Conception et Réalisation d'une application web pour la gestion d'un cabinet médical

Loustau Valentin Duboureau Guillaume Ephrem Jennifer
Abdoul Goudoussy Diallo

Table des matières

Table des figures

Chapitre 1

Introduction

La gestion d'un cabinet médical peut vite devenir compliquée pour les professionnels de santé. L'analyse, la collecte et la gestion des données sont des éléments clés pour fournir des soins de qualité aux patients. Pour faciliter ces tâches, nous avons conçu une application de gestion d'un cabinet médical qui permettra aux professionnels de santé de gérer efficacement leurs rendez-vous, les dossiers médicaux ainsi que les ordonnances de leurs patients. Notre interface intuitive et simple d'utilisation permettra aux patients de trouver un rendez-vous avec le spécialiste de leur choix en un rien de temps.

L'objectif et la problématique, ici, est de concevoir une application web qui soit la mieux construite possible. Cela permettra ainsi d'avoir une gestion d'un cabinet médicale la plus simple et fluide possible. Pour cela, nous devrons répondre à certains besoins, que nous détaillerons dans les parties suivantes, tels que la centralisation des dossiers médicaux et de facilité la navigation et l'expérience utilisateur, que se soit pour le médecin ou la patient.

Dans ce rapport, nous allons décrire en détail la conception de cette application en expliquant les choix que nous avons faits pour les différentes fonctionnalités. Nous fournirons un cahier d'analyse des besoins avec des diagrammes de cas d'utilisations et un diagramme de classe pour illustrer la conception de l'application. Nous présenterons également les technologies et outils que nous avons utilisés pour développer cette application, notamment ReactJS et Java. Vous trouverez en dernière partie de ce rapport, toutes les fonctionnalités, imagées, que nous avons implémentées.

Enfin, nous discuterons des améliorations futures possibles de notre application et des opportunités d'extension de ses fonctionnalités.

Chapitre 2

Analyse des besoins

2.1 Identification des acteurs

Un acteur représente un rôle joué par une personne externe ou par un processus qui interagit avec le système.

Les acteurs de notre système sont :

- **Patient** : il s'agit d'un acteur qui utilise le site pour gérer (ajouter, consulter, supprimer) ses rendez-vous, consulter son dossier médical. Il peut également mettre à jour ses informations personnelles et communiquer ses infos au médecin.
- **Médecin** : il s'agit d'un acteur qui gère les dossiers des patients, prescrit des ordonnances et les imprime. Il s'occupe de la gestion des rendez-vous de ses patients et les notifie s'il effectue quelconque modification.

2.2 Les besoins fonctionnels

2.2.1 Pour le patient

1. Création de compte

   - Quantifications : Pouvoir créer qu'un seul compte par adresse mail.
   - Contraintes ou difficultés techniques : Sécurité des données personnelles des patients.
   - Énonciation des risques et parades : Risque de vols des données.
     Parade : Protocole de sécurité pour protéger les données.
   - Spécification des tests de contrôle : Tests unitaires.

2. Gestion et consultation des rendez-vous en ligne

   - Quantifications : Pouvoir prendre plusieurs rendez-vous selon les créneaux libres sur le planning. De plus, possibilité de visualiser ses rendez-vous en temps réel.
   - Éléments de faisabilité : Mettre en place un système de gestion de rendez-vous en ligne tel que doctolib et vérifier sa faisabilité.
   - Contraintes ou difficultés techniques : Assurer la synchronisation en temps réel avec le calendrier du médecin, garantir la disponibilité d'un rendez-vous.
   - Énonciation des risques et parades : Chevauchement de prise de rendez-vous pour des patients différents, ou sélectionner un rendez-vous qui n'existe plus.
     Parade : Bloquer la prise d'un rendez-vous pour le premier arrivé.
   - Spécification des tests de contrôle : Tests unitaires.

2.2.2 Pour le médecin

1. Gérer (ajouter/modifier/supprimer documents) les dossiers médicaux des patients

   - Quantifications : Vue complète des dossiers des patients en temps réel.
   - Éléments de faisabilité : Évaluation de différents systèmes de dossiers médicaux pour comprendre les fonctionnalités.
   - Contraintes ou difficultés techniques : Sécurité des données.
   - Énonciation des risques et parades : Risques de pertes de données à cause d'un souci technique, risque de non synchronisation/mise à jour.
     Parade : Historique de sauvegarde.
   - Spécification des tests de contrôle : Tests de sécurité.

2. Gestion de ses rendez-vous en ligne

   - Quantification : Possibilité de visualiser son calendrier de rendez-vous en temps réel.
   - Éléments de faisabilité : Mettre en place un système de gestion de rendez-vous en ligne tel que doctolib et vérifier sa faisabilité.
   - Contraintes ou difficultés techniques : Assurer la synchronisation en temps réel avec le calendrier, garantir la disponibilité d'un rendez-vous (ne pas placer le rendez-vous sur le créneau d'un autre patient, dû à une synchronisation non effectué).
   - Énonciation des risques et parades : Rendez-vous en simultané pour des patients différents, sélectionner un rendez-vous qui n'existe plus.
     Parade : Bloquer la prise d'un rendez-vous pour le premier arrivé.
   - Spécification des tests de contrôle : Tests unitaires.

3. Communication avec les patients

   - Éléments de faisabilité : Mise en place d'un système de notifications par e-mail.
   - Contraintes ou difficultés techniques : Aucune contrainte.

4. Consulter son planning des rendez-vous

   - Quantification : Le médecin doit pouvoir consulter son planning pour chaque jour de la semaine, ainsi que pour une période donnée : 1 semaine. Le planning doit être mis à jour en temps réel lorsque des rendez-vous sont ajoutés ou supprimés.
   - Éléments de faisabilité : La consultation du planning peut être réalisée en utilisant une interface graphique simple, qui affiche les rendez-vous (stockés dans une base de données) en fonction de l'heure et de la date tel que le logiciel Google Calendar.
   - Contraintes ou difficultés techniques : Garantir la confidentialité des informations stockées dans la base de données.
   - Énonciation des risques et parades : Risque que les rendez-vous soient compromis si la base de données est piratée. Risque que le planning du médecin ne soit pas à jour après modification.
     Parade : Le système peut être conçu pour utiliser des méthodes de cryptage pour protéger les données sensibles. Actualisation du planning après chaque modification.
   - Spécification des tests de contrôle : Des tests de validation peuvent être effectués pour vérifier que le système affiche correctement les rendez-vous dans l'interface utilisateur. Des tests de contrôle peuvent être effectués pour vérifier que les données sont correctement stockées dans la base de données et affichées dans l'interface utilisateur.

5. Bilan de santé (Le système doit assurer l'impression des fiches malades et les bilans) :

   - Contraintes ou difficultés techniques : Les fichiers doivent être au format PDF exclusivement pour ne pas perdre d'informations et pour une meilleure impression.

- Énonciation des risques et parades : Risque de mauvais transfert/perte de données lors de l'envoie du bilan par le médecin au patient.
  Parade : Nous pourrons vérifier si le document reçu par le patient est vide (null) ou non.

2.3 Les besoins non-fonctionnels

Ce sont des besoins en relation avec la performance du système, la facilité d'utilisation, l'ergonomie des interfaces, la sécurité etc. Et parmi ces besoins nous citons :

1. Sécurité des données médicales des patients

   - Éléments de faisabilité : Mise en place d'un système d'identification et d'authentification de chaque utilisateur (patients/médecins) pour garantir que seuls les utilisateurs autorisés aient accès au système et aux données sensibles.
   - Contraintes ou difficultés techniques : Complexité de la gestion des autorisations d'accès pour les différents utilisateurs.
   - Énonciation des risques et parades : Risque de faille et accès à un intrus aux données des patients.
     Parade : Il faudrait pour cela protéger ces données.
   - Spécification des tests de contrôle : Connexion au logiciel avec les différents acteurs et vérifier leur autorisation (un patient n'a pas accès au rendez-vous de tous les patients par exemple).

2. Simplicité et ergonomie de l'interface graphique

   - Éléments de faisabilité : Mise en place d'une interface intuitive et facile à utiliser pour les utilisateurs, en utilisant des icônes, des boutons et d'autres éléments de conception familiers (système de navigation logique), avec des choix judicieux de couleurs, de polices et d'images pour renforcer la clarté et la lisibilité de l'interface.
   - Contraintes ou difficultés techniques : Temps nécessaire pour trouver le juste équilibre entre simplicité et fonctionnalité mais aussi pour la conception et la mise en œuvre d'une telle interface.
   - Énonciation des risques et parades : Risque de confusion pour les utilisateurs et bug de navigation.
     Parade : Il faudrait faire tester le logiciel par plusieurs utilisateurs et recueillir un feedback.
   - Spécification des tests de contrôle : Tests unitaires.

3. Performance du système en temps de réponse, stockage mémoire

   - Quantification : Temps de réponses en fonction des différentes actions (pour une requête utilisateur, pour le chargement d'une page).
   - Éléments de faisabilité : Un essai peut être implémenté pour mesurer la performance actuelle du système et trouver des améliorations à faire.
   - Contraintes ou difficultés techniques : Posséder un stockage suffisant pour contenir toutes les données enregistrées, réussir à élaborer des algorithmes pour exécuter les requêtes en temps réel.
   - Énonciation des risques et parades : Les risques peuvent être un stockage insuffisant en mémoire.
     Parade : Utilisation de technologies de stockage plus performantes.
   - Spécification des tests de contrôle : Tests de performance pour mesurer les temps de réponse, tests de stockage pour vérifier que la mémoire est suffisante.

2.4 Diagrammes de cas d'utilisation

2.4.1 Créer un compte

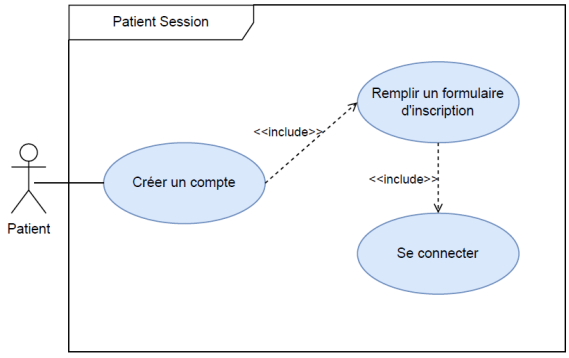

FIGURE 2.1 – Diagramme de cas d'utilisation : créer un compte

2.4.2 S'authentifier

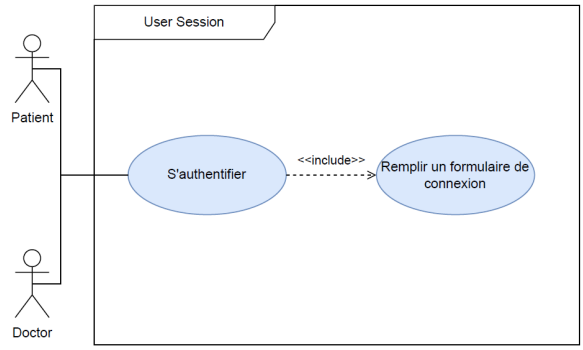

FIGURE 2.2 – Diagramme de cas d'utilisation : s'authentifier

2.4.3 Gérer les rendez-vous en ligne (Patient)

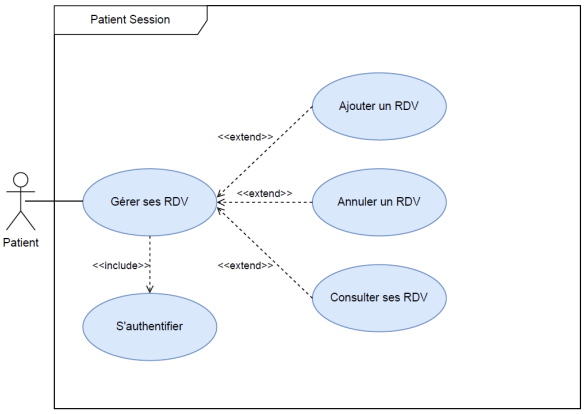

FIGURE 2.3 – Diagramme de cas d'utilisation : Gérer les rendez-vous en ligne (Patient)

2.4.4 Gérer les rendez-vous en ligne (Médecin)

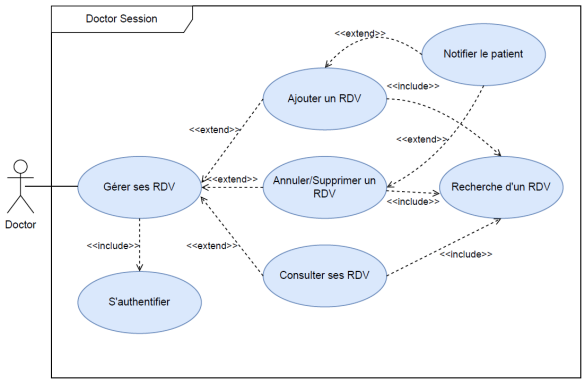

FIGURE 2.4 – Diagramme de cas d'utilisation : Gérer les rendez-vous en ligne (Médecin)

2.4.5 Consulter les rendez-vous

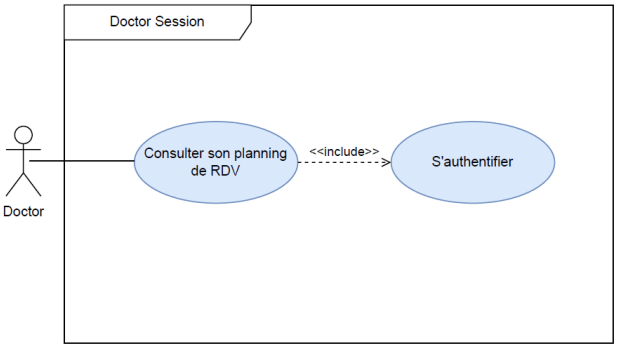

FIGURE 2.5 – Diagramme de cas d'utilisation : Consulter les rendez-vous

2.4.6 Consulter le dossier médical

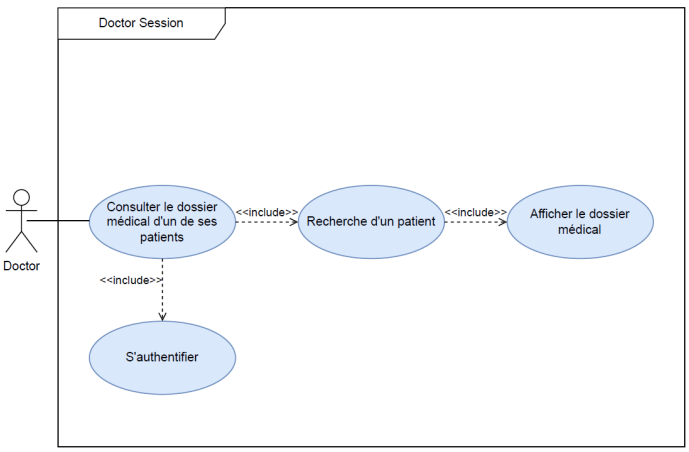

FIGURE 2.6 – Diagramme de cas d'utilisation : Consulter le dossier médical

2.4.7 Gérer le dossier médical

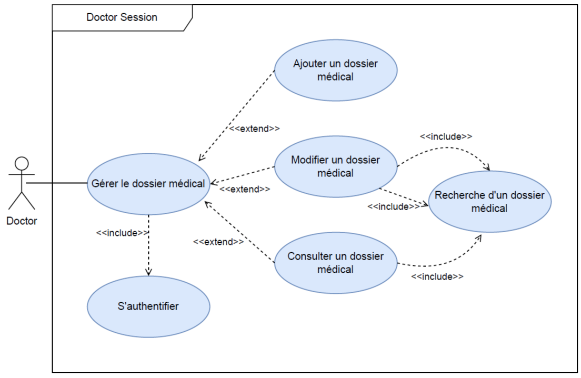

FIGURE 2.7 – Diagramme de cas d'utilisation : Gérer le dossier médical

2.4.8 Génerer le bilan de santé

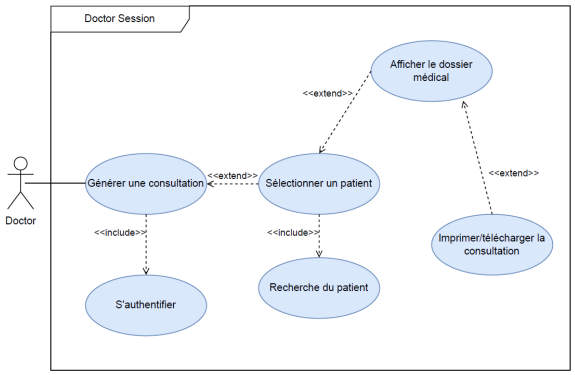

FIGURE 2.8 – Diagramme de cas d'utilisation : Génerer le bilan de santé

2.5 Diagrammes de classe du Domain

Concernant le diagramme de classe, nous avons représenté seulement les classes du Domain car elle représente le coeur de l'architecture DDD.

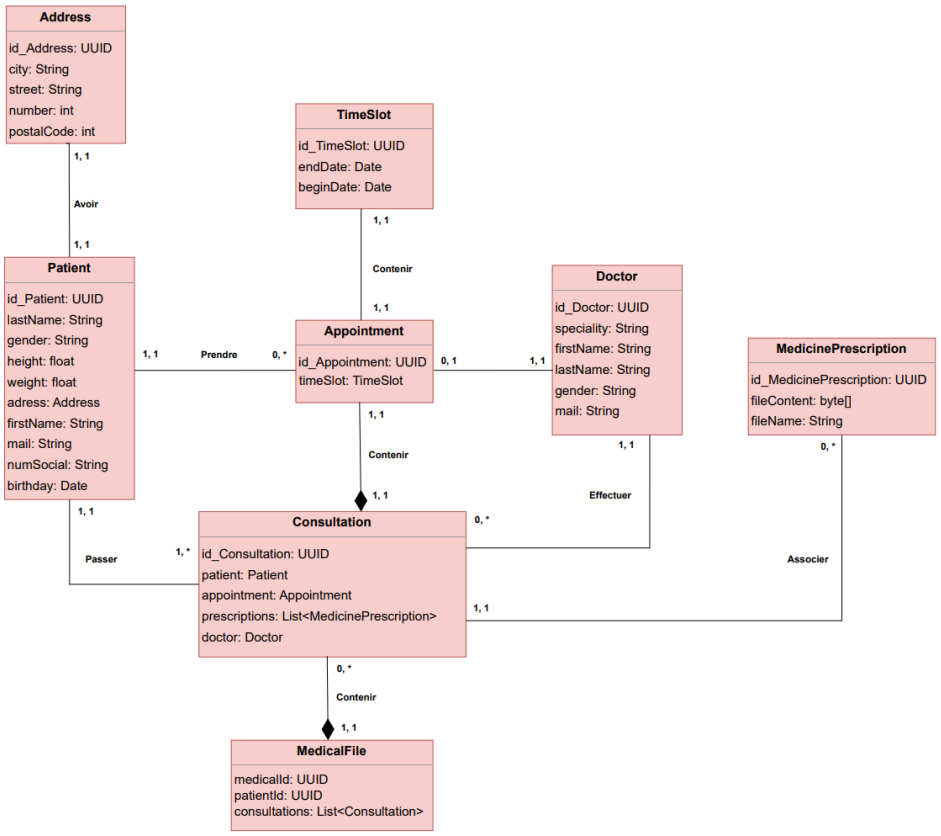

FIGURE 2.9 – Diagramme de classes

Chapitre 3

Conception

L'architecture logicielle de notre application web pour la gestion d'un cabinet médical se compose d'un front-end, d'un back-end et d'une base de données. Pour le back-end, nous avons décidé d'utiliser une architecture DDD (Domain-Driven Design) qui se compose de quatre couches distinctes : une couche domaine, une couche infrastructure, une couche application et une couche interface utilisateur.

Dans la couche domaine, on trouve les éléments clés de notre application tels que les patients, les médecins, les rendez-vous, les créneaux horaires, les consultations, les adresses des patients et les prescriptions médicales. Nous avons distingué les "values object" des Entity et de notre Aggrégat, le dossier médical. Ainsi, nous pouvons recréer nos objets en les récupérant de la base de données ou alors sauvegarder ces objets. C'est ce qui nous permet de faire le pont entre la base de données et le frontend qui reçoit et met en forme les données.

Nous avons également mis en place une couche infrastructure en utilisant des bases de données relationnelles pour stocker les informations de l'application. Cette couche infrastructure est responsable de créer pour chaque classe du domaine une table et de gérer les requêtes et les mises à jour dans celles-ci.

Dans la couche interface utilisateur, on a des controllers qui servent d'intérmédiaires entre la partie client et la partie serveur. Les controllers reçoivent les entrées utilisateurs, les envoient à la couche application et envoient une réponse au client. Les controllers renvoient en général une réponse HTTP afin de distinguer les différents code HTTP que l'on reçoit (ok, forbiden...)

Avec cette architecture DDD, nous avons créé une application structurée pour répondre aux besoins spécifiques d'un cabinet médical. Le code est ainsi plus facile à lire car les données sont structurées par couches.

Concernant le frontend, nous nous occupons ici de présenter les données, et c'est donc une partie de conception graphique et de présentation aux utilisateurs et non pas une partie de conception d'un modèle pour la modélisation du cabinet médical. Les données sont donc traitées dans le backend pour pouvoir être stockées dans la base de données.

Chapitre 4

Réalisation

4.1 Outils de développement

- Java : Nous avons utilisé le langage de programmation orienté objet Java pour notre projet.
- ReactJS : Pour le côté Front-end, nous avons utilisé la bibliothèque JavaScript ReactJS qui est très populaire dans le monde du développement web et très facile d'utilisation. Elle fonctionne avec des composants où chaque composant représente une partie de l'interface utilisateur. Ces composants sont réutilisables ce qui nous permet de gagner du temps. Elle possède également une documentation bien organisée et facile à comprendre.
- Maven : Pour la construction du projet nous avons choisi Maven, un outil de gestion de projet facile à configurer et à utiliser. Il peut être étendu grâce à des plugins pour répondre à des besoins spécifiques.
- Git : Pour pouvoir au mieux travailler sur ce projet chacun de notre côté, nous avons décidé d'utiliser Git qui est un logiciel de gestion de versions décentralisé. Il permet de travailler sur des parties différentes du code en même temps et il est très utile pour le travail en équipe.
- Postgresql : Pour la gestion de notre base de données nous avons utilisé Postgresql qui est un système de gestion de bases de données relationnelles. Il est fiable et robuste, de plus nous avions déjà utilisé ce système donc c'était facile pour nous de l'utiliser pour notre projet.
- JUnit : Pour les tests, nous avons utilisé JUnit qui est un framework de test unitaire pour Java et qui permet de tester des fragments de code de manière isolée et automatisée.

4.2 Côté back-end

Pour la partie back-end, nous avons entamé le processus par l'implémentation de la couche domaine en y incluant les protagonistes majeurs du cabinet médical : les docteurs et les patients. Nous avons poursuivi notre travail en créant des value objects et des entités tels que les adresses postales, les rendez-vous, les consultations, les ordonnances et les utilisateurs. Nous avons conclu en créant l'agrégat, qui est un dossier médical.

Ensuite, nous avons développé la couche infrastructure en créant quatre dépôts distincts : un pour les médecins, un pour les patients, un pour les utilisateurs et un pour le cabinet médical dans son ensemble. Ces dépôts contiennent les requêtes et les insertions dans les tables de la base de données.

Par la suite, nous avons travaillé sur la couche application, qui récupère les informations de la couche infrastructure afin de les transmettre à la couche interface utilisateur.

Enfin, nous avons implémenté la couche interface utilisateur, dans laquelle se trouvent des contrôleurs pour les patients, les médecins et les utilisateurs qui font le lien entre la partie serveur et la partie client.

4.3 Côté front-end

Le développement de l'interface utilisateur de notre application web a été effectué en utilisant React JS. Ce framework a permis de créer une interface utilisateur attrayante et réactive pour les utilisateurs.

Nous avons commencé par concevoir la page principale, qui comprend une liste déroulante permettant aux utilisateurs de sélectionner la spécialité du médecin recherché (Dentiste, Généraliste, etc.).

Ensuite, nous avons créé les pages d'inscription et de connexion pour permettre aux patients de se connecter à leur compte ou d'en créer un nouveau. Une partie d'administration a également été développée, avec un router dédié permettant d'accéder à toutes les pages disponibles pour les patients connectés, telles que la page "Mes rendez-vous", "Mes documents" et "Mon compte", où les patients peuvent modifier leurs informations personnelles. Un header a également été ajouté pour faciliter la navigation entre ces différentes pages.

Dans la partie publique de l'application, accessible à tous les utilisateurs, y compris ceux qui ne sont pas connectés, nous avons créé des pages dédiées à chaque médecin pour afficher les créneaux horaires disponibles. Nous avons ensuite créé une partie réservée uniquement aux médecins, où ils peuvent accéder à leur planning, visualiser leurs patients et modifier leurs informations personnelles.
En somme, nous avons mis en place un router public qui comprend un router pour les patients et un autre pour les médecins, permettant une navigation aisée et intuitive pour tous les utilisateurs.

4.4 Base de données

Nous avons créé une base de données à l'aide du système Postgresql dans laquelle nous avons ajouté des tables pour chaque classe de notre domaine DDD.

— **Table doctors** : on stocke les informations relatives aux docteurs comme leur nom, prénom, spécialité et les informations qu'on retrouve sur leur page Medicolib.

— **Table patients** : on stocke les informations personnelles des patients comme leur nom, prénom, genre, poids, taille, adresse mail et date de naissance.

— **Table address** : on stocke les différentes parties d'une adresse postale d'un patient tel que le numéro et nom de la rue, le code postal et le nom de la ville.

— **Table users** : on stocke les adresses mails et mots de passe cryptés des utilisateurs ainsi que leur rôle (patient ou doctor).

— **Table availableTimeSlots** : on stocke les créneaux de rendez-vous disponibles pour chaque médecin.

— **Table appointments** : on stocke tous les rendez-vous pris pour chaque médecin et chaque patient.

— **Table documents** : on stocke les documents que les patients peuvent ajouter en amont de leur rendez-vous.

— **Table consultations** : on stocke les informations d'une consultation entre un médecin et un patient et un identifiant d'ordonnance si il y en a une à la suite de la consultation.

— **Table prescriptions** : on stocke les ordonnances que le médecin prescrit au patient.

— **Table medicalFile** : on stocke les dossiers médicaux des patients pour chaque médecin, qui correspondent à une liste de leurs consultations.

— **Table price** : on stocke le prix des services proposés pour tous les médecins c'est-à-dire l'identifiant du docteur avec l'intitulé de la prestation et le prix.

Chapitre 5

Tests

Nous allons maintenant aborder la partie sur les tests. Nous avons utiliser JUnit pour effectuer des tests unitaires sur les contrôleurs. JUnit est un framework permettant de développer des tests unitaires automatisables. Ainsi, nous pouvous nous assurer que le code répond toujours aux besoins après avoir implémenté ou effectué des modifications. Ces tests correspondent à des assertions qui permettent de tester les résultats attendus. L'objectif ici, est de tester les méthodes des différents contrôleurs que nous avons implémentés pour vérifier que le code que nous avons produit est bien correct. Les tests sont séparés du code de la classe permettant ainsi de la tester. Le principe d'implémentation de ces tests est le suivant :

- Création d'une instance du contrôleur et de tout autre objet nécessaire aux tests
- Appel de la méthode à tester
- Comparaison du résultat attendu avec le résultat que l'on vient d'obtenir

L'avantage est de pouvoir écrire un certain nombre de tests permettant de tester chacun différents cas de figure et de trouver le plus de bugs possibles dans notre application. Nous utilisons également, des objets Mock (du framework Mockito). MockMvc est en fait un objet de Spring permettant de simuler un objet qu'on aura besoin d'utiliser pour effectuer nos tests. Cela permet notamment, de simuler des accès à la base de données. Tout cela nous permet ainsi de rédiger nos tests sur les contrôleurs. Nous pouvons ainsi vérifier que les reqûetes effectuées et les objets retournés et/ou intégrés dans la base de données sont conformes au résultat attendu. Nous avons trois contrôleurs donc trois classes de tests correspondantes.

5.0.1 UserControllerTest

Cette section correspond aux tests sur toute la partie "connexion" de notre site web. Plus précisement, nous testons ici que les connexions et rénitialisation de mot de passe s'effectuent bien. Nous avons effectué des tests positifs et négatifs, c'est-à-dire avec des bons identifiants de connexion et des mauvais et nous nous s'assurons que les réponses HTTP renvoyées sont bien identiques. Par exemple, si un utilisateur se connecte avec un mauvais mot de passe, il faut que la réponse attendue soit "Forbiden". Ainsi, si un des tests ne passent pas, il est simple de savoir où l'erreur se produit. Voici un exemple d'un de nos tests pour vérifier qu'un utilisateur ne peut pas rénitialiser son mot de passe s'il n'est pas enregistré en tant qu'utilisateur (qu'il n'a pas de compte).

```
@Test
public void testResetPasswordWithNonUser() throws Exception {
    Map<String, String> map = new HashMap<>();
    map.put("mail", "test@gmail.com");

    when(medicalPractice.checkUserExist("test@gmail.com")).thenReturn(false);

    mockMvc.perform(post("/new-password")
            .contentType(MediaType.APPLICATION_JSON)
            .content(asJsonString(map)))
            .andExpect(status().isNotFound());
}
```

5.0.2 PatientControllerTest

Cette section va correspondre aux tests du controleur du patient et va, comme précédemment, tester les différentes fonctionnalités de l'API Rest (avec les codes HTTP). Nous testons ainsi toutes les demandes liées aux patients : modifications d'informations, prises et annulations de RDV, ajout de documents...Comme précédemment, nous avons, pour certaines méthodes, tester plusieurs cas, lorsque la méthode en question peut retourner une réponse "ok" ou "forbiden" par exemple. Voici un exemple d'un de nos tests pour vérifier qu'un utilisateur peut supprimer un document :

```
@Test
void testDeleteDocument() throws SQLException, IOException {
    HashMap<String, String> mockMap = new HashMap<>();
    mockMap.put("apptid", "94b9b73a-f561-4ca9-a6c3-6ae7e0361773");
    mockMap.put("mail", "johndoe@gmail.com");
    MockMultipartFile file = new MockMultipartFile("test", "test.txt", "text/
        plain", "file content".getBytes());
    patientController.uploadFile(file, mockMap.get("mail"), mockMap.get("apptid
        "));
    HashMap<String, String> map = new HashMap<>();
    map.put("id", "94b9b73a-f561-4ca9-a6c3-6ae7e0361773");
    map.put("name", file.getName());
    ResponseEntity<String> response = patientController.deleteDocument(map);
    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertEquals("File deleted successfully", response.getBody());
}
```

5.0.3 DoctorControllerTest

Cette section va correspondre aux tests du controleur concernant les docteurs. Nous allons, comme précédemment effectuer des tests sur les différentes méthodes du contrôleur et plus précisement sur les points suivants : la consultation du planning par le docteur, l'ajout de consultations, l'annulation de RDV...Voici un exemple de test pour récupérer la liste des patients d'un docteur :

```
@Test
void testGetAllPatientByDoctor() throws ParseException, SQLException {
    SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
    Date date = format.parse("2023-04-03");
    List<Patient> expectedPatients = new ArrayList<>();
    Patient patient = new Patient(
        UUID.fromString("dccf9cfd-f2cc-4e44-8357-dd4140e17b73"),
        "John" , "Doe", "M", date, "54165", "johndoe@gmail.com", null, 0, 0);
    expectedPatients.add(patient);
    Map<String, String> requestBody = new HashMap<>();
    requestBody.put("mail", "doctor@gmail.com");
    when(medicalPractice.getPatientsByDoctor(medicalPractice.
        getInformationsDoctorByMail("doctor@gmail.com"))).thenReturn(
        expectedPatients);
    ResponseEntity<List<Patient>> response = doctorController.
        getAllPatientByDoctor(requestBody);
    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertEquals(expectedPatients, response.getBody());
}
```

Pour conclure cette partie sur les tests, nous pouvons dire que les tests représentent une partie très importante du projet car elles nous permettent de suivre l'évolution de notre application et de détecter des bugs rapidement et lors de l'ajout de nouvelles fonctionnalités. L'avantage de JUnit est que c'est très simple à utiliser, et très rapide à mettre en place et cela nous permet d'écrire beaucoup de tests permettant de couvrir une grande partie de notre programme.

Chapitre 6

Résultats

(Mettre ici les captures et les fonctionnalités)

Chapitre 7

Conclusion