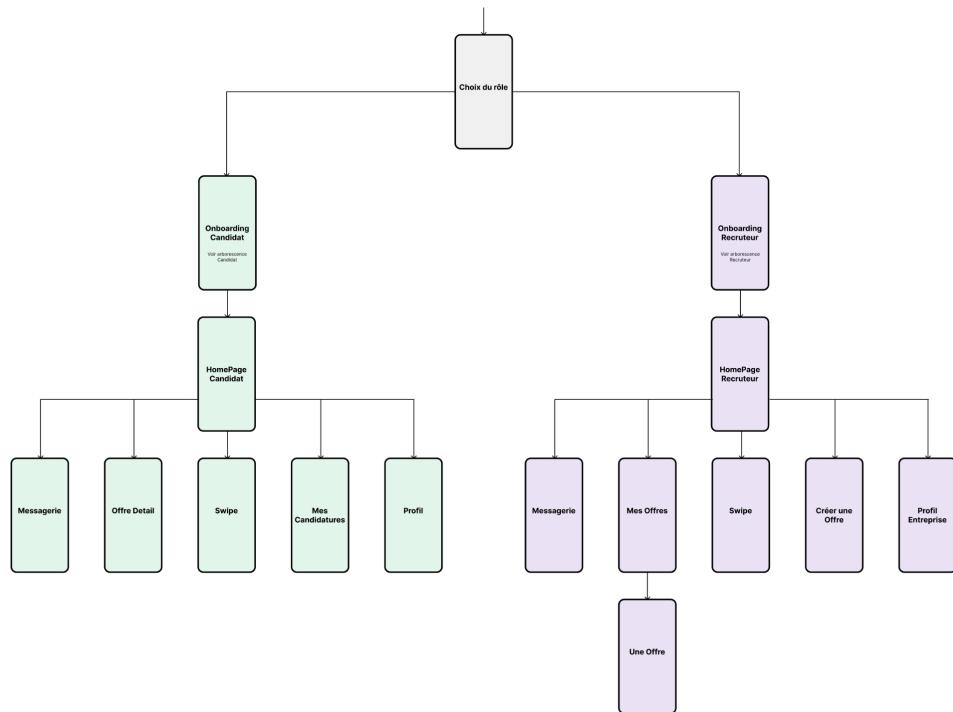
En complément, des navigations imbriquées (stack navigation) permettent d'accéder à des écrans secondaires tels que :

- la consultation d'une offre
- la création d'une annonce (recruteur)
- la lecture d'un message
- la gestion du profil utilisateur

Les technologies utilisées pour gérer cette navigation sont :

- @react-navigation/native pour la navigation globale
- @react-navigation/bottom-tabs pour la tab bar
- @react-navigation/stack pour les navigations internes à chaque onglet

Cette architecture nous permet de centraliser les routes, de personnaliser les parcours selon les rôles, et de conserver une navigation unifiée et cohérente tout au long de l'expérience.



Pages & composants

L'interface de SKILLY est composée d'écrans modulaires, construits à partir de composants réutilisables issus du design system défini en amont. Cette approche permet une cohérence visuelle, une meilleure maintenabilité, et une intégration plus rapide côté frontend.

Écrans principaux :

Rôle	Écrans clés
Candidat	Candidat Accueil, Swipe, Offres, Messagerie, Profil
Recruteur	Accueil, Création d'offre, Candidatures, Messagerie, Profil

Composants UI utilisés :

- **Card** : pour afficher les offres, profils ou candidatures
- **SwipeableCard** : pour l' interface de matching
- **MessageBubble** : messages envoyés/reçus dans le chat
- **Header & TabBar** : navigation persistante
- **Button** : CTA principaux, secondaires et désactivés
- **InputField** : formulaires de login, offre, message
- **StatusBadge** : état de candidature, statut utilisateur

Chaque composant est typé avec **TypeScript** et stylé via styled-components et **NativeWind**, ce qui facilite leur intégration, leur test et leur réutilisation dans les différents écrans.

Intégration de l'API

L'application mobile SKILLY, développée avec React Native, consomme les données via des API REST fournies par notre backend Golang.

Pour assurer une communication fiable, performante et maintenable entre le frontend et le backend, nous avons mis en place deux outils clés :

- **Axios** pour intercepter et enrichir les requêtes sortantes
- **TanStack Query** (anciennement react-query) pour orchestrer les échanges de données côté client et gérer le cache.

Sécurisation des appels avec AXIOS

Nous utilisons Axios non pas comme client HTTP principal, mais comme intercepteur global pour injecter automatiquement le token JWT dans les headers Authorization de toutes les requêtes sortantes.

Cela garantit que l'utilisateur reste authentifié et autorisé sur toutes les routes protégées, sans avoir à gérer manuellement les en-têtes dans chaque appel.

```

1 // services/api.ts      Thomas Spinec, il y a 3 mois • feat: ajout de la ge
2 import axios from "axios";
3 import AsyncStorage from "@react-native-async-storage/async-storage";
4
5 const API_BASE_URL : string | undefined = process.env.EXPO_PUBLIC_API_URL;
6 console.log(message: "API_BASE_URL", ... optionalParams: API_BASE_URL);
7 const instance : AxiosInstance = axios.create(config: {
8   baseURL: API_BASE_URL,
9   timeout: 10000,
10 });
11
12 // Ajout automatique du token à chaque requête
13 instance.interceptors.request.use(onFulfilled: async (config) => {
14   const token : string | null = await AsyncStorage.getItem(key: "token");
15   if (token) {
16     config.headers.Authorization = `Bearer ${token}`;
17   }
18   return config;
19 });

```

Ce comportement fonctionne de manière transparente avec TanStack Query, qui se charge des requêtes métier.

Tanstack

La logique d'interaction avec l'API est centralisée dans des hooks personnalisés basés sur useQuery et useMutation.

Chaque entité (job, user, candidature, message) dispose de ses fonctions de récupération ou d'envoi de données.

The screenshot shows a Mac OS X window with a dark theme. It contains two code snippets side-by-side:

```

1 // Query pour récupérer les offres d'emploi de
2 // l'entreprise (uniquement pour les recruteurs)
3 const {
4   data: jobPosts,
5   isLoading: isLoadingJobPosts,
6   error: jobPostsError,
7   refetch: refetchJobPosts,
8 } = useQuery({
9   queryKey: ["jobPosts"],
10  queryFn: JobPostService.getCompanyJobPosts,
11  enabled: role === "recruiter",
12});

```

```

1 // Mutation pour créer une nouvelle offre d'emploi
2 // (uniquement pour les recruteurs)
3 const {
4   mutate: createJobPost,
5   isPending: isCreatingJobPost,
6   error: createJobPostError,
7 } = useMutation(
8   { mutationFn: JobPostService.createJobPost,
9     onSuccess: () => {
10       queryClient.invalidateQueries({ queryKey: ["jobPosts"] });
11     },
12   });

```

Exemple d'utilisation de TanStack pour récupérer les offres d'emplois sur une page

```

1 const { jobPosts, isLoadingJobPosts, jobPostsError, refetchJobPosts } =
2   useJobPost();

```

Avantages clés :

- Gestion automatique du cache (pas besoin de useState)
- Revalidation automatique après modification
- Affichage conditionné selon l'état (chargement, erreur, succès)

Socket

Messagerie instantanée avec WebSocket (Gorilla)

L'application SKILLY propose une messagerie en temps réel entre candidats et recruteurs, accessible uniquement après qu'un match a eu lieu. Cette fonctionnalité est essentielle pour permettre des échanges directs, rapides et sécurisés, et améliore considérablement l'expérience utilisateur.

Pour la mettre en œuvre, nous avons utilisé une connexion persistante WebSocket, gérée via la bibliothèque Gorilla WebSocket en Go, permettant une communication bidirectionnelle sans délai entre le frontend React Native et le backend.

Architecture générale

L'ensemble de la messagerie est regroupé dans un module dédié chat/, contenant :

- models/ : définitions des objets Client, Hub, Room, Message
- handlers/ : gestionnaires d'entrée des connexions
- broadcast/ : logique de diffusion globale
- db/ : sauvegarde des messages en base Mongo DB
- [chat.go](#) : logique centrale de gestion des connexions et échanges

```
back/chat/
├── models/          # Modèles de données (Client, Hub, Room, Message)
├── handlers/        # Gestionnaires des routes WebSocket
├── broadcast/       # Système de diffusion globale
└── db/              # Configuration de la base de données MongoDB
└── global.go         # Gestion des connexions WebSocket globales
```

Le système repose sur trois entités principales :

- Hub : gestionnaire global des connexions, des salles et des diffusions

```

type Hub struct {
    Clients    map[string]*Client      // Clients connectés
    Rooms     map[string]*Room        // Salles de chat
    Unregister chan *Client          // Canal pour déconnecter
    Register   chan *Client          // Canal pour connecter
    Broadcast  chan Message         // Canal pour diffuser
}

```

- Client : Chaque utilisateur connecté

```

type Client struct {
    Id           string            // Identifiant unique
    Hub          *Hub              // Référence au Hub
    Rooms        map[string]*Room // Salles auxquelles il appartient
    Conn         *websocket.Conn   // Connexion WebSocket
    Send         chan []byte       // Canal d'envoi de messages
    MessageService MessageService // Service pour sauvegarder les messages
}

```

- Room : espace de discussion entre deux utilisateurs

```

type Room struct {
    ID          bson.ObjectId    // Identifiant MongoDB
    Name        string           // Nom de la salle
    CreatedAt  time.Time        // Date de création
    Clients    map[*Client]bool // Clients dans cette salle
    // Canaux pour la communication
    Unregister chan *Client
    Register   chan *Client
    Broadcast  chan Message
}

```

Fonctionnement simplifié

1. Connexion WebSocket

Lorsqu'un utilisateur rejoint une salle (/ws/:roomId), une connexion WebSocket est initiée. Le client est enregistré dans le Hub, lié à une Room.

```

func ServeWs(hub *models.Hub, room string, w http.ResponseWriter, r *http.Request) {
    conn, err := upgrader.Upgrade(w, r, nil)
    client := models.NewClient(r.URL.Query().Get("id"), hub, conn, messageService)
    hub.Register ← client
}

```

2. Réception des messages

Le message est reçu, validé, puis :

- enregistré en base MongoDB
- diffusé en temps réel à tous les clients de la salle (broadcast)

3. Envoi des messages

Chaque client écoute un canal d'envoi (Send). Lorsqu'un message lui est adressé, il est transmis via WriteMessage.

4. Déconnexion et nettoyage

Les déconnexions sont automatiquement détectées : la ressource est libérée, le client est retiré de la room, et les logs sont mis à jour.

Sécurité et performances

- Authentification par token JWT en paramètre d'URL ou via headers
- Connexions gérées en goroutines pour la performance
- Timeouts configurés (lecture/écriture/ping) pour libérer les ressources inactives
- Taille maximale des messages limitée (512octets)

Stockage des messages

Chaque message reçu est automatiquement :

- Validé (json.Unmarshal)
- Transformé en DTO (CreateMessageDTO)
- Enregistré via un messageService dans MongoDB

Cela permet de garantir la persistance des conversations, et la récupération de l'historique si un utilisateur recharge la page.

Diffusion globale

Outre les messages privés en salle, le système permet aussi une diffusion globale (par exemple: notifications temps réel) à tous les utilisateurs connectés, sauf l'expéditeur.

```

// Diffusion à tous les utilisateurs sauf l'expéditeur
func BroadcastToAllUsersExcept(excludeUserID string, message []byte) {
    for userID, client := range globalConnections {
        if userID == excludeUserID {
            continue
        }
        select {
        case client.Send <- message:
            // Message envoyé avec succès
        default:
            // Buffer plein, fermeture de la connexion
            client.Conn.Close()
        }
    }
}

```

Intégration côté frontend

Sur React Native, la messagerie utilise la bibliothèque WebSocket native:

- Connexion à l'URL WebSocket de la salle
- Reception temps réel des messages
- Affichage immédiat dans le chat
- Notification locale sur un nouveau message

Les tests

Tests automatisés

Validation bout-en-bout de la plateforme SKILLLY

Dans le cadre de SKILLLY, nous avons construit plusieurs jeux d'essais pour valider le bon déroulement des parcours critiques,

Chaque jeu d'essai associe :

- des **données d'entrée contrôlées** (formulaires, tokens, paramètres)
- des **résultats attendus** (réponses JSON, statuts HTTP, effets en base)

- et une **vérification automatisée ou manuelle** selon le scénario

Cette démarche permet de garantir la **cohérence fonctionnelle**, la **résilience de l'API**, et la **conformité aux attentes utilisateurs**, tout au long du développement.

Notre application s'appuie sur une stratégie de tests automatisés complète, couvrant à la fois le backend Go et le frontend React Native, afin de garantir la robustesse, la sécurité et la qualité de l'expérience utilisateur sur l'ensemble des parcours critiques.

Architecture générale

Les tests sont organisés en deux volets complémentaires :

Backend :

L'ensemble des tests unitaires et d'intégration du backend est centralisé dans le dossier test, structuré par domaine fonctionnel :

- db/ : vérification des connexions et de la structure des bases de données (Postgres, MongoDB)
- auth/ : scénarios d'inscription, connexion, gestion des droits user/, jobPost/, application/, match/, skill/, certification/, chat/ : tests CRUD et logique métier pour chaque entité
- middleware/ : validation des middlewares d'authentification et de gestion des rôles
- Le fichier central exec_test.go orchestre l'exécution de tous les tests, assurant une couverture exhaustive de chaque module.

Frontend (E2E) :

Les tests end-to-end sont regroupés dans le dossier e2e, avec des scripts Detox simulant des parcours utilisateurs réels sur l'application mobile :

- Connexion et inscription (candidat, recruteur)
- Navigation entre les écrans
- Gestion des erreurs et des cas limites
- Parcours de candidature, consultation d'offres, messagerie, etc.

Organisation des tests métiers backend

Le fichier main_test.go constitue le point d'entrée de la suite de tests automatisés du backend SKILLY. Il orchestre l'initialisation des environnements de test en préparant les bases de données Postgres et MongoDB via les fonctions SetupTestPostgres et

SetupTestMongo, puis initialise les repositories nécessaires à l'exécution des tests métier. Après l'exécution de l'ensemble des tests (m.Run()), il assure un nettoyage complet des environnements de test grâce aux fonctions de nettoyage dédiées, garantissant ainsi l'isolation, la reproductibilité et la fiabilité des campagnes de tests backend.

```
1 func TestMain(m *testing.M) {
2
3     // Setup the tests databases
4     setup.SetupTestPostgres()
5     setup.SetupTestMongo()
6
7     testUtils.InitTestRepositories()
8
9     // Run the tests
10    code := m.Run()
11
12    // Clean up
13    setup.CleanupTestPostgres()
14    setup.CleanupTestMongo()
15
16    // Exit with the test code
17    os.Exit(code)
18 }
```

Le fichier exec_test.go centralise l'exécution de l'ensemble des tests métiers du backend SKILLY. Structuré par domaine fonctionnel, il organise les différents scénarios de test via des fonctions TestXxx et des sous-tests t.Run, couvrant la base de données (Postgres/MongoDB), l'authentification, les utilisateurs, les offres d'emploi, les candidatures, les matchs, les compétences, les certifications, les middlewares de sécurité et la messagerie instantanée. Cette architecture modulaire permet de valider de façon exhaustive la cohérence, la sécurité et la robustesse de chaque composant métier, tout en facilitant l'évolution et la maintenance de la plateforme.

```
1
2 func TestDB(t *testing.T) {
3     t.Run("PostgresDatabaseConnection", db_test.PostgresDatabaseConnection)
4     t.Run("MongoDatabaseConnection", db_test.MongoDatabaseConnection)
5     t.Run("PostgresTableCheck", db_test.PostgresTableCheck)
6     t.Run("MongoCollectionCheck", db_test.MongoCollectionCheck)
7
8 }
9
10 func TestAuth(t *testing.T) {
11     t.Run("RegisterCandidate", auth_test.RegisterCandidate)
12     t.Run("RegisterRecruiter", auth_test.RegisterRecruiter)
13     t.Run("Login", auth_test.Login)
14 }
15
16 func TestUser(t *testing.T) {
17     t.Run("CreateUser", user_test.CreateUser)
18     t.Run(" GetUserByEmail", user_test.GetUserByEmail)
19     t.Run("UpdateUser", user_test.UpdateUser)
20     t.Run("GetAllUsers", user_test.GetAllUsers)
21     t.Run(" GetUserById", user_test.GetUserById)
22 }
```

Test unitaire de création d'utilisateur

La fonction de test CreateUser valide le processus de création d'un nouvel utilisateur dans le backend SKILLY. Elle construit un objet CreateUserDTO avec des informations factices (email, mot de passe, prénom, nom, rôle), puis appelle la méthode de repository correspondante pour insérer l'utilisateur en base. Le test vérifie l'absence d'erreur lors de la création, l'existence de l'utilisateur retourné, la cohérence des champs saisis (email, prénom, nom) et s'assure que le mot de passe stocké est bien différent du mot de passe en clair, garantissant ainsi le hashage côté serveur. Ce test assure la conformité, la sécurité et la fiabilité du processus d'enregistrement utilisateur dans la couche de persistance.

```
1 func CreateUser(t *testing.T) {
2
3     newUser := userDto.CreateUserDTO{
4         Email:      "test@test.com",
5         Password:   "password123",
6         FirstName:  "Test",
7         LastName:   "User",
8         Role:       models.RoleRecruiter,
9     }
10    user, err := testUtils.UserRepo.CreateUser(newUser, config.DB)
11    require.NoError(t, err, "Failed to create user")
12
13    assert.NotNil(t, user, "Expected user to be created")
14
15    assert.Equal(t, newUser.Email, user.Email, "Expected user email to match")
16    assert.Equal(t, newUser.FirstName, user.FirstName, "Expected user first name to match")
17    assert.Equal(t, newUser.LastName, user.LastName, "Expected user last name to match")
18    assert.NotEqual(t, newUser.Password, user.Password, "Expected user password to be hashed")
19 }
```

Test d'intégration du service d'inscription candidat

La fonction de test RegisterCandidate valide le parcours complet d'inscription d'un candidat via l'API d'authentification du backend SKILLLY. Elle simule une requête HTTP POST sur l'endpoint /auth/register/candidate en injectant les données d'un candidat de test, puis exécute la méthode de service correspondante dans un contexte Gin simulé. Le test vérifie le code de réponse HTTP, la présence des champs essentiels (user et token) dans la réponse, ainsi que la conformité des données utilisateur retournées (email, prénom, nom, rôle). Enfin, il s'assure que le token JWT est bien généré. Ce test d'intégration garantit la cohérence du flux d'inscription, l'interopérabilité entre les couches HTTP, service et modèle, et la conformité de la réponse attendue côté client.

```
1 func RegisterCandidate(t *testing.T) {
2
3     jsonData, err := json.Marshal(testUtils.TestCandidate)
4     require.NoError(t, err)
5
6     // Create HTTP request
7     req, err := http.NewRequest("POST", "/auth/register/candidate", bytes.NewBuffer(jsonData))
8     require.NoError(t, err)
9     req.Header.Set("Content-Type", "application/json")
10
11    w := httptest.NewRecorder()
12
13    // Create Gin context
14    c, _ := gin.CreateTestContext(w)
15    c.Request = req
16
17    // Call the service method
18    authService.RegisterCandidate(c)
19
20    // Assert the response status code
21    assert.Equal(t, http.StatusOK, w.Code)
22
23    var response map[string]interface{}
24    err = json.Unmarshal(w.Body.Bytes(), &response)
25    require.NoError(t, err)
26
27    // Verify response contains user and token
28    assert.Contains(t, response, "user")
29    assert.Contains(t, response, "token")
30
31    // Verify user data
32    user := response["user"].(map[string]interface{})
33    assert.Equal(t, testUtils.TestCandidate.Email, user["email"])
34    assert.Equal(t, testUtils.TestCandidate.FirstName, user["first_name"])
35    assert.Equal(t, testUtils.TestCandidate.LastName, user["last_name"])
36    assert.Equal(t, string(models.RoleCandidate), user["role"])
37
38    token := response["token"].(string)
39    assert.NotEmpty(t, token)
40 }
```

Visuel de l'exécution des tests backend

```
89 === RUN  TestCertification
90 === RUN  TestCertification/CreateCertification
91 === RUN  TestCertification/GetCertificationById
92 === RUN  TestCertification/UpdateCertification
93 --- PASS: TestCertification (0.00s)
94     --- PASS: TestCertification/CreateCertification (0.00s)
95     --- PASS: TestCertification/GetCertificationById (0.00s)
96     --- PASS: TestCertification/UpdateCertification (0.00s)
97 === RUN  TestMiddlewares
98 === RUN  TestMiddlewares/AuthMiddleware
99 [GIN] 2025/07/17 - 09:30:40 | 200 |      52.409µs |           | GET    "/test"
100 === RUN  TestMiddlewares/AuthMiddlewareUnauthorized
101 [GIN] 2025/07/17 - 09:30:40 | 401 |      3.798µs |           | GET    "/test"
102 === RUN  TestMiddlewares/RoleMiddleware
103 [GIN] 2025/07/17 - 09:30:40 | 200 |      34.715µs |           | GET    "/test"
104 === RUN  TestMiddlewares/RoleMiddlewareForbidden
105 [GIN] 2025/07/17 - 09:30:40 | 403 |      5.029µs |           | GET    "/test"
106 --- PASS: TestMiddlewares (0.00s)
107     --- PASS: TestMiddlewares/AuthMiddleware (0.00s)
108     --- PASS: TestMiddlewares/AuthMiddlewareUnauthorized (0.00s)
109     --- PASS: TestMiddlewares/RoleMiddleware (0.00s)
110     --- PASS: TestMiddlewares/RoleMiddlewareForbidden (0.00s)
111 === RUN  TestChat
112 === RUN  TestChat/CreateRoom
113 Inserting entity into collection: room
114 === RUN  TestChat/GetAllRooms
115 === RUN  TestChat/CreateMessage
116 Creating message: {ObjectID("000000000000000000000000") test_room Hello, World! 2025-07-17 09:30:40.766666624 +0000 UTC m=+0.384194376}
117 Inserting entity into collection: message
118 === RUN  TestChat/GetAllMessages
119 --- PASS: TestChat (0.00s)
120     --- PASS: TestChat/CreateRoom (0.00s)
121     --- PASS: TestChat/GetAllRooms (0.00s)
122     --- PASS: TestChat/CreateMessage (0.00s)
123     --- PASS: TestChat/GetAllMessages (0.00s)
```

Organisation des tests end-to-end avec Detox

le fichier [login.test.js](#) orchestre les différents cas de figure de connexion :

- **Login fail Tests :**

Vérification des cas d'échec (identifiants manquants, invalides, etc.), avec assertion sur l'affichage des messages d'erreur appropriés.

- **Login success Tests :**

Validation des connexions réussies pour les différents rôles (recruteur, candidat), avec vérification de la redirection et de l'affichage du message d'accueil personnalisé.

Fonctionnement

- **Initialisation**

Chaque test démarre sur l'écran d'accueil, attend que le bouton de connexion soit visible, puis navigue vers l'écran de login.

- **Saisie et soumission**

Les champs email et mot de passe sont remplis automatiquement selon le scénario testé. La soumission du formulaire est simulée, en prenant soin de gérer le clavier virtuel et le focus pour garantir la fiabilité sur Android et iOS.

- **Vérification des résultats**

- En cas d'échec, le test vérifie la présence et le contenu du message d'erreur (loginError).
- En cas de succès, il attend l'affichage du message d'accueil (homeGreeting) sur l'écran principal post-authentification.

- **Gestion de l'état**

Après chaque test, l'état de l'application est réinitialisé :

- Les champs de connexion sont vidés après chaque test d'échec.
- Après chaque test de succès, une déconnexion automatique est effectuée via le menu profil, garantissant l'indépendance des tests.



```
1  describe("Login fail Tests", () => {
2      afterEach(async () => {
3          await ClearLoginFields();
4      });
5
6      failTests.forEach(({ title, email, password, contains }) => {
7          it(title, async () => {
8              await Login(email, password);
9              if (contains) {
10                  await expect(element(by.id("loginError"))).toBeVisible();
11                  await expect(element(by.id("loginError"))).toHaveText(contains);
12              }
13          });
14      });
15  });
```



```

1  describe("Login success Tests", () => {
2      afterEach(async () => {
3          await LogOut();
4          await waitFor(element(by.id("loginButton")))
5              .toBeVisible()
6              .withTimeout(10000);
7          await element(by.id("loginButton")).tap();
8          await waitFor(element(by.id("emailInput")))
9              .toBeVisible()
10             .withTimeout(10000);
11      });
12
13      successTests.forEach(({ title, email, password, contains }) => {
14          it(title, async () => {
15              await Login(email, password);
16              if (contains) {
17                  await expect(element(by.id("homeGreeting"))).toBeVisible();
18                  await expect(element(by.id("homeGreeting"))).toHaveText(contains);
19              }
20          });
21      });
22  });

```

Visuel de l'exécution des tests e2e frontend

```

15:54:16.089 detox[43195] i Login: should display the login button
15:54:16.122 detox[43195] i Login: should display the login button [OK]
15:54:16.123 detox[43195] i Login: should display the login screen
15:54:17.933 detox[43195] i Login: should display the login screen [OK]
15:54:17.933 detox[43195] i Login > Login fail Tests: Missing email
15:54:26.562 detox[43195] i Login > Login fail Tests: Missing email [OK]
15:54:26.564 detox[43195] i Login > Login fail Tests: Invalid email
15:54:36.326 detox[43195] i Login > Login fail Tests: Invalid email [OK]
15:54:36.328 detox[43195] i Login > Login fail Tests: Invalid password
15:54:45.262 detox[43195] i Login > Login fail Tests: Invalid password [OK]
15:54:45.266 detox[43195] i Login > Login success Tests: Login successful recruteur
15:54:59.352 detox[43195] i Login > Login success Tests: Login successful recruteur [OK]
15:54:59.353 detox[43195] i Login > Login success Tests: Login successful candidat
15:55:13.549 detox[43195] i Login > Login success Tests: Login successful candidat [OK]

```

Cahiers de recette

Backend – Tests unitaires et d'intégration

1) Initialisation et nettoyage

Test	Objectif	Pré-requis / Données d'entrée	Résultat attendu
Setup/Cleanup des bases de test	Préparer et nettoyer les bases Postgres et MongoDB pour garantir l'isolation des tests	—	Les bases sont prêtes avant/après tests

2) Connexion aux bases de données

Test	Objectif	Jeux d'essai	Résultat attendu
Postgres Database Connection	Vérifier la connexion à la base Postgres	—	Connexion établie sans erreur
MongoDB Database Connection	Vérifier la connexion MongoDB	—	Connexion établie sans erreur
Postgres Table Check	Vérifier la présence des tables attendues dans Postgres	—	Tables présentes
Mongo Collection Check	Vérifier la présence des collections attendues dans MongoDB	—	Collections présentes

3) Authentification

Test	Objectif	Pré-requis / Données d'entrée	Résultat attendu
RegisterRecruiter	Tester l'inscription d'un recruteur	Données recruteur de test	200 OK, user et token dans la réponse
RegisterCandidate (service)	Tester l'inscription d'un candidat via l'API HTTP	Données candidat de test	200 OK, user et token dans la réponse
Login	Tester la connexion avec des identifiants valides/invalides	Email/mot de passe	Succès ou message d'erreur approprié

4) Utilisateurs

Test	Objectif	Pré-requis / Données d'entrée	Résultat attendu
CreateUser	Créer un utilisateur et vérifier le hashage du mot de passe	Données utilisateur	Utilisateur créé, mot de passe hashé
GetUserByEmail	Récupérer un utilisateur par email	Email	Utilisateur correspondant
UpdateUser	Modifier les infos d'un utilisateur	Données modifiées	Infos mises à jour
GetAllUsers	Lister tous les utilisateurs		Liste complète
GetUserById	Récupérer un utilisateur par ID	ID utilisateur	Utilisateur correspondant
DeleteUser	Suppression d'un utilisateur par ID	ID utilisateur	suppression de l'utilisateur correspondant

5) Offres d'emploi

Test	Objectif	Pré-requis / Données d'entrée	Résultat attendu
CreateJobPost	Créer une offre d'emploi	Données offre	Offre créée
GetJobPost	Récupérer une offre par ID	ID offre	Offre correspondante
GetAllJobPost	Liste de toutes les offres		Liste complète
UpdateJobPost	Modifier une offre d'emploi	Données modifiées	Offre mise à jour
DeleteJobPost	Suppression d'une offre par ID	ID offre	Suppression de l'offre correspondante

Frontend – Tests end-to-end (Detox)

Parcours de connexion

Test	Objectif	Jeux d'essai	Résultat attendu
Affichage écran d'accueil	Vérifier la présence du bouton de connexion	—	Bouton visible
Affichage écran de login	Vérifier la navigation vers l'écran de connexion	—	Champs email/mot de passe visibles
Login fail (plusieurs cas)	Tester les erreurs de connexion (email manquant, invalide, mauvais mot de passe)	Différents jeux d'essai	Message d'erreur affiché
Login success (recruteur/candidat)	Tester la connexion réussie pour chaque rôle	Identifiants valide	Message d'accueil affiché
LogOut	Tester la déconnexion après un login réussi	—	Retour à l'écran de login

Déploiement et Démarche DevOps

Le déploiement d'une application sécurisée et performante nécessite une méthodologie rigoureuse et des outils adaptés pour garantir une mise en production fluide, reproductible et sécurisée. Dans le cadre de notre projet, nous avons mis en place une architecture conteneurisée avec Docker, et préparé une intégration continue (CI) via GitHub Actions pour automatiser les tests et vérifications à chaque modification du code.

Bien que la mise en production automatisée (CD) ne soit pas en place actuellement, nous avons préparé l'architecture pour pouvoir l'implémenter facilement via un serveur distant et GitHub Actions.

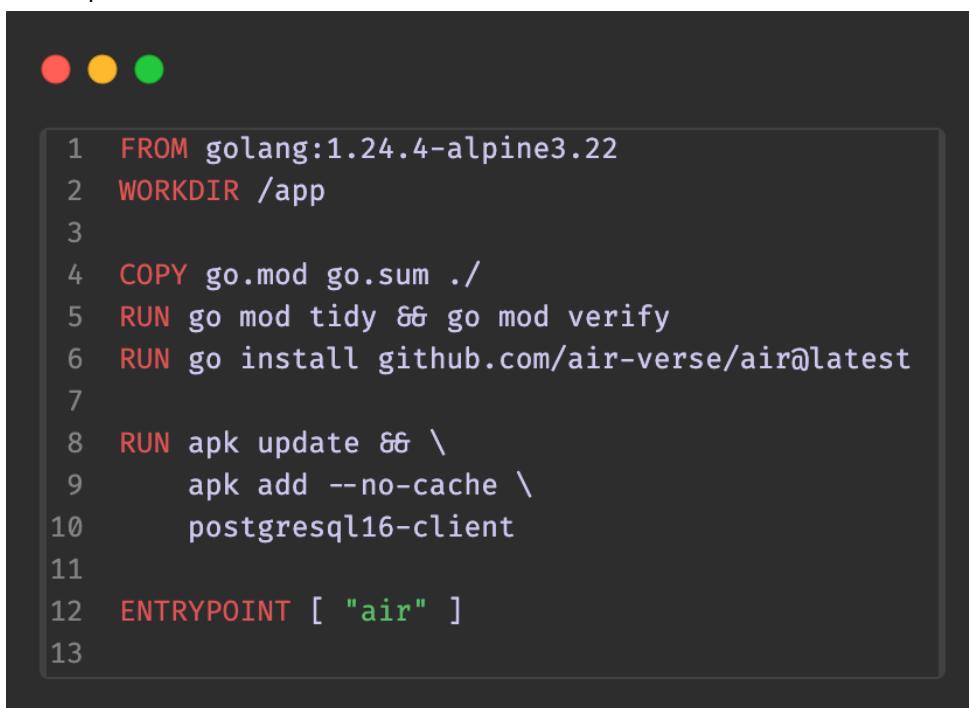
Conteneurisation avec Docker

Nous avons utilisé Docker pour conteneuriser le backend de l'application, développé en Go.

Cela permet de :

- Garantir un environnement homogène entre tous les développeurs
- Faciliter les tests en local et les futurs déploiements
- Réduire les erreurs liées aux différences de configuration
- Simuler un environnement de production réaliste

Exemple de fichier Dockerfile :



```
1 FROM golang:1.24.4-alpine3.22
2 WORKDIR /app
3
4 COPY go.mod go.sum ./
5 RUN go mod tidy && go mod verify
6 RUN go install github.com/air-verse/air@latest
7
8 RUN apk update && \
9     apk add --no-cache \
10    postgresql16-client
11
12 ENTRYPOINT [ "air" ]
13
```

Docker Compose (multi-conteneurs)

Pour orchestrer les services nécessaires à l'application (API + Base de données PostgreSQL), nous avons utilisé Docker Compose :

```
1 services:
2   back:
3     container_name: Server
4     build:
5       context: .
6       dockerfile: Dockerfile.dev
7     ports:
8       - "8080:8080"
9     volumes:
10    - .:/app
11   environment:
12     - NODE_ENV=development
13   depends_on:
14     - postgres
15     - mongodb
```

```
1 postgres:
2   image: postgres:16.4
3   container_name: skillly_db
4   environment:
5     POSTGRES_USER: ${DB_USER}
6     POSTGRES_PASSWORD: ${DB_PASSWORD}
7     POSTGRES_DB: ${DB_NAME}
8   ports:
9     - "3308:5432"
10  volumes:
11    - db_data:/var/lib/postgresql/data
```

```
1 mongodb:
2   image: mongo:latest
3   container_name: skillly_mongo
4   environment:
5     MONGO_INITDB_ROOT_USERNAME: ${DB_USER}
6     MONGO_INITDB_ROOT_PASSWORD: ${DB_PASSWORD}
7     MONGO_INITDB_DATABASE: ${DB_NAME}
8   ports:
9     - "27018:27017"
10  volumes:
11    -
12      ./docker-entrypoint-initdb.d/:/docker-entrypoint-initdb.d/:ro
13      - db_data:/mongo-data/db
```

Variables d'environnement

Les variables sensibles sont centralisées dans un fichier .env (non versionné) afin de garantir la **sécurité** et la **portabilité**. Ces variables sont injectées dans les services Docker et l'application Go via `os.Getenv()`.

Intégration Continue (CI) avec GitHub Actions

Une pipeline CI a été configurée pour:

- Lancer les **tests unitaires**
- Vérifier le bon **build Docker**
- Préparer les étapes d'intégration continue

Piste pour une future mise en production automatisée (CD) via GitHub Actions

Notre application est déjà structurée de manière à pouvoir intégrer une mise en production continue à l'aide de GitHub Actions couplé à un déploiement SSH.

Tous les éléments nécessaires sont en place :

- Dockerisation du backend
- Fichier `dockerfile.prod` prêts à l'emploi
- Environnement `.prod` préconfiguré

Il ne manque que la connexion à un serveur distant disposant d'URLs réelles pour sortir du contexte local.

Stratégie envisagée :

1. Un push sur main déclenche le workflow GitHub Actions
2. Connexion au serveur distant via clé SSH sécurisée
3. Exécution des commandes :
4. L'application est déployée sur une URL comme <https://api.skillly.app>.

Les **clés SSH et secrets** sont gérés via **GitHub Secrets** pour garantir la sécurité du processus.

Veille technologique & résolution de problème

Le développement d'une application moderne, comme SKILLLY, repose sur l'utilisation de nombreuses technologies en constante évolution (frameworks, bibliothèques, dépendances, outils de build, etc ...).

La veille technologique est essentielle pour :

- anticiper les changements techniques (breaking changes)
- réagir rapidement aux incidents
- maintenir la compatibilité du projet avec les dernières versions d'outils

Elle fait partie intégrante de la résolution de problèmes, notamment en environnement JavaScript / React Native où les mises à jour peuvent être fréquentes et impactantes.

Exemple de veille : incompatibilité entre react-native-deck-swiper et Expo SDK 53

Pendant le développement, nous avons effectué une mise à jour vers Expo SDK 53. Cette version récente d'Expo a introduit des changements internes dans la gestion des composants natifs et dans l'architecture du moteur.

Suite à cette mise à jour :

- La librairie `react-native-deck-swiper`, utilisée pour l'interface de swipe (matching), a cessé de fonctionner
- L'animation des cartes swipe était totalement désactivée ou provoquait un crash au moment du rendu

Démarche de résolution

1. Analyse de l'erreur

- Lecture attentive des logs via Expo CLI pour identifier les composants défectueux
- Observation de plantages systématiques dès le chargement de la vue de swipe

2. Recherche de solutions

- Consultation des notes de version de EXPO SDK 53 (release notes) et des issues GitHub
- Vérification du statut de maintenance de `react-native-deck-swiper` (aucunes mises à jour depuis plusieurs mois)

3. Tests de librairies alternatives

- Évaluation de librairies similaires :
 - `react-native-snap-carousel`
 - `react-native-reanimated-carousel`
 - Problèmes rencontrés : incompatibilités Android & iOS, comportement instable ou absence de swipe fluide

4. Décision technique

- Nous avons décidé de ne plus utiliser de librairie externe pour cette fonctionnalité.
- Nous avons recréé manuellement le comportement de swipe, en combinant :

- PanResponder pour capter le mouvement du doigt
- Animated.View pour gérer les translations et rotations
- useRef et useState pour gérer les cartes actives et transitions

5. Résultat

- Un Composant personnalisé, plus léger et plus adapté à notre besoin
- Plus de dépendance externe fragile : le swipe est maintenant sous contrôle complet de l'équipe
- Meilleure compréhension de la logique d'animation dans React Native

Bilan

Cet incident nous a permis de :

- Développer un comportement clé sur mesure
- Renforcer notre maîtrise des APIs bas niveau de React Native
- Gagner en fiabilité et indépendance technologique

Conclusion

SKILLLY a été bien plus qu'un simple exercice de développement. Il m'a permis de travailler sur un sujet concret, avec une vraie utilité, dans un cadre stimulant à la fois技iquement et humainement.

J'ai particulièrement pris plaisir à explorer des aspects techniques que je connaissais peu en début de formation, notamment le backend en Go, l'architecture d'une API sécurisée, et l'intégration des WebSockets pour la messagerie.

Comprendre comment structurer un backend robuste, comment optimiser les échanges entre le front et le back, ou encore comment organiser les couches d'une application moderne a été une vraie source de motivation au quotidien.

Ce projet m'a également permis de renforcer ma rigueur, ma capacité à collaborer, et surtout ma curiosité. La phase de recherche et de résolution de problème, notamment lors de la mise à jour d'Expo ou l'intégration de la messagerie, m'a poussé à sortir de ma zone de confort pour aller chercher des solutions concrètes, tester, recommencer, comprendre.

Je ressors de cette expérience avec le sentiment d'avoir construit quelque chose de complet, cohérent et utile. SKILLLY restera pour moi un projet de référence, mais surtout un jalon important dans ma progression vers le métier de développeur.