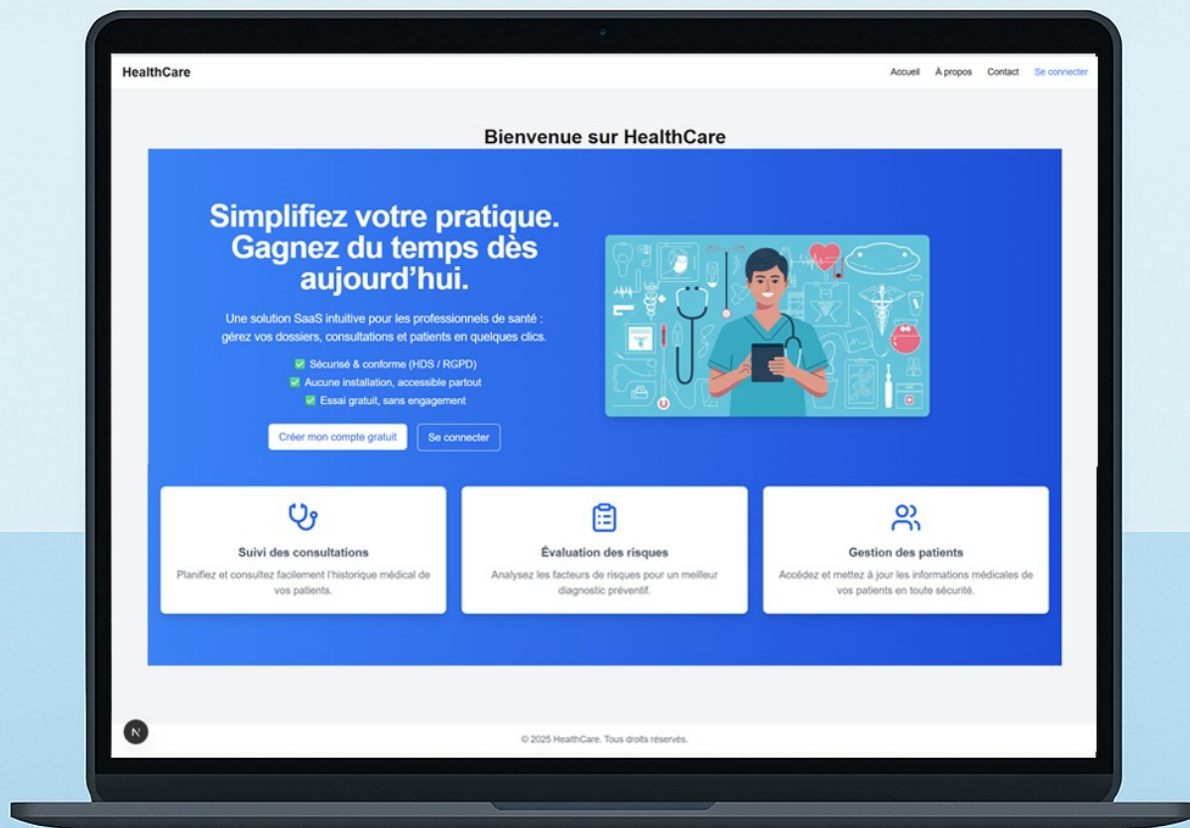


HealthCare

Dossier de Projet



Jean-Ely Gendrau

Concepteur Développeur d'Applications
RNCP(37873)

La Plateforme

Dossier de Projet CDA – HealthCare (Medica)

Sommaire

1. Présentation du projet

- 1.1 Contexte et objectifs
- 1.2 Périmètre fonctionnel
- 1.3 Public cible et cas d'usage
- 1.4 Technologies et outils utilisés
- 1.5 Rôles et responsabilités
- 1.6 Organisation du projet

2. Analyse des besoins et conception

- 2.1 Recueil des besoins
- 2.2 Analyse fonctionnelle
- 2.3 Modélisation (UML, MCD, MLD)
- 2.4 Architecture logique et technique

3. Développement de l'application

- 3.1 Frontend (interfaces et navigation)
- 3.2 Backend (microservices et logique métier)
- 3.3 API, sécurité et authentification
- 3.4 Base de données et persistance
- 3.5 Tests unitaires et fonctionnels

4. Déploiement et exploitation

- 4.1 Conteneurisation (Docker, Jib)
- 4.2 Orchestration (Docker Compose / Swarm)
- 4.3 Environnements DEV / PROD
- 4.4 Documentation technique et scripts

5. Bilan de projet

- 5.1 Résultats obtenus
- 5.2 Difficultés rencontrées et solutions
- 5.3 Perspectives d'amélioration

6. Conclusion générale

1. Présentation du projet

1.1 Contexte et objectifs

Dans le cadre de ma formation au titre professionnel de Concepteur Développeur d'Applications (CDA), j'ai réalisé un projet nommé HealthCare (Medica). J'ai choisi ce sujet car il me permettait de répondre à une problématique concrète : concevoir une plateforme modulaire capable de gérer des données de santé de manière sécurisée, évolutive et exploitable en contexte professionnel.

L'objectif principal était de mettre en œuvre mes compétences techniques en architecture logicielle, développement full-stack, sécurité, base de données et déploiement en environnement conteneurisé.

1.2 Périmètre fonctionnel

J'ai défini un périmètre réaliste qui comprend les fonctionnalités suivantes :

- Authentification sécurisée des utilisateurs avec JWT
- Gestion des patients (création, modification, consultation)
- Enregistrement de notes médicales (stockées en NoSQL)
- Calcul automatique d'un score de santé basé sur les notes
- Visualisation du score depuis une interface web ergonomique

1.3 Public cible et cas d'usage

L'application s'adresse principalement aux professionnels de santé souhaitant analyser rapidement l'état d'un patient à partir de notes textuelles. J'ai imaginé les cas d'usage suivants :

- Un médecin généraliste consulte la fiche d'un patient
- Une note est ajoutée après une visite médicale
- Le score de risque est calculé automatiquement

1.4 Technologies et outils utilisés

Pour mener à bien ce projet, j'ai utilisé les outils suivants :

- Frontend : React avec Next.js, Tailwind CSS, TypeScript
- Backend : Java 17, Spring Boot, Spring Security, Spring Cloud
- Base de données : PostgreSQL (relationnelle) et MongoDB (NoSQL)
- Authentification : JWT
- Conteneurisation : Docker, Docker Compose, Jib
- Orchestration : Docker Swarm (simulation de PROD)
- Gestion de configuration : Spring Cloud Config

1.5 Rôles et responsabilités

J'ai été responsable de l'intégralité du projet. Cela inclut :

- L'analyse du besoin et la rédaction du périmètre fonctionnel
- La conception technique de l'architecture microservices
- Le développement de tous les modules (front et back)

- La configuration des bases de données et de la sécurité
- L'écriture des scripts d'installation et d'orchestration
- La rédaction de la documentation et du présent dossier

1.6 Organisation du projet

Le projet est organisé selon une architecture microservices. Chaque domaine métier est isolé dans un service indépendant, ce qui facilite la maintenance et l'évolutivité. Voici la structure :

- | | |
|-------------------------|--|
| ➤ frontend | ➤ interface utilisateur SPA |
| ➤ module-patient | ➤ gestion des patients |
| ➤ module-note | ➤ enregistrement de notes médicales |
| ➤ module-scoring | ➤ calcul du score à partir des notes |
| ➤ module-authentication | ➤ gestion des utilisateurs et des tokens |
| ➤ module-gateway | ➤ point d'entrée unique via API Gateway |
| ➤ module-config | ➤ configuration centralisée Spring |

Ces modules sont conteneurisés, orchestrés et configurés pour fonctionner ensemble dans des environnements DEV et PROD simulés.

2. Analyse des besoins et conception

2.1 Recueil des besoins

Comme le projet a été réalisé en autonomie, j'ai moi-même défini le besoin fonctionnel.

Je me suis inspiré de situations concrètes rencontrées dans le domaine médical :

- Un professionnel de santé doit pouvoir consulter un dossier patient rapidement
- Il doit enregistrer des notes médicales après chaque visite
- Ces notes doivent être analysées pour générer un score de risque de santé

J'ai ainsi construit un cahier des charges fonctionnel simple mais réaliste, orienté autour des notions de sécurité, de modularité, et d'automatisation du scoring médical.

2.2 Analyse fonctionnelle

J'ai identifié plusieurs cas d'usage pour structurer les attentes du système :

- Se connecter via un compte sécurisé
- Accéder à la liste des patients
- Visualiser les détails d'un patient
- Ajouter une note médicale à un patient
- Générer et visualiser le score de santé à partir des notes

Ces cas d'usage ont ensuite été utilisés pour définir les routes API REST entre les microservices, et organiser le découpage fonctionnel en modules indépendants.

2.3 Modélisation (UML, MCD, MLD)

Pour structurer ma réflexion et faciliter le développement, j'ai conçu plusieurs modèles :

- Un diagramme de cas d'usage, pour représenter les interactions entre l'utilisateur et le système
- Un MCD (Modèle Conceptuel de Données), qui m'a permis de visualiser les entités Patient, Note, Score et leurs relations
- Un MLD (Modèle Logique de Données), traduit ensuite dans PostgreSQL
- Un diagramme de classes simplifié, utilisé pour organiser mes objets dans le code backend

J'ai aussi pris en compte la particularité du module `note-service`, qui utilise MongoDB. Chaque note est donc un document NoSQL stocké indépendamment, mais relié logiquement au patient par son ID.

2.4 Architecture logique et technique

J'ai fait le choix d'une architecture microservices pour isoler chaque fonctionnalité métier dans un service dédié. Cela m'a permis de :

- Travailler indépendamment sur chaque domaine (patients, notes, score...)
- Faciliter les tests, les mises à jour, et le déploiement partiel
- Expérimenter l'orchestration multi-service (Docker Compose / Swarm)

Le système repose également sur une API Gateway (Spring Cloud Gateway), qui centralise les accès à tous les services. J'ai aussi mis en place un serveur Consul

(HashiCorp) pour gérer la découverte dynamique des microservices, ainsi qu'un serveur de configuration externe via Spring Config Server.

Cette architecture assure à la fois la **modularité**, la **scalabilité**, et une **séparation claire des responsabilités**, ce qui répond aux bonnes pratiques du développement professionnel moderne.

3. Développement de l'application

3.1 Frontend (interfaces et navigation)

Pour la partie frontend, j'ai choisi le framework Next.js basé sur React. Cela m'a permis de développer une interface utilisateur en SPA (Single Page Application), tout en profitant du support SSR (Server Side Rendering) si nécessaire.

J'ai structuré l'interface avec des composants réutilisables, typés en TypeScript. J'ai utilisé Tailwind CSS pour le style, car il permet une intégration rapide et cohérente sans devoir gérer des feuilles de style séparées.

Voici les principales fonctionnalités que j'ai mises en œuvre côté frontend :

- Formulaire de connexion avec gestion du token JWT
- Tableau de bord affichant la liste des patients
- Détail d'un patient : informations personnelles, notes médicales, score
- Ajout d'une note depuis l'interface, avec appel API

L'ensemble des appels REST est centralisé via des services Axios. J'ai également intégré une logique de redirection si l'utilisateur n'est pas authentifié.

3.2 Backend (microservices et logique métier)

Chaque fonctionnalité métier est développée sous forme de microservice Spring Boot. J'ai respecté une structure propre pour chaque module, avec :

- Un contrôleur REST
- Un service métier
- Un repository (pour la base relationnelle ou MongoDB)

Les microservices développés sont les suivants :

- | | |
|-------------------------|--|
| ➤ module-patient | ➤ permet de créer, consulter, modifier ou supprimer un patient |
| ➤ module-note | ➤ enregistre et consulte les notes médicales stockées en MongoDB |
| ➤ module-scoring | ➤ reçoit les notes et renvoie un score basé sur un algorithme simple (présence de mots-clés) |
| ➤ module-authentication | ➤ gère les utilisateurs, la connexion, |

et génère un token JWT

- `module-gateway`
- filtre, redirige et sécurise tous les appels vers les microservices

J'ai également intégré des headers HTTP personnalisés, ainsi que des contrôles sur les rôles pour certaines routes sensibles.

3.3 API, sécurité et authentification

La sécurité de l'application repose sur un système JWT. Lors de la connexion, un token est généré et stocké côté client. Ce token est ensuite envoyé avec chaque requête API dans l'en-tête Authorization.

J'ai utilisé Spring Security pour :

- Protéger les routes sensibles
- Filtrer les requêtes selon le rôle de l'utilisateur
- Gérer les erreurs d'accès et les expirations de session

Toutes les API REST sont bien définies et testées manuellement via Postman.

3.4 Base de données et persistance

J'ai utilisé deux types de base de données :

- PostgreSQL pour les entités relationnelles (Patient, Utilisateur, Score)
- MongoDB pour les notes, stockées comme des documents JSON

Chaque microservice possède sa propre base de données. C'est une bonne pratique en architecture microservices, qui permet d'éviter les dépendances croisées.

J'ai configuré chaque datasource dans les fichiers de configuration (application.yml), et utilisé des repositories Spring Data pour gérer les opérations CRUD.

3.5 Tests unitaires

Afin de garantir la fiabilité et la robustesse des services métier, j'ai mis en place une batterie de tests unitaires sur le microservice `patient-service`. Les tests ont été réalisés à l'aide de **JUnit 5**, et la couverture a été mesurée avec **JaCoCo**.

L'objectif était de vérifier le comportement des méthodes critiques de la classe PatientService, notamment : createPatient, updatePatient, getClient, et getAllPatient. Ces méthodes couvrent les principales opérations métier de la gestion des patients.

Les tests ont été exécutés avec succès via Maven, comme en atteste la sortie terminale suivante :


```

[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.683 s -- in
fr.devstarting.healthcare.service.PatientServiceTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 14, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jar:3.4.2:jar (default-jar) @ healthcare ---
[INFO] Building jar: D:\WWW\Workspace-HealthCare\microservices\module-
patient\target\healthcare-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot:3.4.3:repackage (repackage) @ healthcare ---
[INFO] Replacing main artifact D:\WWW\Workspace-HealthCare\microservices\module-
patient\target\healthcare-0.0.1-SNAPSHOT.jar with repackaged archive, adding nested
dependencies in BOOT-INF/.
[INFO] The original artifact has been renamed to D:\WWW\Workspace-
HealthCare\microservices\module-patient\target\healthcare-0.0.1-SNAPSHOT.jar.original
[INFO]
[INFO] --- jacoco:0.8.12:report (report) @ healthcare ---
[INFO] Loading execution data file D:\WWW\Workspace-HealthCare\microservices\module-
patient\target\jacoco.exec
[INFO] Analyzed bundle 'patient' with 13 classes
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 15.093 s
[INFO] Finished at: 2025-09-21T17:45:20+02:00
[INFO] -----











```

Process finished with exit code 0

La couverture obtenue via JaCoCo est la suivante :

 patient >  fr.devstarting.healthcare.service >  PatientService

PatientService

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
• lambda\$getClient\$0(UUID)		0 %		n/a	1	1	2	2	1	1
• update(Patient, Patient)		95 %		87 %	2	9	0	19	0	1
• updatePatientProcessing(UUID, Patient)		100 %		n/a	0	1	0	6	0	1
• getClient(UUID)		100 %		n/a	0	1	0	4	0	1
• PatientService(PatientRepository, PatientMapper)		100 %		n/a	0	1	0	5	0	1
• lambda\$updatePatientProcessing\$2(UUID)		100 %		n/a	0	1	0	2	0	1
• createPatient(Patient)		100 %		n/a	0	1	0	3	0	1
• lambda\$updatePatientProcessing\$1(Patient, Patient)		100 %		n/a	0	1	0	1	0	1
• getAllPatient()		100 %		n/a	0	1	0	1	0	1
Total	15 of 198	92 %	2 of 16	87 %	3	17	2	42	1	9

- Couverture des instructions : **92%**
- Couverture des branches : **87%**
- Total de méthodes testées : **9 méthodes** sur 9

Ce niveau de couverture garantit que la majorité des cas métiers sont correctement vérifiés, y compris les cas limites. Les tests valident les interactions avec les dépendances telles que le *PatientRepository*, grâce à l'utilisation de mocks et d'assertions ciblées.

Ces tests permettent d'améliorer la maintenabilité du code et servent de filet de sécurité en cas d'évolution du service. Ils illustrent également ma capacité à adopter une démarche qualité dans un projet logiciel structuré.

Je prévois de compléter cette couverture avec des tests plus avancés (mocking, intégration complète) dans la suite du projet.

4. Déploiement et exploitation

4.1 Conteneurisation (Docker, Jib)

Dès la phase de développement, j'ai choisi de conteneuriser tous les composants de l'application afin d'assurer leur portabilité et leur isolation. J'ai utilisé Docker comme socle principal.

Pour les microservices en Java, j'ai utilisé **Jib**, un plugin Maven qui permet de générer des images Docker optimisées sans avoir besoin d'un Dockerfile. Cela facilite le processus de build et rend le déploiement plus fluide.

Chaque microservice a donc son image Docker indépendante, tout comme le frontend Next.js qui est également packagé sous forme de conteneur.

4.2 Orchestration (Docker Compose / Swarm)

J'ai organisé le lancement de l'application avec **Docker Compose** en mode développement. Cela me permet de démarrer tous les services (frontend, backend, base de données, configuration, gateway...) avec une seule commande.

Pour simuler un environnement de production, j'ai également configuré un fichier `stack-swarm-dev.yml` compatible avec **Docker Swarm**. Cette orchestration me permet de tester la montée en charge, la séparation des rôles, et la gestion des dépendances réseau entre services.

Le réseau Docker est configuré en mode bridge, avec des alias nommés pour chaque service, ce qui permet une communication fluide et sécurisée.

4.3 Environnements DEV / PROD

J'ai pris soin de séparer les configurations pour l'environnement de développement et celui de production (simulé).

- Environnement DEV : ports exposés localement, logs détaillés, profils Spring actifs, rechargement à chaud

- Environnement PROD : ports restreints, logs réduits, variables d'environnement injectées via fichiers `.env`

Les configurations sont centralisées dans le `module-config` via Spring Cloud Config Server. Cela permet d'ajuster les paramètres sans redéployer les services.

4.4 Documentation technique et scripts

Pour faciliter la mise en route de l'application, j'ai créé plusieurs scripts :

- *install.sh* : installe les dépendances, initialise les sous-modules, prépare l'environnement
- *image-jib.sh* : génère toutes les images Docker avec Jib
- *docker-compose-dev.yml* : lance l'environnement de développement complet
- *stack-swarm-dev.yml* : déploie l'environnement en mode Swarm

J'ai documenté l'ensemble de ces fichiers dans le *README.md* principal du projet. Celui-ci contient :

- Un tableau récapitulatif des services et de leurs ports
- Des instructions pas à pas pour le clonage et l'installation
- Une description claire de l'infrastructure réseau, sécurité et persistance

Cette documentation permet à n'importe quel développeur de reprendre le projet rapidement, ce qui est essentiel dans un contexte professionnel.

5. Bilan de projet

5.1 Résultats obtenus

Ce projet m'a permis de consolider et de mettre en pratique toutes les compétences attendues d'un concepteur-développeur d'applications. J'ai réussi à :

- Concevoir une architecture logicielle modulaire et évolutive
- Développer une application complète de type front + back
- Mettre en place un système d'authentification sécurisé avec JWT
- Exploiter deux types de bases de données (relationnelle et NoSQL)
- Conteneuriser et orchestrer tous les services avec Docker
- Documenter, tester et automatiser l'installation de l'environnement

L'application fonctionne comme prévu : chaque service est isolé, la communication entre microservices est fluide, et l'interface permet une navigation claire pour l'utilisateur

5.2 Difficultés rencontrées et solutions

J'ai rencontré plusieurs défis au cours de ce projet :

- **Volume de travail** : gérer seul toute la conception, le code, la configuration et les tests a demandé beaucoup de rigueur et d'organisation. J'ai planifié les tâches par module et utilisé Git pour maintenir une bonne traçabilité.
- **Interopérabilité entre bases PostgreSQL et MongoDB** : les données de note (MongoDB) étant indépendantes de la base relationnelle, j'ai dû construire une logique d'agrégation dans les services pour croiser les informations à l'affichage.

- **Gestion des configurations multi-environnement** : j'ai résolu cela avec Spring Config Server et des profils `dev` / `prod` bien séparés.
- **Orchestration réseau** : comprendre et maîtriser le fonctionnement de Docker Swarm m'a demandé du temps.

5.3 Perspectives d'amélioration

Même si le projet est fonctionnel, je prévois plusieurs axes d'amélioration :

- Ajouter une interface d'administration complète pour gérer les utilisateurs
- Améliorer la couverture de tests (unitaires, d'intégration, end-to-end)
- Intégrer un pipeline CI/CD avec GitHub Actions pour automatiser les builds et déploiements
- Mettre en place un système de monitoring et d'alerting avec Prometheus et Grafana
- Optimiser les performances de certaines requêtes API (pagination, filtres)

Enfin, ce projet pourrait évoluer vers un véritable produit de suivi médical en ajoutant des fonctionnalités comme la génération de rapports PDF, la gestion de rendez-vous ou l'analyse automatique par IA.

6. Conclusion générale

Ce projet m'a permis de passer de la théorie à la pratique de manière complète. J'ai pu appliquer les compétences acquises durant la formation CDA dans un contexte concret et réaliste.

En concevant moi-même l'architecture, en développant chaque module, en mettant en place la sécurité, les bases de données, le déploiement, et en rédigeant la documentation technique, j'ai couvert l'ensemble du cycle de vie d'une application moderne.

J'ai beaucoup appris sur :

- L'importance d'une bonne organisation des tâches, surtout en travaillant seul
- La gestion multi-environnement avec Docker
- Le fonctionnement d'une architecture orientée microservices
- La mise en place d'une sécurité robuste côté API

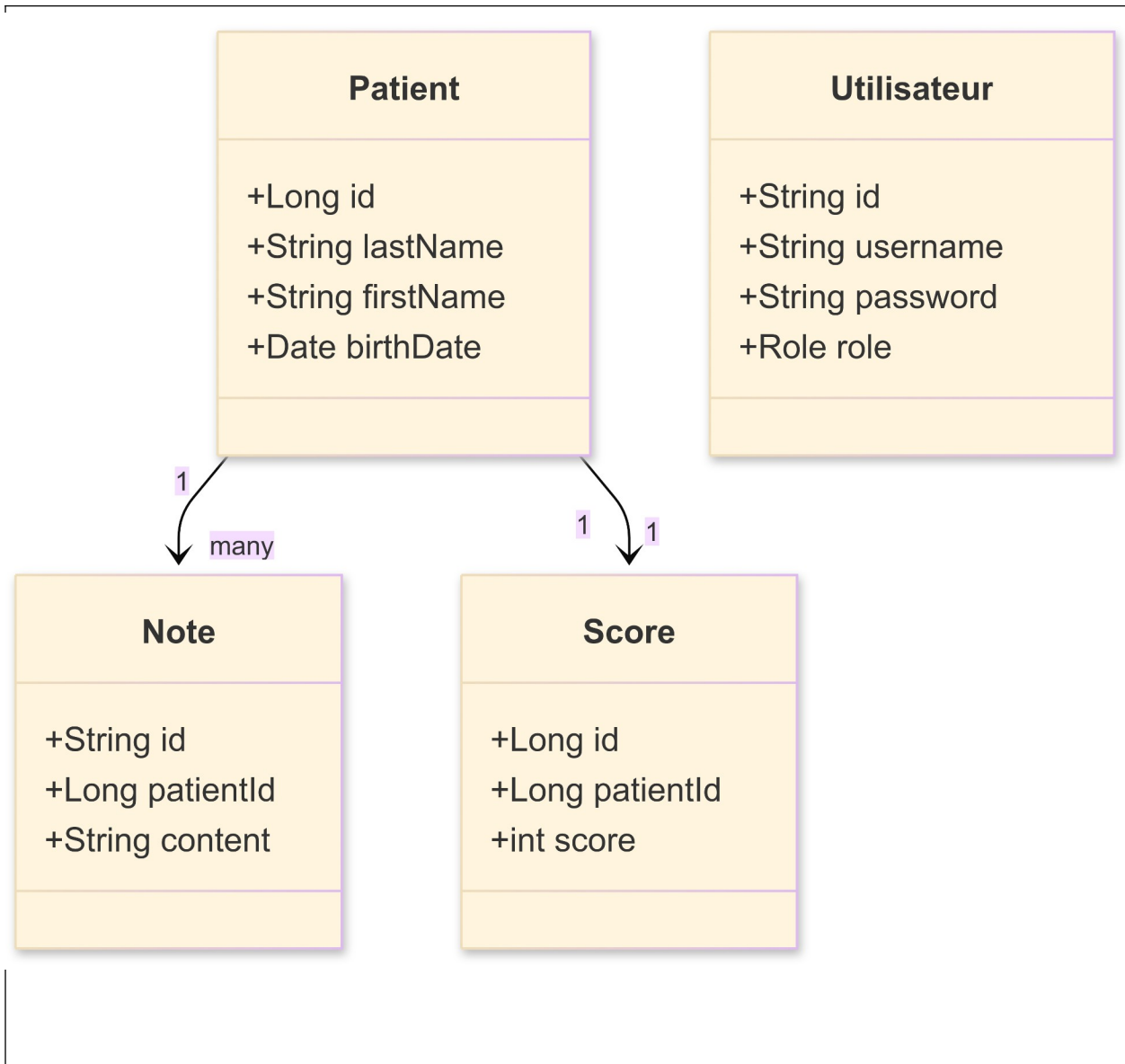
Je suis fier du résultat, car ce projet représente non seulement une application fonctionnelle, mais aussi une vraie démonstration de ma capacité à concevoir, développer, déployer et maintenir une solution logicielle professionnelle.

Enfin, ce projet a renforcé mon envie de continuer à évoluer dans le développement back-end et les architectures distribuées, tout en gardant une ouverture sur le DevOps et l'industrialisation logicielle.

Il constitue pour moi une excellente base de portfolio et un appui solide pour mes prochaines expériences professionnelles.

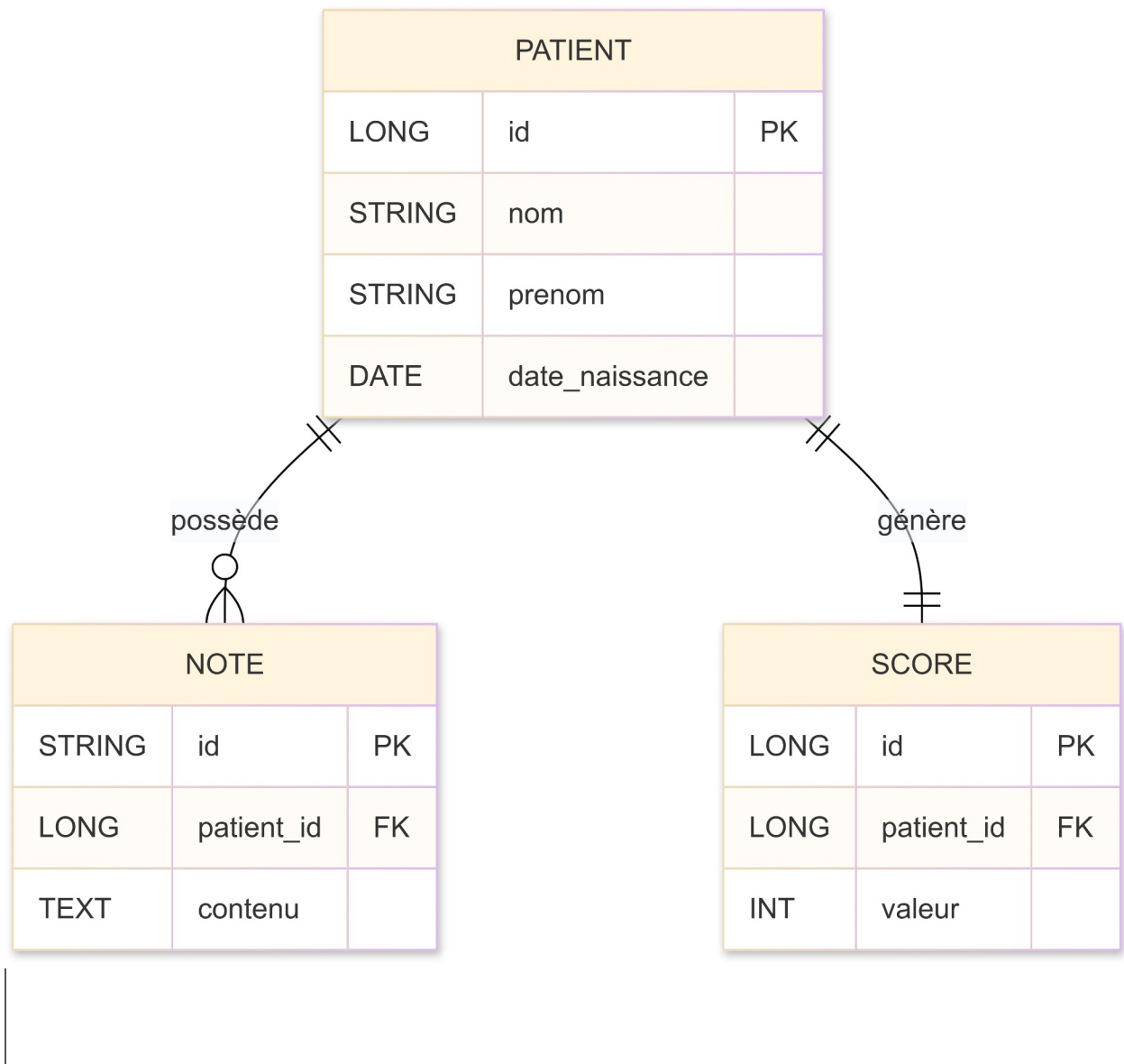
Annexes

Annexe A – Diagramme de classes UML



Ce diagramme illustre la structure des entités principales : Patient, Note, Score, Utilisateur, et leurs relations dans l'environnement Spring Boot.

Annexe B – Modèle Conceptuel de Données (MCD)



Le MCD met en évidence les relations entre patients, notes et scores dans un contexte relationnel (PostgreSQL).

Annexe C – Modèle NoSQL (MongoDB – Note)

NOTE		
STRING	_id	PK
LONG	patientId	
STRING	content	
DATE	createdAt	

Structure typique d'un document de note dans la base MongoDB utilisée par le microservice `note-service`.

Annexe D – Extraits de code commentés

Exemple 1 : Service de scoring (Java)

```
@Service
public class ScoringService {
    public int calculateScore(List<String> notes) {
        int score = 0;
        for (String note : notes) {
            if (note.contains("Hemoglobin A1c") || note.contains("Weight Loss")) {
                score++;
            }
        }
        return score;
    }
}
```

Ce service permet de calculer un score de risque en analysant des mots-clés médicaux dans les notes.

Exemple 2 : Authentification JWT (Java)

```
@RestController
@RequestMapping("/auth")
public class AuthController {
    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody LoginRequest request) {
        String token = authService.authenticate(request);
        return ResponseEntity.ok(new JwtResponse(token));
    }
}
```

Point d'entrée sécurisé pour la génération de tokens JWT.

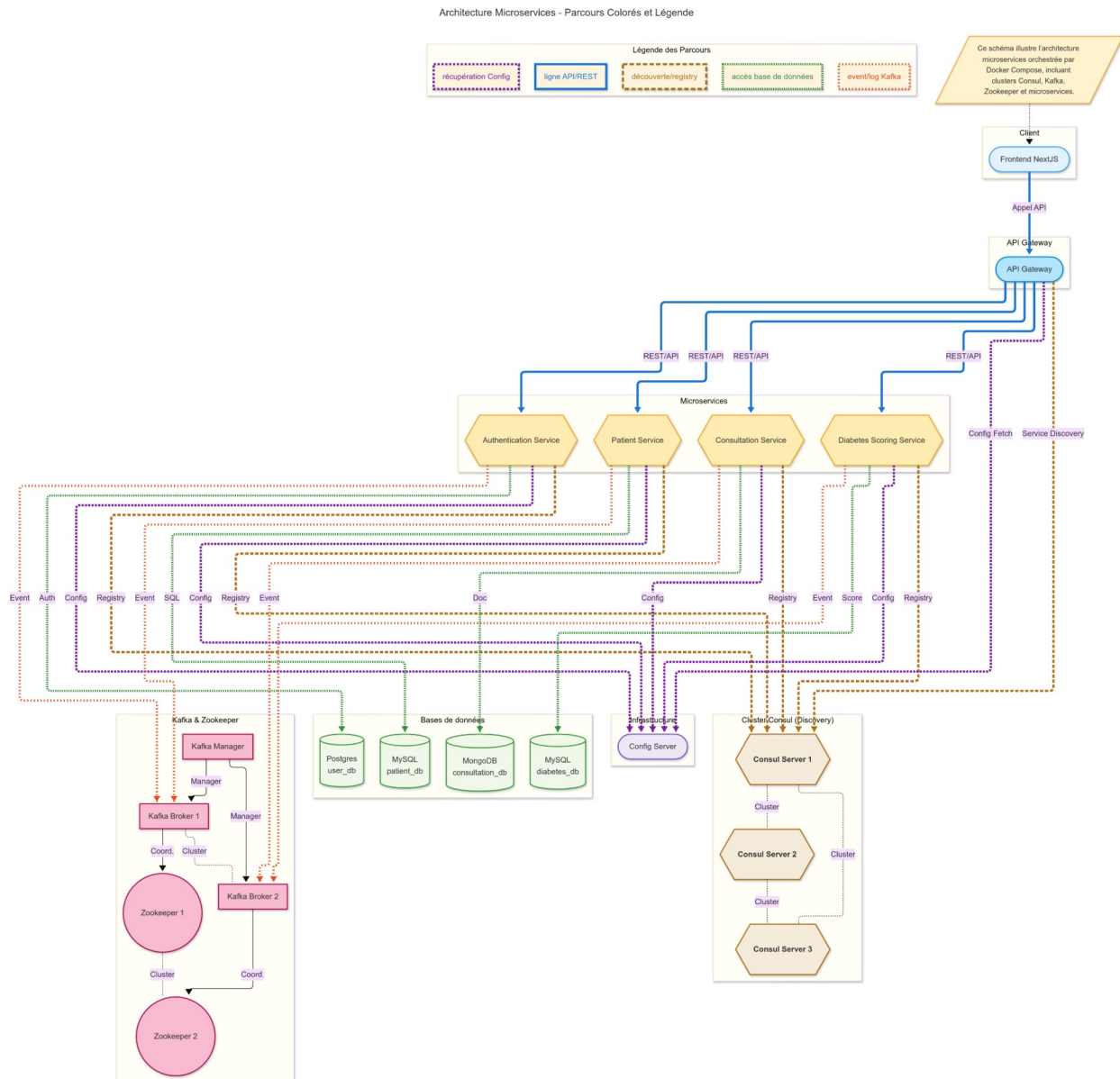
Exemple 3 : Composant React – Dashboard patient

```
const PatientCard = ({ patient }) => (
  <div className="rounded shadow-md p-4">
    <h3 className="text-xl font-bold">{patient.lastName} {patient.firstName}</h3>
    <p>Age : {patient.age}</p>
    <Link href={` /patients/${patient.id}`}>Voir fiche</Link>
  </div>
);
```

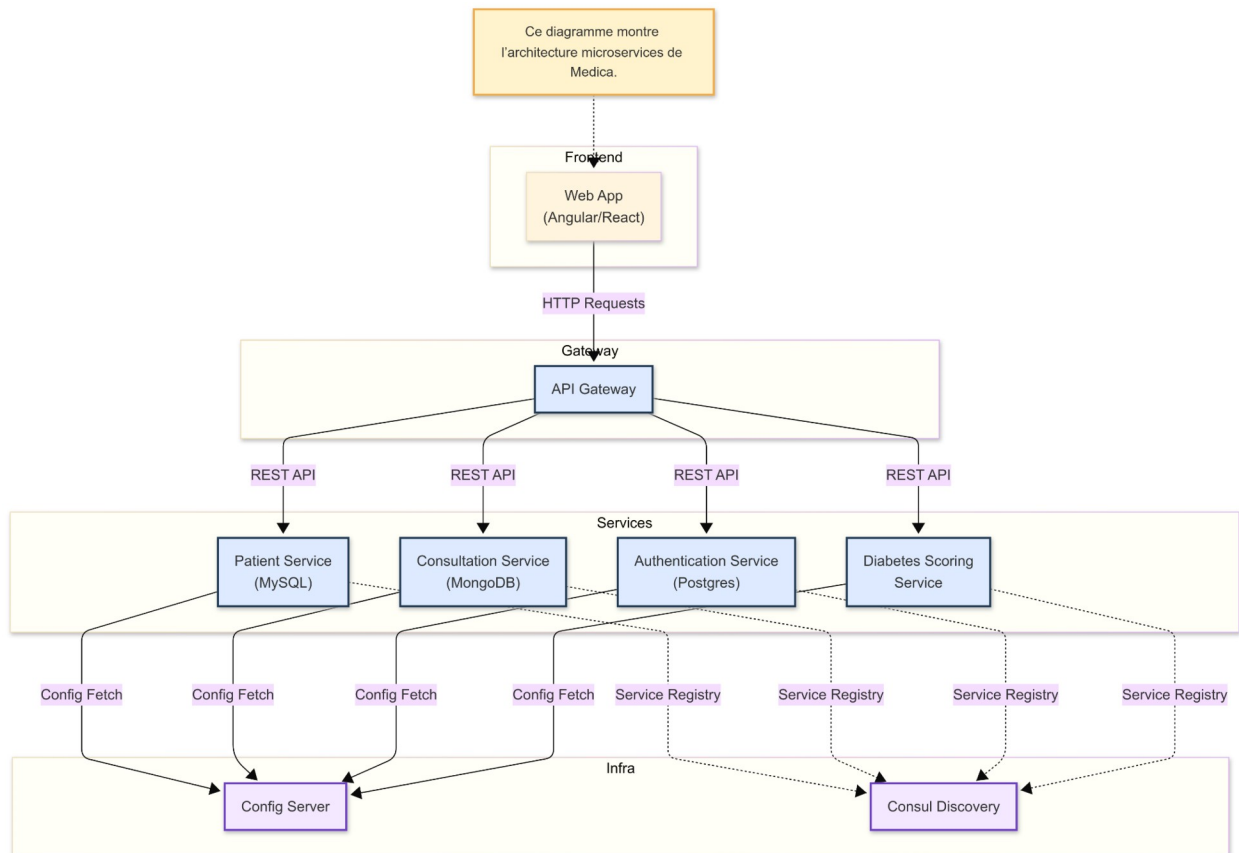
Carte affichant un résumé de chaque patient dans l'interface React/Next.js.

Annexe G – Diagrammes d'Architecture

Les diagrammes suivants présentent l'architecture microservices du projet :



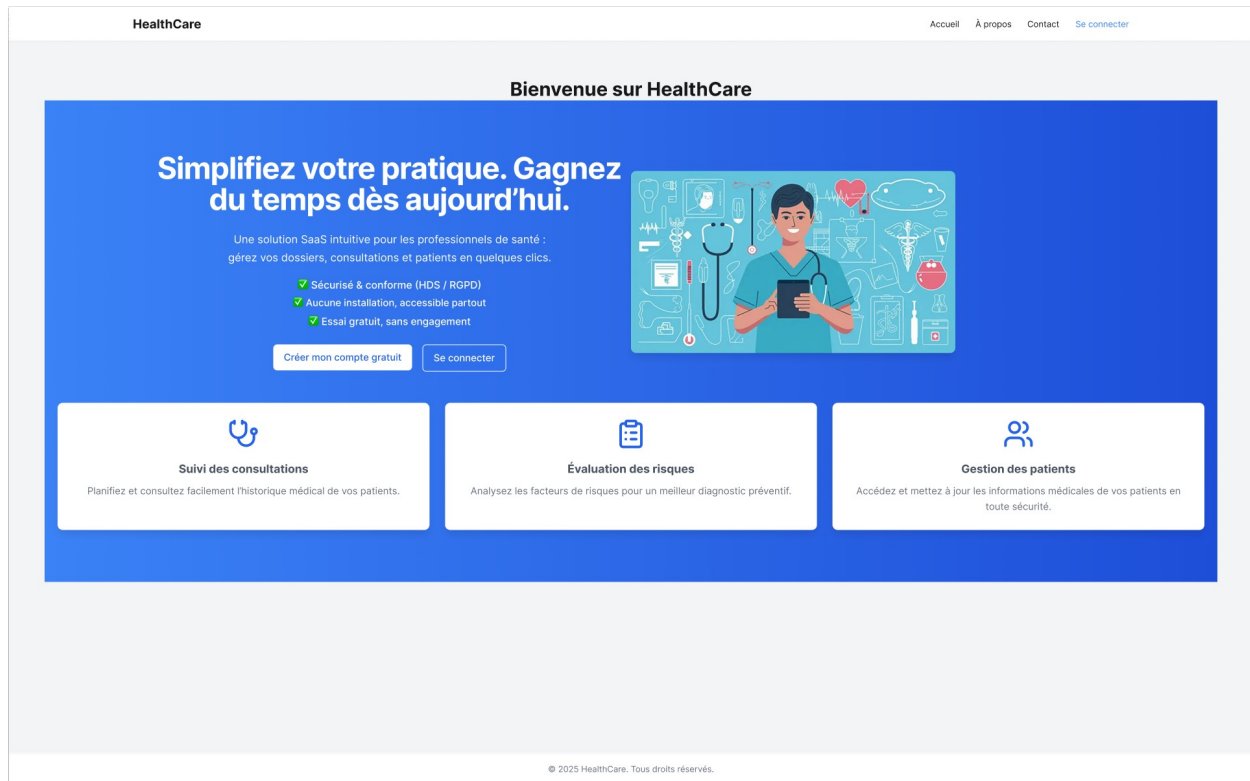
Architecture détaillée avec infrastructure Docker, Kafka, BDD et microservices.



Vue simplifiée de l'architecture logique de Workspace HealthCare.

Annexe H – Maquettes de l'Interface Utilisateur


Extraits visuels des écrans du projet (accueil, login, dashboard, etc.) :



Maquette – Écran d'accueil (version desktop)

Bienvenue sur Healthcare Panel

Plateforme sécurisée pour les praticiens et les patients.
Veuillez vous connecter pour accéder à votre espace personnel.

 Je suis un praticien

Connexion

Se connecter avec Google

Se connecter avec GitHub

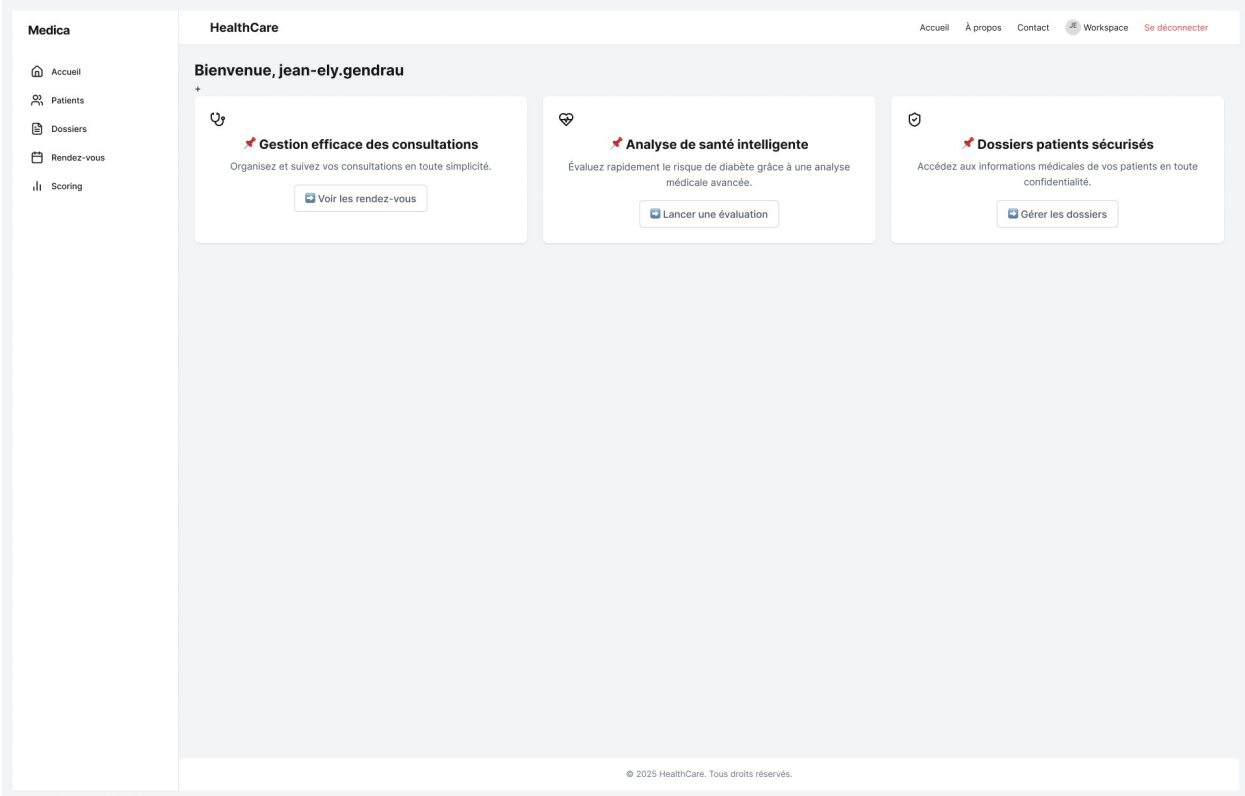
Maquette – Écran de connexion (version desktop)

Accès refusé

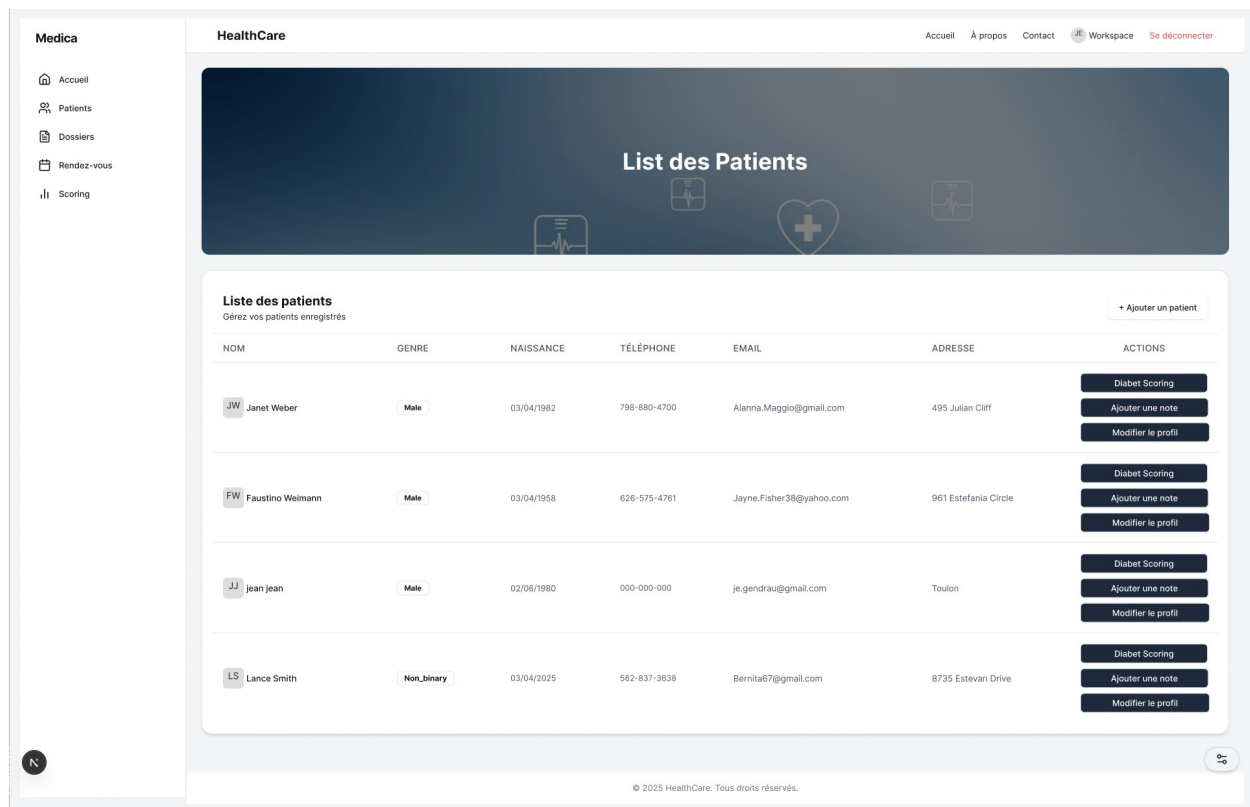
Vous n'avez pas l'autorisation d'accéder à cette page. Veuillez contacter un administrateur ou vous reconnecter avec un autre compte.



Maquette – Écran refusant un accès non autorisé (version desktop)



Maquette – Écran d’accueil workspace PRACTIONNER (version desktop)



Maquette – Écran workspace list des patients - PRACTIONNER (version desktop)