



**rappor<sup>t</sup> de stage**

**au sein de la société ACT**

**Du 12-11-2018 au 15-02-2019**

**Frédéric PEGUILHAN**

**Formation  
Concepteur - Développeur Informatique  
CDI 18-1**



## Remerciements

Je tiens à remercier vivement ACT et particulièrement Haydar Bonaud pour m'avoir donné la possibilité de faire mon stage.

Je remercie le pôle numérique de l'ADRAR ainsi que la région Occitanie pour m'avoir permis d'accéder à cette formation.

Je remercie également l'équipe de formateurs du pôle numérique, particulièrement Benoit Schiex et Cédric Viaud pour leurs conseils et leur bienveillance.

Enfin je remercie mes camarades de formation pour leur soutien et leur aide.

|   |    |
|---|----|
| A. Le projet  | 4  |
| I. Abstract   | 4  |
| II. Présentation                                      | 5  |
| III. Compétences couvertes par le stage               | 6  |
| B. Cahier des charges                                 | 7  |
| I. Présentation du projet « votre rendez-vous santé » | 7  |
| II. Charte graphique                                  | 8  |
| III. Cadre technique de l'application                 | 9  |
| IV. Prestations attendues                             | 9  |
| C. Spécification fonctionnelle                        | 10 |
| I. Fonctionnalités visiteur                           | 10 |
| II. Fonctionnalités patient                           | 10 |
| III. Fonctionnalités praticien                        | 11 |
| IV. Analyse des besoins                               | 12 |
| II. Use Case  | 13 |
| a. Diagramme use case                                 | 13 |
| b. Descriptions textuelles de « use case »            | 13 |
| III. Diagramme de séquence                            | 16 |
| IV. Diagramme de classe                               | 17 |
| IV. Maquettage  | 18 |
| D. Conception   | 21 |
| I. Modèle Conceptuel de Données                       | 21 |
| II. Modèle Logique de Données                         | 23 |
| III. Modèle Physique de Données                       | 24 |
| IV. Arborescence du site                              | 25 |
| E. Spécification technique                            | 26 |
| F. Réalisation  | 27 |
| G. Conclusion   | 54 |
| H. Annexes  | 55 |

## **A. Le projet**

### **I. Abstract**

My project consists in presenting tasks I dealt with during my internship.

This was in a start-up company called ACT, focusing on improving human well-being at work. It is also a company willing to offer more services to people, like a "therapeutic" app, which I was in charge of.

This web application is a tool for both patients and practitioners or therapists.

On the one hand, the patients will use it to create their own medical folder, identify the practitioners listed by specialties, localize them via a map, and communicate thanks to a dedicated message system. They also will be able to arrange their medical appointments the same way.

On the other hand, the practitioners will be able to manage each patient folder, medical documents and appointments. Obviously, they will also use the app to communicate with their patients.

I was in charge of the implementation of the « appointment » features using Angular 7 and Spring Boot 2.

I have chosen this project because I wanted to work with Java language and I knew that I would work on this feature from A to Z.

Despite the fact that I was discovering and learning two new frameworks, I reached the objectives. I worked within the ADRAR premises but I had the opportunity to get in touch with my tutor regularly to make sure I was on the right track. I know that ACT is satisfied with my job because my work on this app will be deployed to be testing.

## **II. Présentation**

J'ai effectué mon stage pour ACT. C'est une start-up qui oeuvre dans le domaine du bien-être. Matthieu Buchoud, son fondateur, est un ancien manager et dirigeant d'entreprise, aujourd'hui coach en gestion de stress et consultant en entreprise.

Peu avant le début de mon stage, ACT a commencé le développement d'une application Web. Cette application s'adresse aux professionnels de santé qui utilisent la micronutrition. En plus de la prise de rendez vous en ligne, l'atout principal est l'incorporation d'un questionnaire que le patient remplit en ligne avant la consultation. Cela permet au praticien grâce à un algorithme d'avoir déjà une idée des problématiques de la personne et donc de mieux cibler sa consultation. C'est le gros point fort et l'innovation de ce système.

Haydar Bonaud, mon tuteur de stage, est le seul développeur de l'application. Pendant le stage, j'ai été en charge de développer les fonctionnalités liées à la prise de rendez-vous et j'ai utilisé les frameworks Angular 2 et Spring Boot 2. J'ai choisi ce projet parce que je voulais travailler avec le langage Java et que je savais que je développerai ces fonctionnalités de A à Z.

Mon stage s'est déroulé dans les locaux de l'ADRAR et j'avais des entretiens hebdomadaires avec Haydar Bonaud. De plus je pouvais le contacter très facilement.

### **III. Compétences couvertes par le stage**

« Développer des composants d'interface »

- Maquetter une application
- Développer une interface utilisateur
- Développer des composants d'accès aux données
- Développer des pages web en lien avec une base de données

« Développer la persistance des données »

- Concevoir une base de données
- Développer des composants dans le langage d'une base de données
- Utiliser l'anglais dans son activité professionnelle en informatique

« Développer une application n-tiers »

- Concevoir une application
- Collaborer à la gestion d'un projet informatique
- Développer des composants métier
- Construire une application organisée en couches
- Développer une application de mobilité numérique

## B. Cahier des charges

### I. Présentation du projet « votre rendez-vous santé »

L'application « votre rendez-vous santé », développée par ACT, est une solution pour mettre en relation les patients et les praticiens en proposant divers services, notamment la recherche de praticien et la prise de rendez-vous, l'échange de documents personnels comme des prescriptions. Mais également la possibilité d'un suivi à long terme grâce à des questionnaires santé.

L'application doit permettre :

- pour un visiteur :
  - chercher un praticien par spécialité, par localisation ou par nom
  - S'inscrire
  - s'authentifier
- pour un patient :
  - chercher un praticien par spécialité, par localisation ou par nom
  - prendre un rendez-vous
  - annuler un rendez-vous
  - consulter ses rendez-vous
  - échanger des messages avec un praticien
  - échanger des documents avec un praticien
  - modifier ses informations personnelles
  - renseigner un questionnaire médical
- pour un praticien :
  - renseigner ses informations personnelles
  - organiser et consulter son planning : horaires, spécialités, indisponibilités, temps de consultation, sujet de rendez-vous
  - consulter le dossier d'un patient
  - prendre un rendez-vous à un patient
  - annuler un rendez-vous
  - échange de documents avec un patient
  - échange de messages avec un patient
  - consulter un questionnaire « santé » d'un patient
  - consulter une bibliothèque de produits médicaux
  - accéder à son bilan

## II. Charte graphique

Deux ensembles de maquette sont fournis (cf annexes).

Ces deux ensembles regroupent les interfaces initialement prévues pour les praticiens et les patients. Chaque ensemble possède une couleur dominante identifiant le type d'utilisateur :

- Pour le patient : vert rgb (117, 223, 157)

The patient interface features a green header bar with a circular profile picture of a smiling man (Paul Dupont) and the text "mon profil". Below the header is a sidebar with links: Accueil, Rendez - vous, Documents, Questionnaire, Messagerie, and a calendar icon. The main content area shows an appointment for "Prochain rendez - vous" on "15/12/2018" with "DR.JEAN". It includes icons for phone, email, and location. A button labeled "Rendez - vous" is also present. The "Notifications" section lists messages from DR.JEAN, DR.PAUL, and DR.CAMILLE, each with a timestamp (13/04/2017).

- Pour le praticien : bleu rgb (20, 121, 255)

The practitioner interface features a blue header bar with a circular profile picture of a smiling man (Paul Dupont) and the text "Mon profil". The sidebar includes links: Dossier patient, Mes rendez-vous, Messages, Documents types, Prescriptions, Bibliothèque produit, Bilan financier, and a logo for "TOUS POUR LA SANTE". The main content area shows a "Rendez-vous de la journée" table with columns for "HEURE" and "PERSONNE". Appointments are listed for Nathan M. at 08h30 and Véronique S. at 09h30. The "Notifications" section is divided into "AUJOURD'HUI" and "HIER". In "AUJOURD'HUI", there are messages from Nathan M. and Véronique S. dated 13/04/2017. In "HIER", there is a message from Véronique S. dated 12/04/2017.

Ces maquettes permettront de définir plus précisément l'organisation des fonctionnalités dans le site et la navigation entre les différentes vues. Cependant des modifications à ces maquettes peuvent être apportées au cours du développement de l'application si des solutions d'utilisation plus pertinentes pour l'utilisateur final sont trouvées.

De plus, certaines fonctionnalités ne sont pas représentées dans ces maquettes. Dès lors il sera nécessaire de créer les vues correspondantes en veillant à suivre l'organisation générale et les thèmes utilisateur.

### **III. Cadre technique de l'application**

La partie front-end utilise le framework Angular 2 et Material Angular. Le côté client communique avec un back-end développé avec Spring Boot. Le côté serveur est conçu comme un Service web RESTful et les échanges de données front/back sont au format Json. Pour le stockage des données le SGBD mariaDB sera utilisé. L'application est destiné à la France donc il n'y a pas de stratégie d'internationalisation à prévoir.

### **IV. Prestations attendues**

Les fonctionnalités ayant trait à un « rendez-vous / consultation » pour un patient et un praticien sont, a minima, ce qui doit être produit durant le stage. Les interfaces patient devront être « responsive web design ».

Le développement du projet ayant commencé avant le début du stage, une analyse de l'existant est attendue sur deux points : une analyse de la base de données afin de concevoir un MCD et une analyse des services disponibles rendus par l'API.

## C. Spécification fonctionnelle

### I. Fonctionnalités visiteur

Un visiteur doit pouvoir, à partir de la page d'accueil, accéder au formulaires de connexion ou de d'inscription.

La connexion s'effectue avec un email valide et un mot de passe obligatoirement.

L'inscription se fait avec un nom, un prénom, un numéro de téléphone, un email valide et un mot de passe obligatoire.

A partir de la page d'accueil, un visiteur peut effectuer une recherche afin de trouver un praticien et consulter les disponibilités de celui-ci. Cette recherche doit pouvoir se faire par nom ou par spécialité et localisation.

S'il sélectionne un rendez-vous libre le patient est amené à s'inscrire ou se connecter.

### II. Fonctionnalités patient

Un patient est un utilisateur déjà inscrit et connecté.

Il peut :

- ▶ se connecter : à partir d'un formulaire sur une page dédiée, il s'authentifie. Si l'authentification réussit, le patient est redirigé vers son espace personnel.

À partir du moment où un patient est connecté, toutes les fonctionnalités se font à partir de son espace personnel. Il doit pouvoir :

- ▶ se déconnecter. Si la déconnexion est réussie il est alors redirigé vers le formulaire de connexion.
- ▶ Renseigner ou modifier son profil utilisateur
- ▶ Consulter la liste de ses rendez-vous et annuler des rendez-vous
- ▶ Faire une recherche pour trouver un praticien. Cette recherche doit pouvoir se faire par nom de praticien ou par spécialité et localisation. Une recherche par spécialité et localisation amène l'utilisateur à choisir un praticien dans la liste proposée.
- ▶ A partir de la sélection d'un praticien, consulter les disponibilité d'un praticien et choisir un rendez-vous
- ▶ Échanger des messages avec un praticien via un système de messagerie interne
- ▶ Envoyer des documents à un praticien
- ▶ Consulter les documents envoyés par des praticiens
- ▶ Remplir un questionnaire de santé

### **III. Fonctionnalités praticien**

Un praticien est un utilisateur déjà inscrit en tant que praticien.

Il peut :

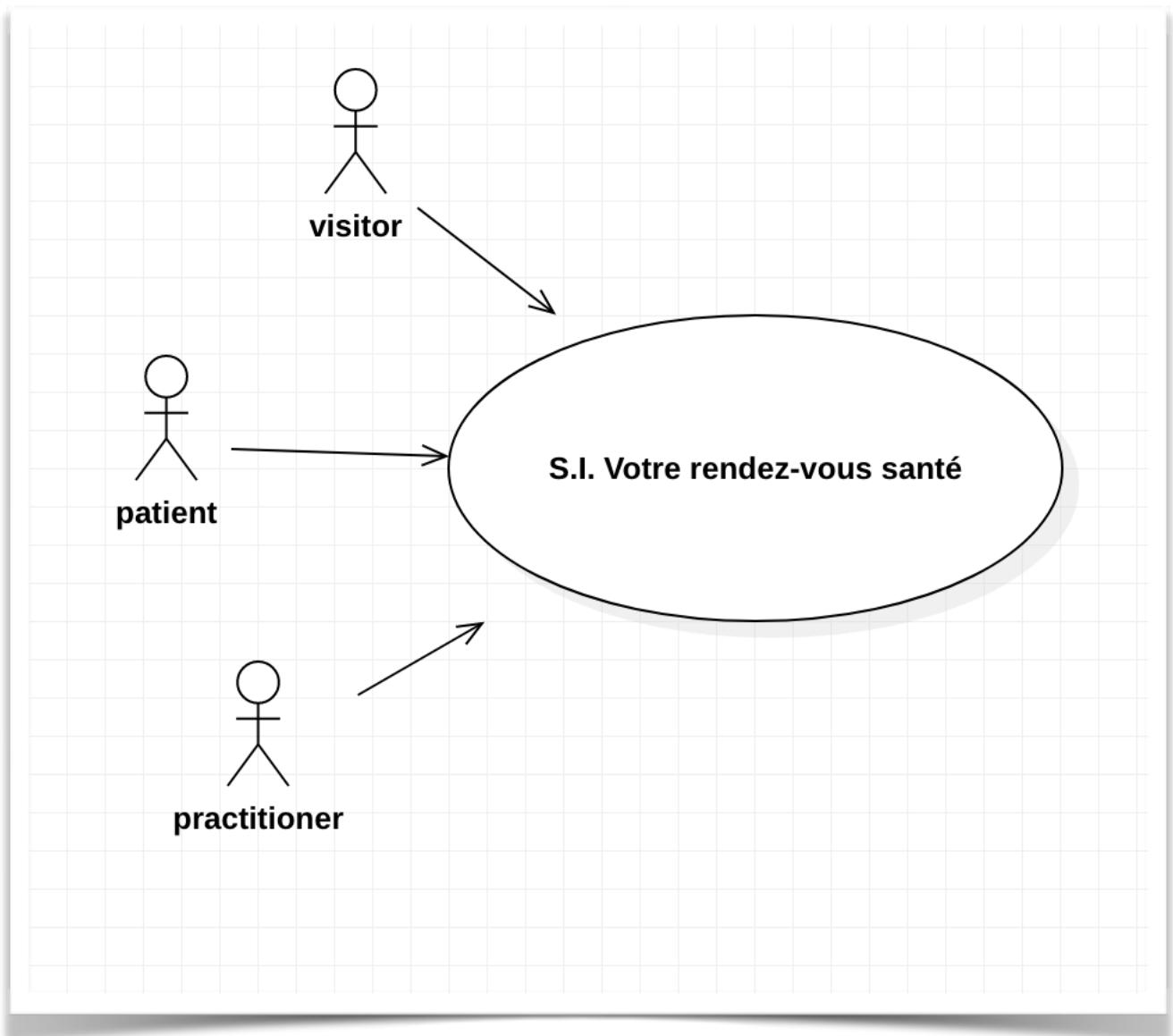
- ▶ se connecter : à partir d'un formulaire sur une page dédiée, il s'authentifie. Si l'authentification est réussie, le praticien est redirigé vers son espace personnel.

À partir du moment où un patient est connecté, toutes les fonctionnalités se font à partir de son espace personnel et il peut dès lors :

- ▶ se déconnecter. Si la déconnexion est réussie il est alors redirigé vers le formulaire de connexion.
- ▶ Modifier ses informations personnelles
- ▶ Organiser son emploi du temps :
  - définir ses spécialités médicales
  - définir les horaires de consultations et les spécialités médicales pour ces horaires
  - Les sujets de rendez-vous possibles pour chacune de ses spécialités
  - La durée des différents types de rendez-vous, ainsi que le temps de pause avant un consultation
  - Créer des indisponibilités ponctuelles
- ▶ Prendre un rendez-vous pour un patient
- ▶ Annuler un rendez-vous. Le patient concerné en est informé par l'envoi d'un message
- ▶ Consulter son planning et voir les détails d'une consultation : horaires de la consultation, spécialité et sujet du rendez-vous, patient concerné et accès au dossier patient
- ▶ Consulter une bibliothèque de produits médicaux
- ▶ Échanger des messages avec des patients via un système de messagerie interne
- ▶ Crédit de documents et envoi de documents à un patient
- ▶ Consulter son bilan financier

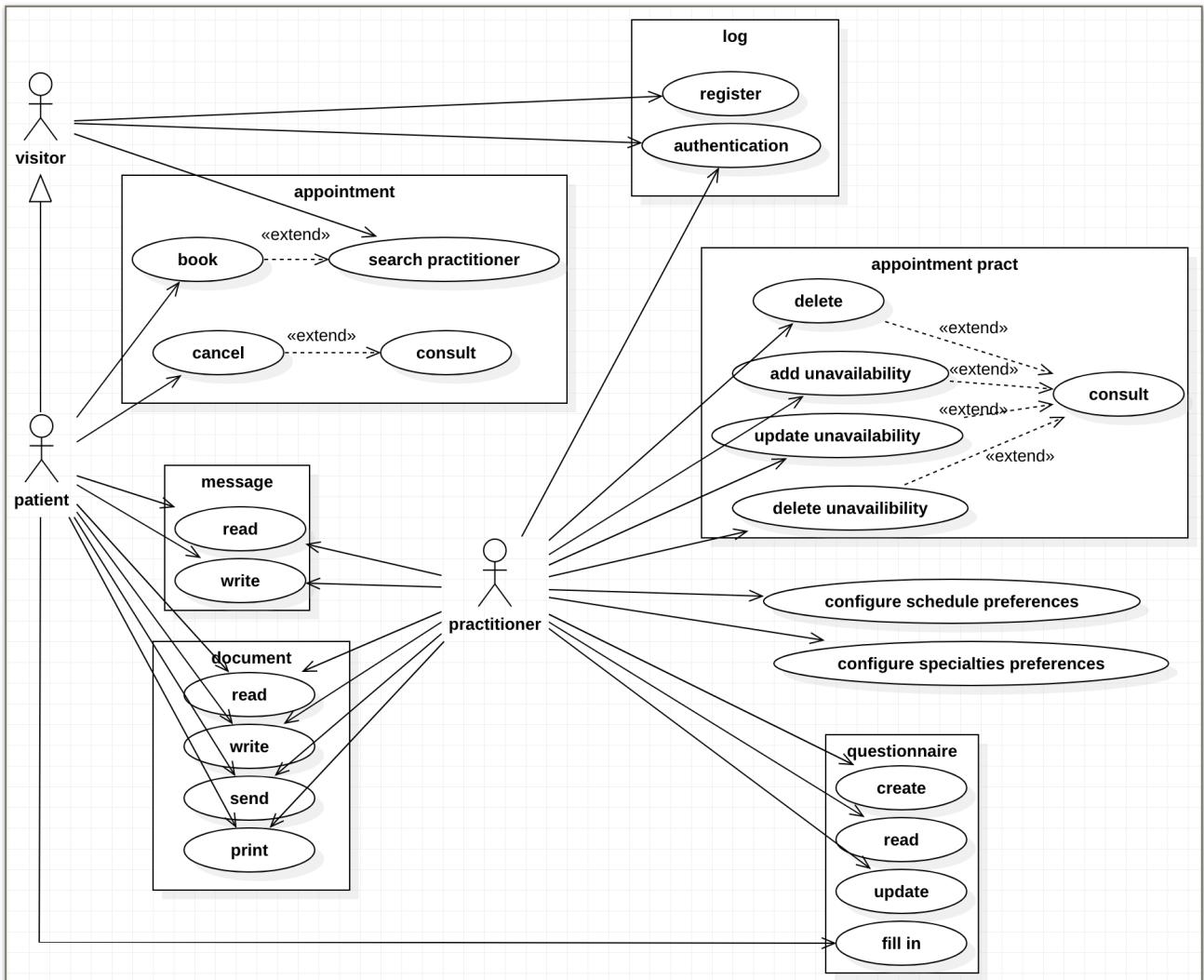
## IV. Analyse des besoins

L'application **Votre Rendez-Vous Santé** sera composé de trois acteurs principaux : *visiteur, patient et praticien*. Nous pouvons alors déterminer le diagramme de contexte suivant :



## II. Use Case

### a. Diagramme use case



### b. Descriptions textuelles de « use case »

→ use case n°1

titre : recherche d'un praticien.

acteur : patient et visiteur.

description : la recherche doit pouvoir être possible pour un patient ou un visiteur.

pré-conditions : l'utilisateur doit être authentifié en tant que patient ; l'utilisateur est sur la page « rendez-vous » de son espace personnel.

version : 1.0

démarrage : le patient a demandé la page « rendez-vous », le visiteur est sur la page d'accueil du site.

### Scénario nominal :

1. Le système affiche la page contenant la liste des spécialités médicales
2. L'utilisateur sélectionne une de ses spécialités médicales
3. Le système recherche les localités dans lesquelles des praticiens pratiquant la spécialité choisie sont présents
4. Le système affiche la liste des localités trouvées
5. L'utilisateur sélectionne une localité et valide la recherche
6. Le système recherche les praticiens en fonction de la spécialité et la localisation
7. Le système affiche la liste des praticiens

### Scénarios alternatifs :

description : aucun praticien n'est trouvé.

Le scénario commence au point 4 du scénario nominal

5.1 L'utilisateur sélectionne une autre localité

Le scénario reprend au point 6

Fin : scénario nominal : à chaque étape d'action par l'utilisateur sur décision de celui-ci.

Post-conditions : aucune.

---

### ➡ use case n°2

titre : réservation d'un rendez-vous

acteur : patient

description : la réservation doit pouvoir être possible pour un patient.

pré-conditions : l'utilisateur doit être authentifié en tant que patient (cas d'utilisation « s'authentifier »).

version : 1.0

démarrage : l'utilisateur a fait une recherche de praticien (use case n°1 « recherche d'un praticien »)

### Scénario nominal :

1. L'utilisateur sélectionne un praticien
2. Le système recherche les sujets de rendez-vous d'un praticien pour la spécialité sélectionnée
3. Le système affiche la liste des sujets de rendez-vous
4. L'utilisateur sélectionne un sujet de rendez-vous
5. Le système recherche la liste des rendez-vous disponibles sur une plage de dates à partir de la date actuelle ou du premier rendez-vous disponible
6. Le système affiche la liste des rendez-vous disponibles
7. L'utilisateur sélectionne un rendez-vous pour le réserver
8. Le système demande la confirmation de réservation
9. L'utilisateur confirme la prise de rendez-vous
10. Le système vérifie la validité du rendez-vous
11. Le système enregistre le rendez-vous

### Scénarios alternatifs :

*description* : le rendez-vous n'est plus disponible entre l'affichage de la liste des rendez-vous et la confirmation du rendez-vous.

11.1 Le système retourne un message informant que ce rendez-vous n'est plus disponible

Le scénario reprend au point 7.

*description* : aucun des rendez-vous affichés par le système ne convient à l'utilisateur.

Le scénario commence au point 6 du scénario nominal

6.1 L'utilisateur sélectionne une autre date

Le scénario reprend alors au point 5 du scénario nominal

Fin : scénario nominal : à chaque étape d'action par l'utilisateur sur décision de celui-ci.

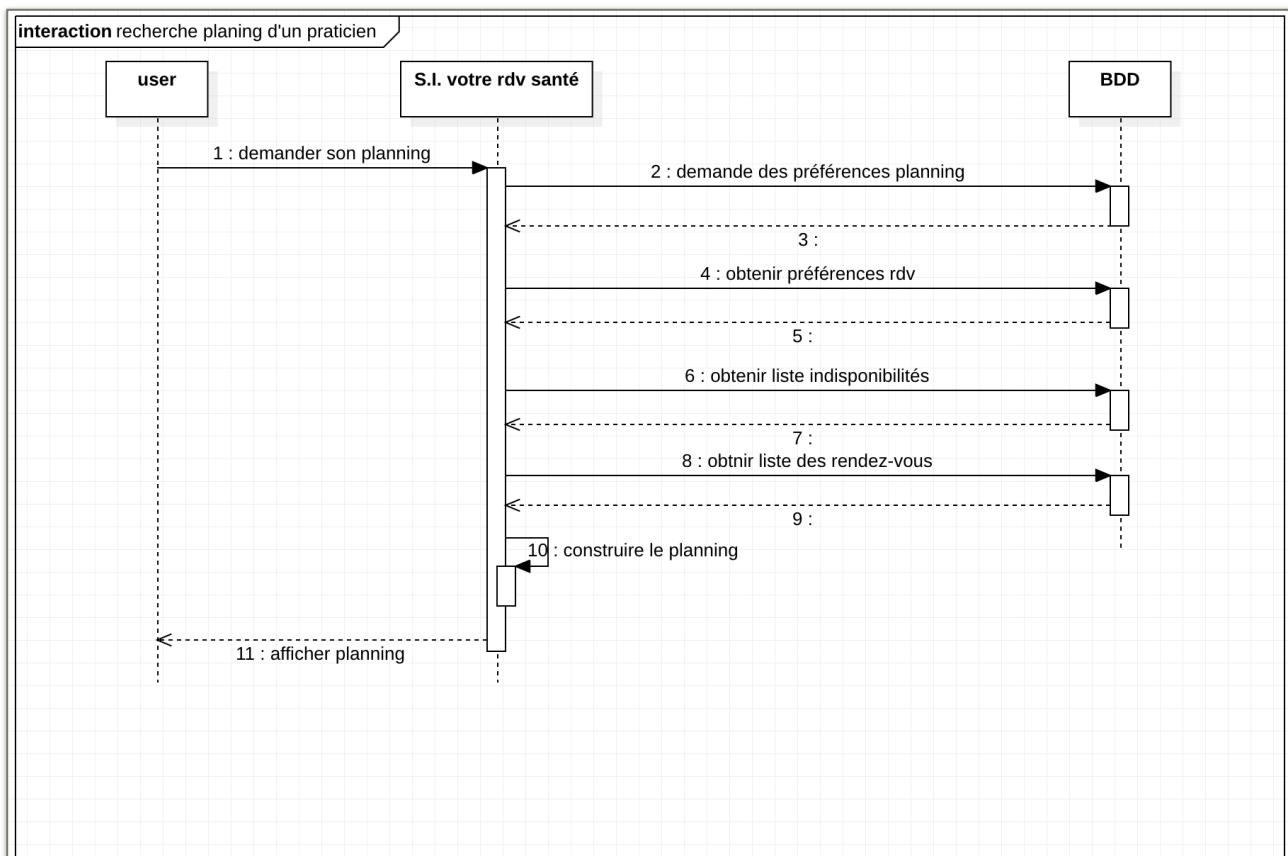
Post-conditions : un message de réussite d'enregistrement est retourné à l'utilisateur et le rendez-vous pris est ajouté à la liste des rendez-vous de l'utilisateur.

---

### III. Diagramme de séquence

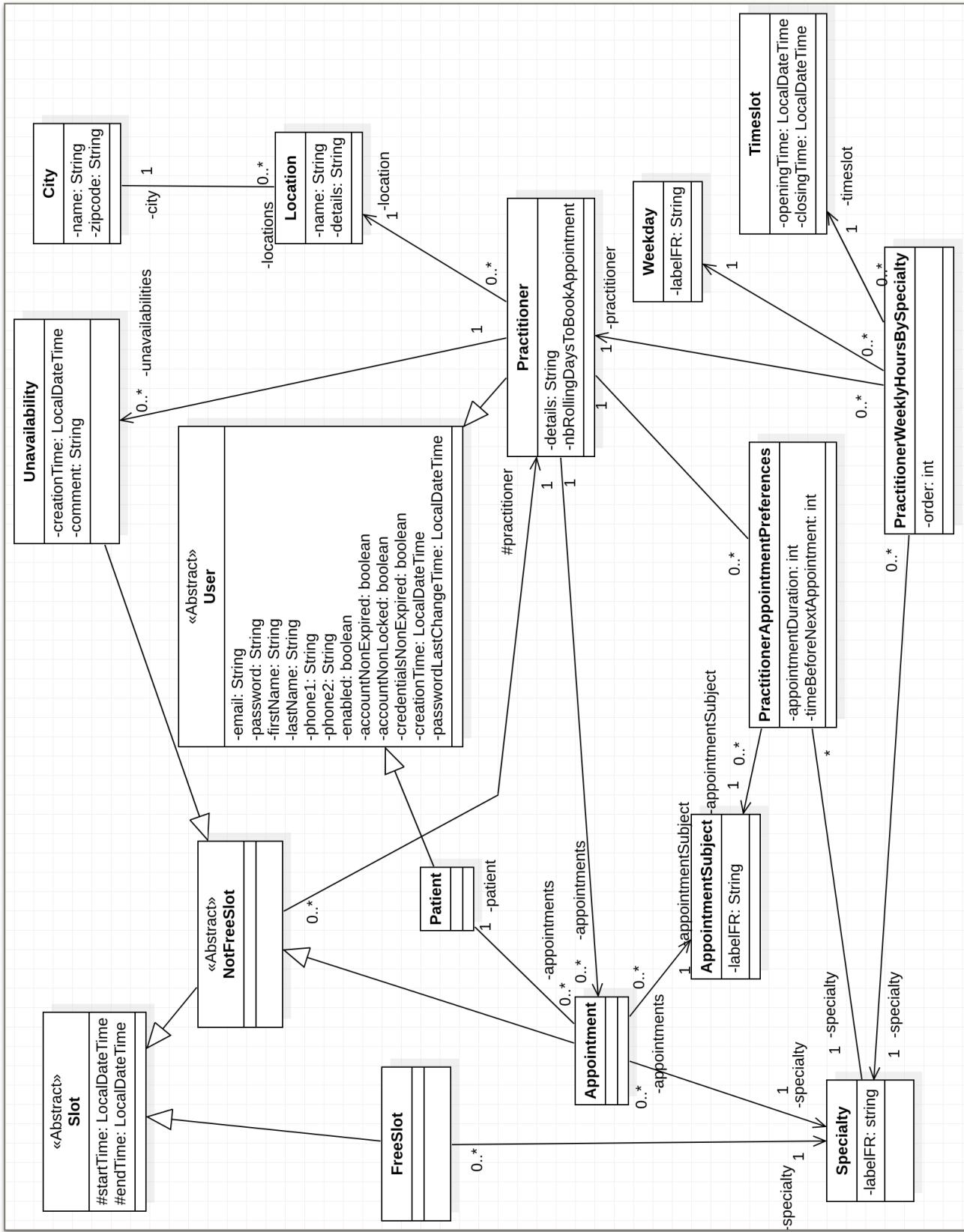
Nous allons voir le diagramme de séquence concernant l'affiche du planning pour un praticien.

Un praticien ne peut voir que son planning, sur lequel est affiché à la fois les rendez-vous pris par des patients, les indisponibilités ponctuelles et les tranches horaires encore libres pour prendre un rendez-vous. Comme ces tranches horaires vides ne sont pas persistées en base de données mais dépendent de plusieurs facteurs (rendez-vous déjà pris, indisponibilités, horaires d'ouvertures, sujet de rendez-vous) il est nécessaire de récupérer les informations qui vont nous permettre de calculer et créer le planning pour l'afficher.



## **IV. Diagramme de classe**

Voici le diagramme de classe théorique de l'application



## IV. Maquettage

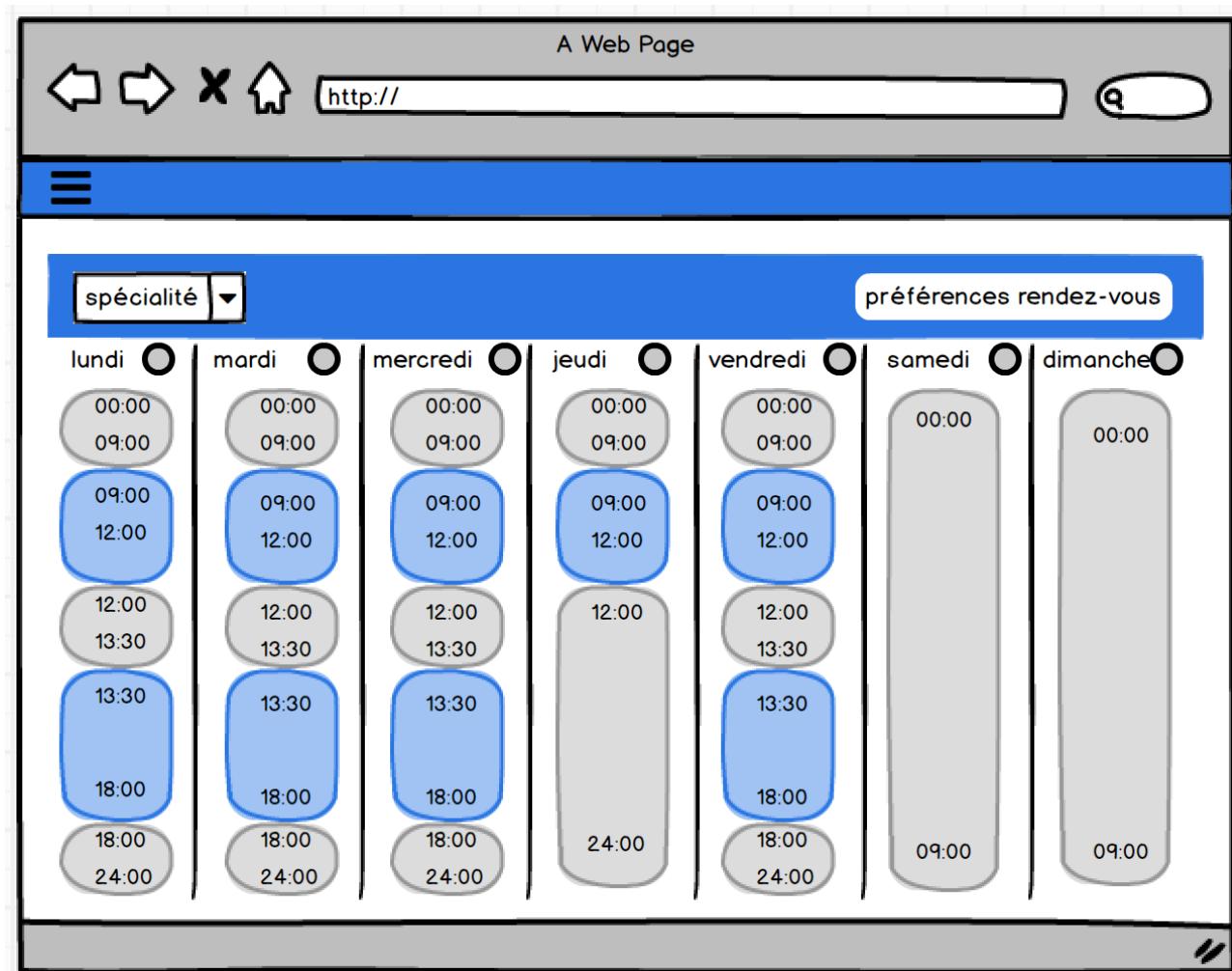
Sur l'ensemble des maquettes fournies en début de stage, certaines fonctionnalités non pas été représentées. Notamment la fonctionnalité permettant à un praticien de paramétriser son planning par défaut.

Création de la vue « configuration planning » (praticien)

L'ensemble des vues « praticien » est pensé pour des écrans de bureau ou tablette uniquement par défaut.

À partir de cette vue un praticien peut fixer des créneaux horaires par spécialité qu'il peut pratiquer.

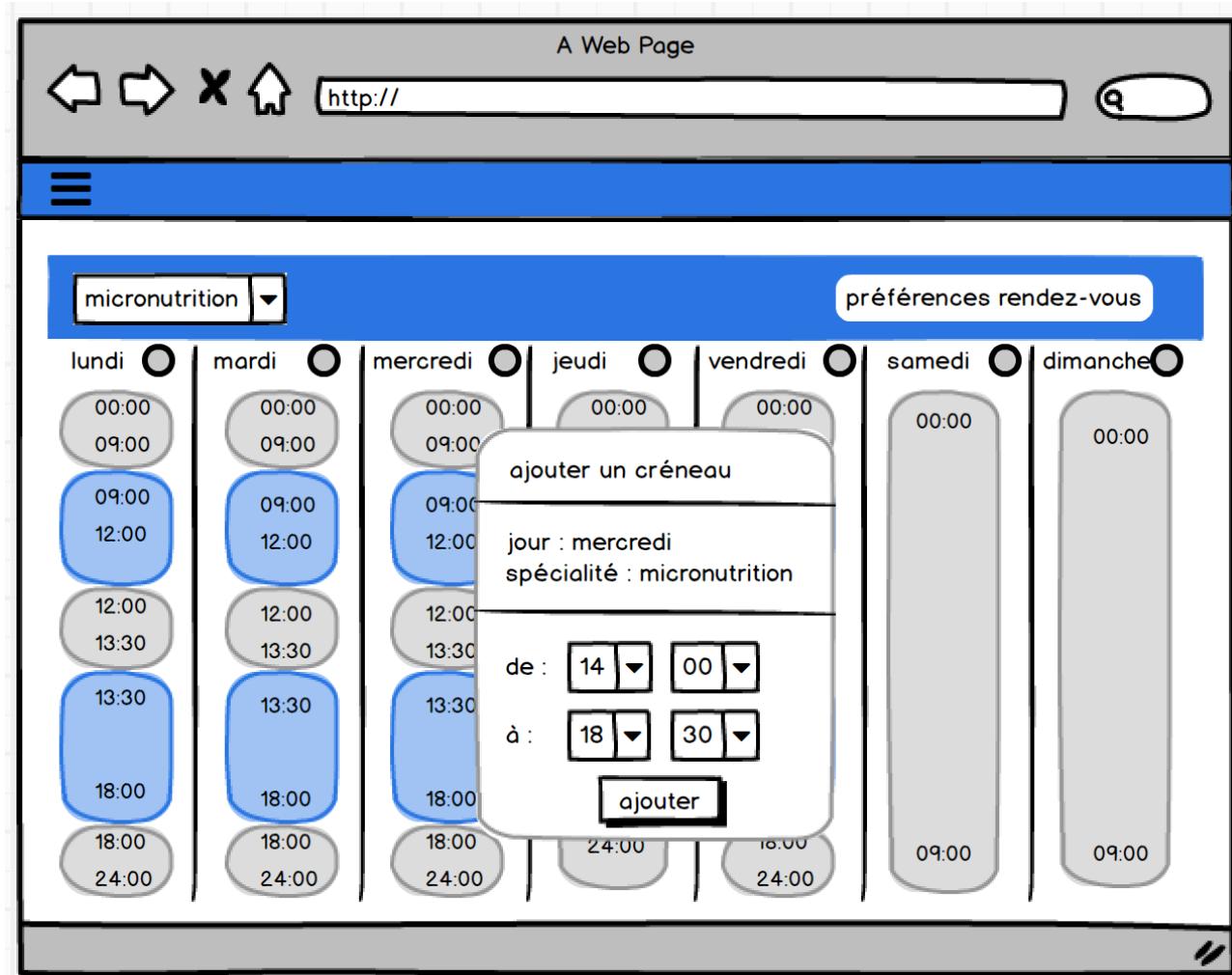
→ Maquette de la vue paramétrage planning d'un praticien :



Le partie centrale représente une semaine type des horaires d'ouverture. Les cadres représentant des créneaux s'étalent sur toutes la hauteur (de minuit à minuit) pour avoir une vision globale. En bleu les horaires d'ouvertures et en gris les temps non ouvrés.

Sur cette page, un utilisateur (un praticien) doit pouvoir choisir grâce à un sélecteur une spécialité médicale parmi celles qui lui sont autorisées. Pour ajouter un créneau horaire à un jour précis il doit cliquer sur un bouton mis en évidence à côté du jour désiré. À ce moment-là une fenêtre s'ouvre proposant au praticien de fixer les horaires d'un nouveau créneau.

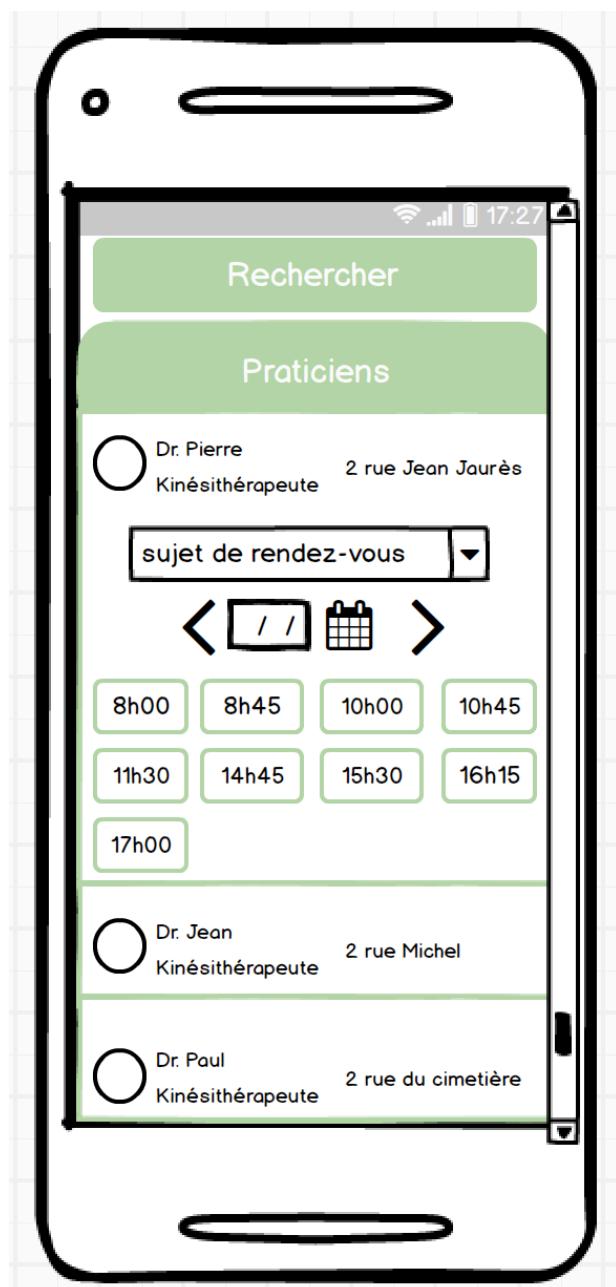
- Maquette de la vue de création d'un nouveau créneau pour une spécialité :



Grâce à des sélecteurs et un bouton le praticien peut spécifier et ajouter un créneau puis à la vue précédente avec un nouvel affichage. Tant que cette fenêtre est ouverte il est impossible d'agir sur le reste de l'application.

Adaptation de la vue prise de rendez-vous « mobile »

Sur les vues « mobile patient » des maquettes fournies, il est suggéré que lorsqu'un utilisateur a choisi un praticien, les rendez-vous disponibles apparaissent sur une vue différente. Finalement il a été décidé que cet affichage pouvait se faire sur la même vue, sous les informations du praticien, sous forme d'accordéon :



## D. Conception

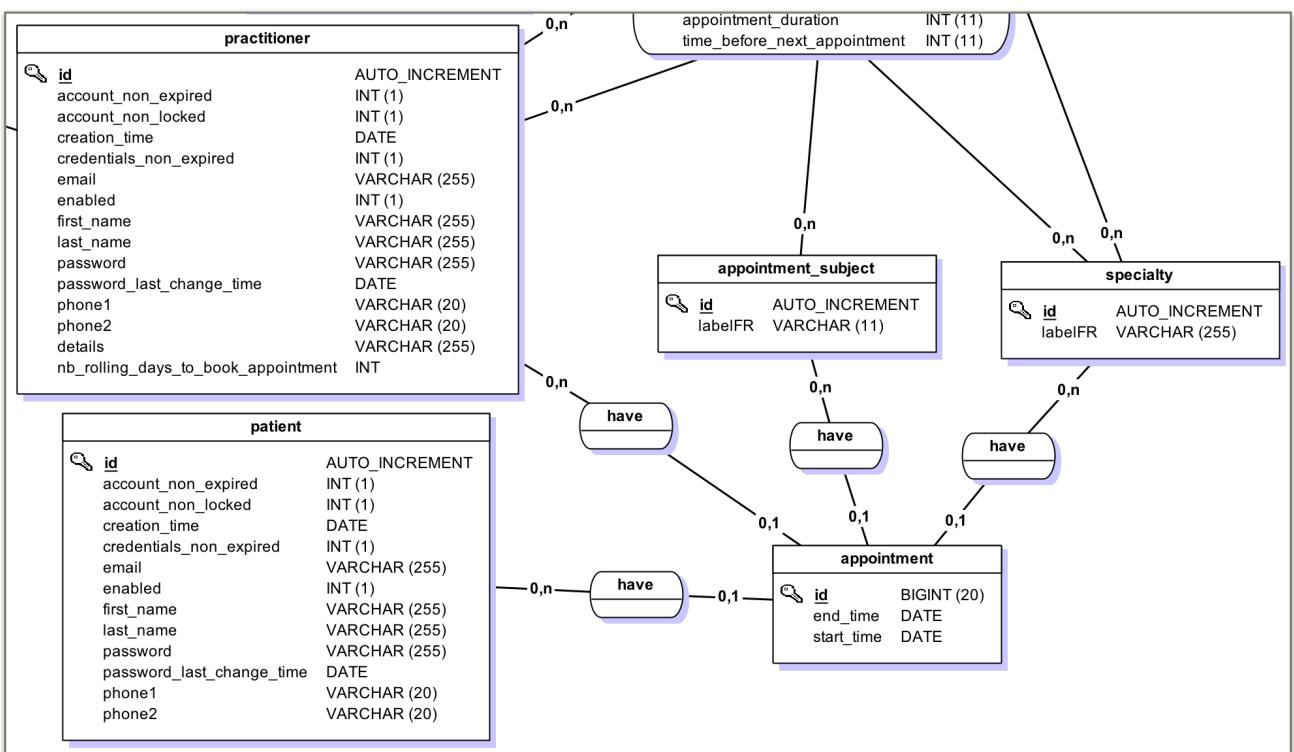
### I. Modèle Conceptuel de Données

Le Modèle Conceptuel de Données (MCD) est une représentation statique des données, il permet de décrire simplement le Système d'Informations à l'aide d'« entités ».

Une entité essentielle de notre MCD est « rendez-vous » (appointment). Un appointment est caractérisé par un id (identifiant unique), une heure de début et une heure de fin. De plus un appointment est lié à un patient, un praticien, une spécialité et doit avoir un sujet de rendez-vous ; chacune de ces références étant déjà une entité, il existe donc une relation « 0,1 » entre un appointment et chacune de ces entités.

De plus :

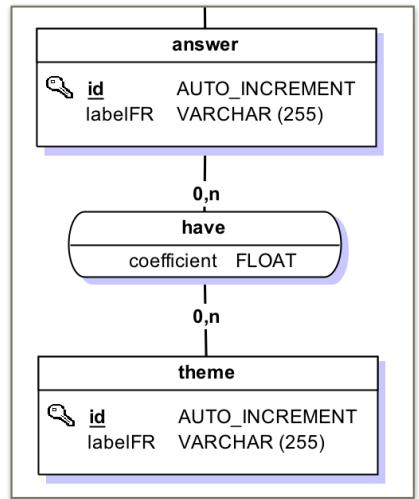
- Un patient peut avoir 0 à N rendez-vous,
- Un praticien peut avoir 0 à N rendez-vous,
- Une spécialité peut être choisie pour 0 à N rendez-vous,
- Une sujet de rendez-vous peut être choisi par 0 à N rendez-vous.



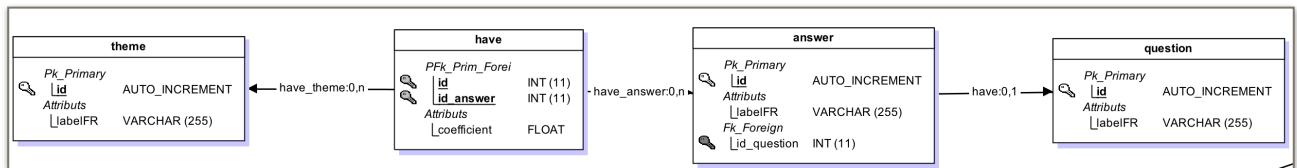
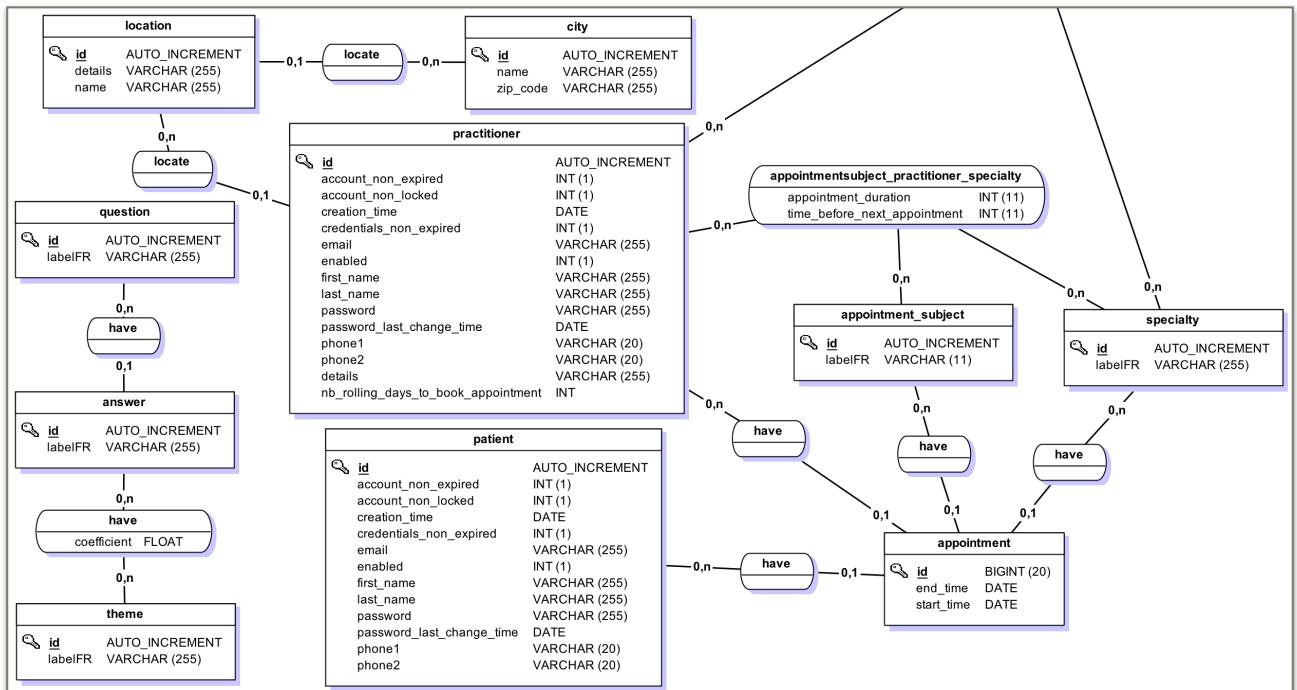
Nous avons également des relations « N, N » entre entité. Par exemple la relation entre theme et answer :

dans cette relation un thème peut avoir aucune ou plusieurs réponse. De même une réponse peut appartenir à un aucun ou plusieurs thèmes. La relation porte un attribut coefficient. Il n'est pas mis dans l'une des deux entités car il est fonction de l'association

entre les deux entités. Par exemple les réponses appartenant à un thème n'auront pas toutes la même importance. Et si une réponse fait partie de plusieurs thèmes, elle n'aura pas forcément le même poids selon le thème.



Voici le MCD en deux parties de l'application moins une partie que ACT ne veut pas dévoiler :



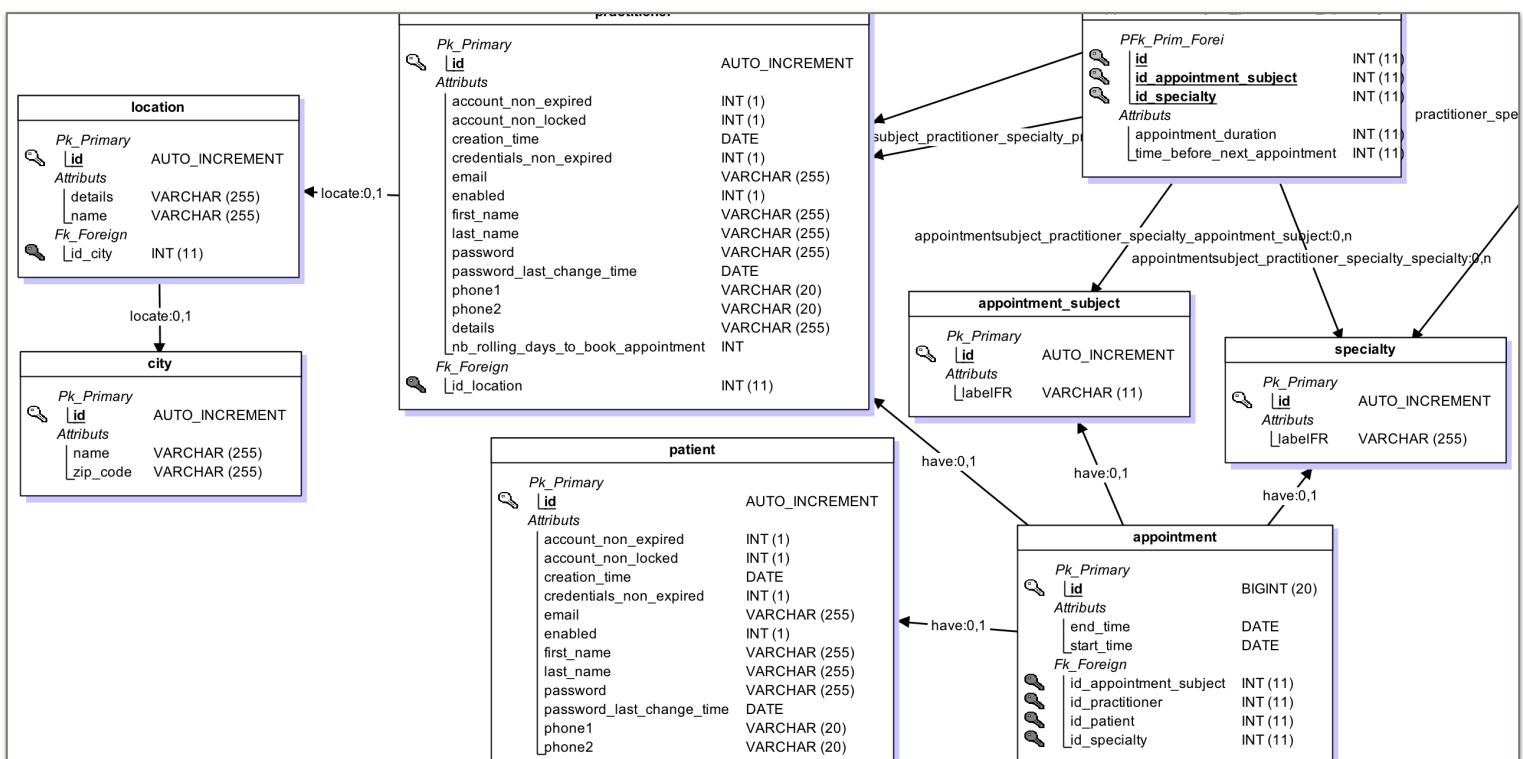
## II. Modèle Logique de Données

Le Modèle Logique de Données (MLD) est une représentation des données en formalisme logique adapté au Système de Gestion de Base de Données envisagé, ici MySQL. C'est l'intermédiaire entre le modèle entité-association et le modèle physique des données. Il est constitué de tables relationnelles, constituées elles-même d'attributs parmi lesquels :

- une clé primaire, identifiant de manière unique chaque occurrence de la table,
- éventuellement une ou plusieurs clés étrangères : ce sont les clés primaires dans une autre table, créant des liens entre les tables

Ainsi :

- L'entité appointment devient la table appointment;
- L'identifiant « id » devient la clé primaire « id »(Pk\_Primary)
- L'association « 1,N » entre appointment et patient devient une clé étrangère « id\_patient » dans la table appointment
- La relation n-aire « appointment\_practitioner\_specialty » devient une table supplémentaire avec trois clés étrangères



### III. Modèle Physique de Données

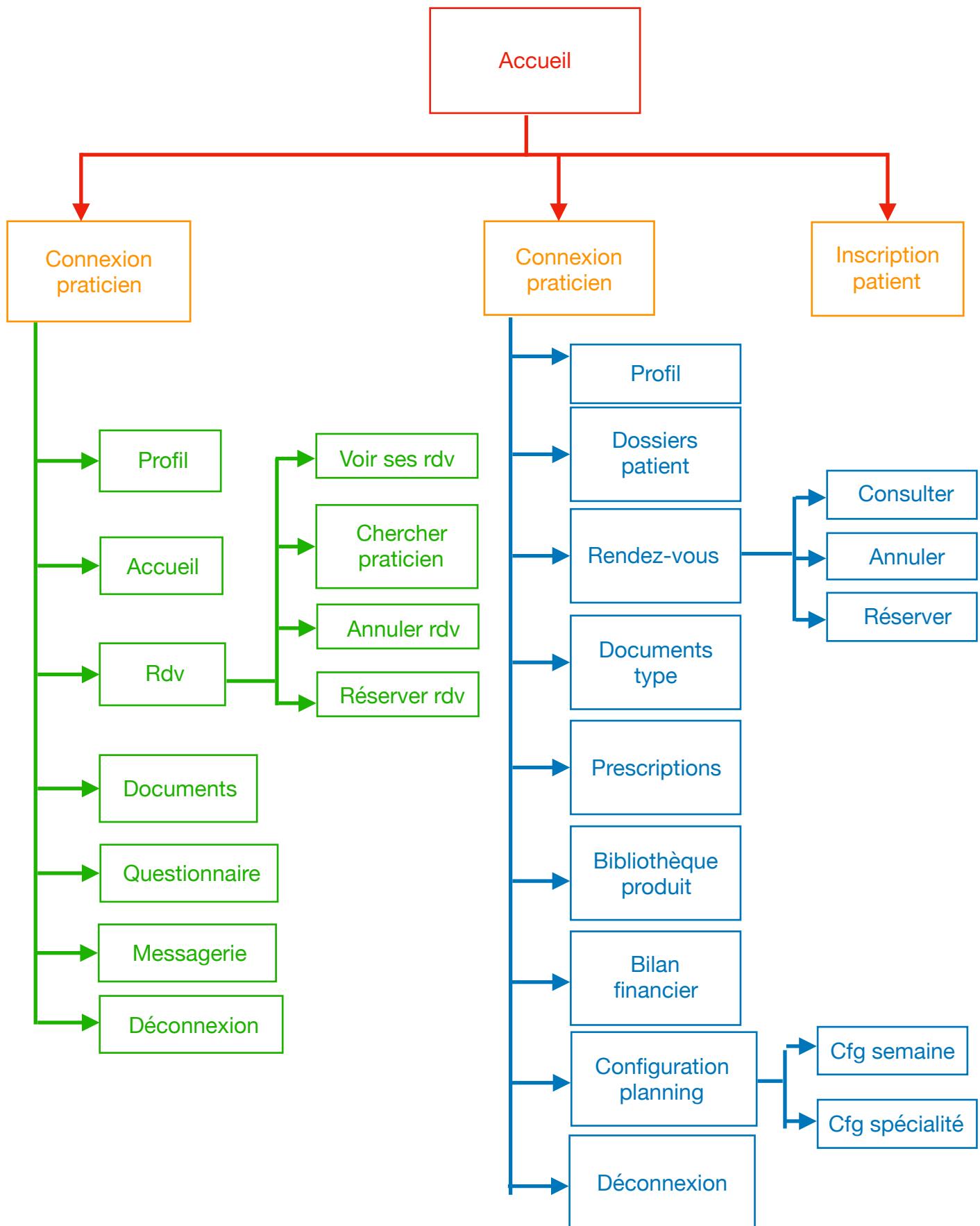
Voici le MPD associé à l'exemple précédent :

attribut = clé primaire,

#attribut = clé étrangère

```
specialty (id, labelFR);
appointment_subject (id, labelFR);
patient (id, email, first_name, last_name,...);
practitioner(id, email, first_name, last_name,..., #location_id);
appointmentsubject_practitioner_specialty(
    #practitioner_id,
    #appointment_subject_id,
    #specialty_id,
    appointment_duration,
    time_before_next_appointment
);
appointment(
    id,
    end_time,
    start_time,
    #appointment_subject_id,
    #practitioner_id,
    #patient_id, specialty_id
);
```

## IV. Arborescence du site



## E. Spécification technique

- **SQL - MAMP - PhpMyAdmin**

Notre système de gestion de base de données est MySQL pour le développement en local.

Le langage SQL nous sert à exploiter notre base de données et a été choisi pour les propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité) des mécanismes de transactions.

PhpMyAdmin est une application web qui nous permet plus facilement de contrôler l'état de notre base de données, de faire des requêtes SQL sans passer en ligne de commande.

- **Spring Boot 2**

C'est un framework créé par Pivotal, conçu pour simplifier le démarrage et le développement d'application Spring grâce à l'auto-configuration et les starters. Nous utilisons les bibliothèques logicielles du JDK 8 et le serveur d'application Tomcat.

- **Hibernate**

C'est un framework gérant la persistance des objets en base de données relationnelles. C'est notre ORM (object-relational mapping) qui sera la couche d'abstraction pour implémenter JPA.

- **Jackson**

c'est une bibliothèque qui permet à partir d'objets Java de générer des données au format Json.

- **Postman**

application pour appeler, tester notre API.

- **Angular 2 et Material Angular**

framework côté client open source basé sur TypeScript. Ne possède pas de contrôleur. TypeScript est un sur-ensemble D'ECMAScript 6, transcompilé en JavaScript pouvant être interprété par n'importe quel navigateur web ou moteur JavaScript.

Material est un module d'intégration qui permet d'obtenir facilement une interface responsive harmonieuse et unie dans le style 'flat' de Google.

- **Git**

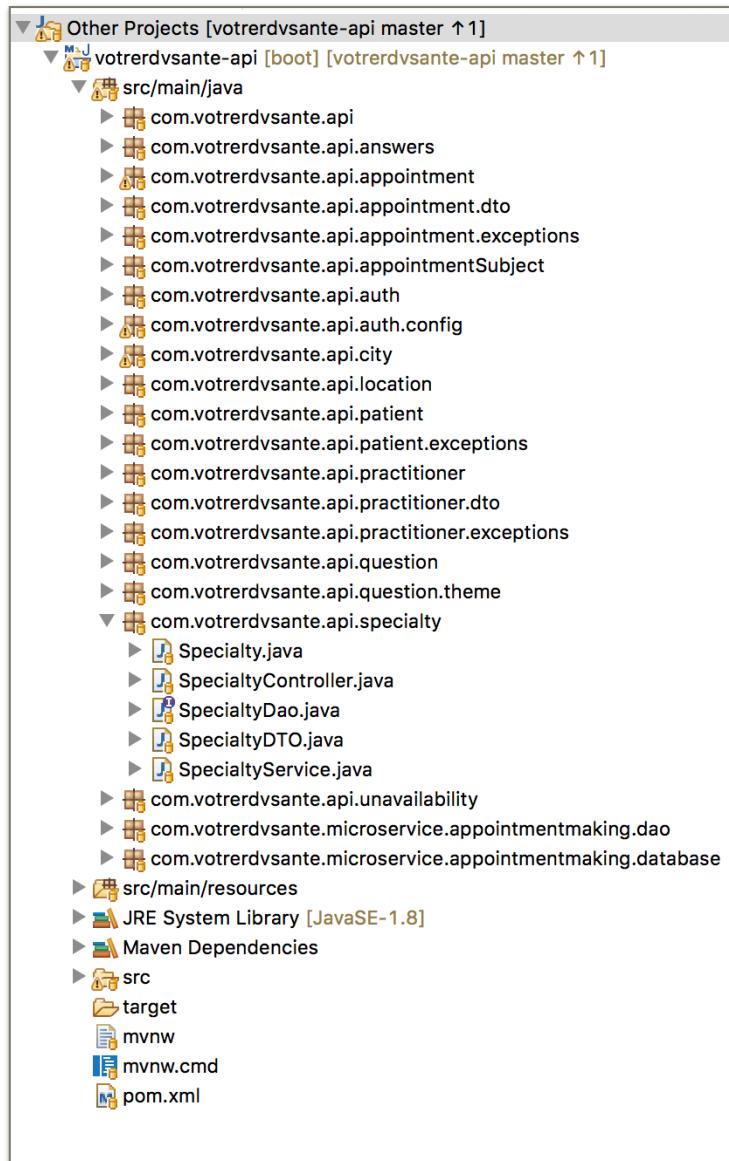
permet de versionner le développement de l'application et de partager en parallèle l'avancement du travail entre plusieurs développeurs.

## F. Réalisation

### Organisation des fichiers de travail côté Back-end

L'application vise à terme, en fonction du nombre d'utilisateur, à être un ensemble de Microservices. Cependant dans la phase de développement et de test, la mise en pratique du principe d'architecture Microservices est représentée au niveau de l'organisation des packages.

Ainsi chaque package est spécialisé dans une seule tâche et comprend les fichiers nécessaires pour effectuer cette tâche.



Par exemple le package « `com.votrerdvsante.api.specialty` » a pour rôle de traiter spécifiquement les informations sur les spécialités médicales. C'est-à-dire que seul ce service connaît comment sont traitées les données et à quelles données sur les spécialités il est possible d'accéder. Dans le cas où un service à besoin d'informations complémentaires qu'il ne connaît pas, il peut également contacter le service qui s'en occupe.

## Fichier Properties

Le fichier « *application.properties* » est un fichier dans lequel il est possible de changer un grand nombre de paramètres et personnaliser l’auto-configuration. Par exemple nous avons choisi de changer le port du serveur :

```
server.port 9092

spring.datasource.url=jdbc:mysql://localhost:8889/votrerdvsante?useUnicode=true&use
spring.datasource.username=*****
spring.datasource.password=*****
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.open-in-view=false
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=false
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect

spring.session.store-type=jdbc
spring.session.jdbc.initialize-schema=always
spring.session.timeout=900
```

Du coup notre API a comme adresse : <http://localhost:9092>

Nous y avons aussi configuré les paramètres de connexions au SGBD.

## Démarrage de l’application

Tout d’abord comme l’agencement des packages de l’application simule un ensemble de microservices il n’y a qu’un seul point de démarrage pour toute l’application. Cette classe est générée automatiquement par Spring Boot :

```
1 package com.votrerdvsante.api;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7
8 public class VotreRdvSanteApi {
9
10    public static void main(String[] args) {
11        SpringApplication.run(VotreRdvSanteApi.class, args);
12    }
13 }
```

L'annotation `@SpringBootApplication` encapsule 3 autres annotations :

- `@Configuration` : permet à la classe de définir des configurations via des beans.
- `@EnableAutoConfiguration` : permet le mécanisme d'auto-configuration de Spring Boot.
- `@ComponentScan` : permet de scanner dans le package dans lequel l'application est localisée afin de trouver les Beans de configuration.

### Génération de la base de données

La base de données est dans le cas de notre application générée par Spring grâce principalement à l'utilisation de Java Persistence API (JPA). L'utilisation pour la persistance d'un Object/Relational Mapping va nous permettre d'assurer la transformation d'objets vers la base de données et vice versa, que cela soit pour les lectures ou des mises à jour (création, modification ou suppression).

Une entité va représenter une table dans une base de données relationnelles, et chaque instance de l'entité correspond à un tuple de la table. Cette entité est un simple POJO (Plain Old Java Object) nommé Entity bean.

Nous allons prendre en exemple la classe `Appointment` pour illustrer les annotations qui nous ont servi à organiser nos données pour le mapping avec la BDD.

Nous avons donc `@Entity` pour déclarer que notre bean va être une entité, `@Id` pour définir un identifiant et un constructeur vide. C'est le minimum requis. `@GeneratedValue` va permettre la génération automatique de cette clé primaire.

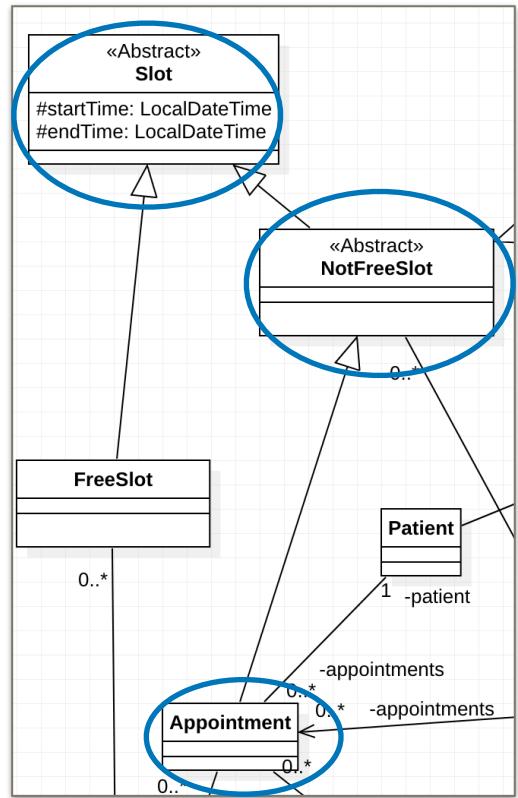
```
    @Entity
public class Appointment extends NotFreeSlot{
    @Id
    @GeneratedValue(
        strategy= GenerationType.AUTO,
        generator="native"
    )
    @GenericGenerator(
        name = "native",
        strategy = "native"
    )
    private long id;

    public Appointment() {
        super();
    }
}
```

The code block shows the `Appointment` class definition. Four annotations are highlighted with red arrows pointing to them:

- `@Entity`: Points to the first line of the class definition.
- `@Id`: Points to the `@Id` annotation within the class definition.
- `@GeneratedValue`: Points to the `@GeneratedValue` annotation with its parameters (`strategy= GenerationType.AUTO` and `generator="native"`).
- `@GenericGenerator`: Points to the `@GenericGenerator` annotation with its parameters (`name = "native"` and `strategy = "native"`).

D'après le diagramme de classe vu précédemment, ce bean hérite de la classe abstraite NotFreeSlot, héritant elle même d'une autre classe abstraite, Slot. Ces deux classes possèdent des attributs dont la visibilité est protected donc Appointment aura accès à ces attributs en plus de ceux qui composent la déclaration de la classe.



Pour que Hibernate puisse prendre en considération les attributs hérités nous avons ajouté à ces deux classes l'annotation @MappedSuperClass :

```
@MappedSuperclass
abstract class Slot {
```

```
@MappedSuperclass
public abstract class NotFreeSlot extends Slot {
```

Notre classe est complétée également par l'annotation @Table qui va nous permettre de spécifier explicitement le nom de la table que l'on veut utiliser ainsi que d'autres informations, en l'occurrence ajouter une contrainte sur la table : l'association des valeurs des colonnes practitioner\_id et startTime devra être unique dans toute la table, c'est à dire l'impossibilité d'avoir deux consultations avec le même jour et horaire de début par praticien.

```
@Table(
    name = "appointment",
    uniqueConstraints = @UniqueConstraint(
        columnNames = {
            "practitioner_id",
            "startTime"
        }
    )
)
```

Quant à l'utilisation de @JoinColumn, nous définissons par cette annotation le nom de la colonne de la table liée au bean. Par exemple, au lieu d'avoir une colonne patient nous aurons patient\_id.

Selon le MLD de notre application nous devons intégrer les clés étrangères dans les tables de la base de données, gérer les relations entre nos différentes tables donc

de nos Entity Bean. Dans JPA nous utilisons `@ManyToOne` et `@OneToMany` dont nous allons voir l'implémentation avec la relation bi-directionnelle Appointment/Patient.

Dans le bean `Appointment`, l'attribut patient (dont le type est Patient) est annoté par `@ManyToOne`. À l'inverse dans le bean `Patient`, l'attribut `appointments` est lui annoté par `@OneToMany`. On pourrait le traduire par « un patient a plusieurs rendez-vous ».

Puisque la relation est bi-directionnelle nous devons préciser dans l'entité qui utilise `@OneToMany` (grâce à `mappedBy`) le nom de l'attribut qui porte la relation.

```
public class Appointment extends NotFreeSlot{
    private long id;..  

  
    @ManyToOne  
    @JoinColumn(name="patient_id")  
    private Patient patient;  

  
    @ManyToOne  
    @JoinColumn(name="specialty_id")  
    private Specialty specialty;  

  
    @ManyToOne  
    @JoinColumn(name="appointmentssubject_id")  
    private AppointmentSubject appointmentSubject;  

  
    public Appointment() {  
        super();  
    }
```

```
@Entity  
@JsonFilter("filter-Patient")  
public class Patient extends User {  
  
    private static final long serialVersionUID =  
        SpringSecurityCoreVersion.SERIAL_VERSION_UID;  
  
    private List<Appointment> appointments;  
  
    @OneToMany(mappedBy = "patient")  
    public List<Appointment> getAppointments() {  
        return appointments;  
    }  
  
    public void setAppointments(  
        List<Appointment> appointments) {  
        this.appointments = appointments;  
    }  
  
    public UserType getUserType() {..}
```

Finalement ces annotations et la ligne « `spring.jpa.hibernate.ddl-auto=create` » dans le fichier application.properties vont générer nos tables en base de données, équivalent au SQL suivant concernant la table appointment par exemple :

```
CREATE TABLE appointment (
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    startTime datetime NOT NULL,
    endTime datetime NOT NULL,
    practitioner_id INT DEFAULT NULL,
    specialty_id INT DEFAULT NULL,
    appointment_subject_id INT DEFAULT NULL,
    patient_id INT DEFAULT NULL
) ENGINE=InnoDB;

ALTER TABLE appointment
    ADD CONSTRAINT myAppConstraint UNIQUE
        (practitioner_id, startTime),
    ADD CONSTRAINT FKspecialty FOREIGN KEY (specialty_id)
        REFERENCES specialty (id),
    ADD CONSTRAINT FKpatient FOREIGN KEY (patient_id)
        REFERENCES patient (id),
    ADD CONSTRAINT FKappointmentsubject FOREIGN KEY (appointmentsubject_id)
        REFERENCES appointment_subect (id),
    ADD CONSTRAINT FKpractitioner FOREIGN KEY (practitioner_id)
        REFERENCES practitioner (id);
```

## Fonctionnalité : « recherche d'un praticien »

Cette fonctionnalité permet d'afficher à l'utilisateur (un patient ou un visiteur) une liste de praticien en fonction d'une spécialité médicale et d'une localité. Cependant pour voir l'architecture générale de l'application je ne vais décrire que la séquence concernant la récupération de toutes les spécialités médicales

Nous allons d'abord voir comment accéder à un service de l'API, puis comment sont récupérées les données et enfin la réponse du serveur.

Puisque chaque service (package) de notre serveur Back doit pouvoir exposer une API REST il est nécessaire de définir les points d'accès aux données dans une unique classe qui sert de contrôleur. C'est cette couche qui contrôle les données entre la vue et le modèle et transforme le langage de la vue en celui du modèle.

C'est l'annotation @RestController au-dessus de la déclaration d'une classe qui dit à Spring que cette classe joue un rôle spécifique. Ici, notre classe est un Web @Controller et Spring considère qu'elle peut gérer des requêtes Web.

```
13 @RestController
14 @CrossOrigin(origins = { "http://localhost:4200", "http://127.0.0.1:4200" })
15 public class SpecialtyController {
16
18+     private SpecialtyService specialtyService;[]
19
21+     * get all specialties[]
25+     public MappingJacksonValue getSpecialties() {[]
29
30+     private MappingJacksonValue applySpecialtyFilter(MappingJacksonValue json) {[]
50 }
51 }
```

À présent il nous faut définir la méthode exposée par l'API. Comme nous voulons demander au serveur de nous retourner une ressource, nous utiliserons une méthode GET du protocole HTTP pour contacter notre API. C'est l'annotation @GetMapping qui fait le lien entre une requête et une méthode. Le paramètre « value » indique l'URI à laquelle cette méthode doit répondre.

```
@GetMapping(value = "/specialties")
public MappingJacksonValue getSpecialties() {
    MappingJacksonValue res = new MappingJacksonValue(specialtyService.findAll());
    return applySpecialtyFilter(res);
}
```

Voici l'adresse complète pour utiliser la méthode getSpecialties :

<http://localhost:9092/getSpecialties>

Au delà de ce point de notre contrôleur, notre vue ne sait pas ce qui se passe pour respecter une architecture n-tiers. C'est notre contrôleur qui communique avec la couche suivante : un service, contenant la logique métier de notre modèle (« specialtyService » dans l'illustration précédente).

L'utilisation d'une instance de classe « SpecialtyService » se fait dans notre cas par l'annotation @Autowired qui fait de l'injection de dépendance, mécanisme primordial dans Spring. Donc Spring cherche et injecte SpecialtyService quand SpecialtyController est instancié, specialtyService devenant un attribut du Controller.

```
@Autowired  
private SpecialtyService specialtyService;
```

La classe SpecialtyService est décorée par @Service, ce qui permet au framework de détecter la classe. On retrouve aussi l'annotation @Autowired qui cette fois va injecter un SpecialtyDao dans notre service.

```
9  @Service  
10 public class SpecialtyService {  
11  
12     @Autowired  
13     private SpecialtyDao specialtyDao;  
14
```

Le modèle DAO (Data Access Object, objet d'accès aux données) est un modèle qui propose de regrouper les accès aux données persistantes dans des classes à part. Donc seule notre classe SpecialtyDao aura ce rôle.

```
1 package com.votrerdvsante.api.specialty;  
2  
3 import java.util.List;  
4  
5 import org.springframework.data.jpa.repository.JpaRepository;  
6 import org.springframework.stereotype.Repository;  
7  
8 @Repository  
9 public interface SpecialtyDao extends JpaRepository<Specialty, Long>{  
10  
11     List<Specialty> findBylabelFRContaining(String match);  
12  
13 }  
14
```

Cette fois-ci nous utilisons `@Repository` pour indiquer à Spring qu'il s'agit d'une classe qui gère les données. De plus notre DAO hérite de l'interface `JpaRepository`, du framework Spring data JPA, offrant une liste de méthodes pour générer toutes sortes d'opérations vers la base de données. Le premier paramètre entre les chevrons correspond à l'entité concernée, le second correspond au type de l'id.

Parmi les opérations qu'offre `JpaRepository`, nous trouvons `findAll()`, qui permet de récupérer toutes les données de l'entité concernée. Cette méthode nous retourne un itérable que nous devons spécifier comme type de retour de la méthode : `List<Specialty>`.

Du coup, dans notre service, nous appelons cette méthode :

```
public List<SpecialtyDTO> findAll() {
    List<Specialty> lst = specialtyDao.findAll();
    List<SpecialtyDTO> lstDTO = this.convertToDTO(lst);
    return lstDTO;
}
```

À ce moment notre service a récupéré les données désirées et pourrait les retourner à son tour directement au Controller. Cependant nous avons préféré que lors des consultations des ressources notre serveur utilise le patron de conception DTO (Data Transfert Object). Ce pattern va nous permettre d'isoler le modèle du domaine de la présentation. En l'occurrence il nous a permis facilement de renommer un attribut. Il peut nous permettre également de transférer ou non, filtrer des données en fonction de la vue.

SpecialtyDTO vs Specialty :

```
public class Specialty implements Comparable<Specialty>{  
    private long id;  
    private String labelFR;  
    private Set<PractitionerAppointmentPreferences> practitioners;  
  
    public Specialty() { }  
  
    public Specialty(long id, String labelFR) {  
        this.id = id;  
        this.labelFR = labelFR;  
    }  
}
```

```
public List<SpecialtyDTO> convertToDTO(List<Specialty> lst) {  
    List<SpecialtyDTO> lstDTO = new ArrayList<SpecialtyDTO>();  
    for (Specialty specialty : lst) {  
        lstDTO.add(new SpecialtyDTO(specialty));  
    }  
    return lstDTO;  
}
```

```
public class SpecialtyDTO {  
    private Long id;  
    private String label;  
  
    public SpecialtyDTO(Specialty o) {  
        this.id = o.getId();  
        this.label = o.getLabelFR();  
    }  
}
```

C'est dans la boucle ForEach de la méthode convertToDTO que le transfert s'effectue : lors de l'instanciation d'un nouveau SpecialtyDTO, on détermine dans le constructeur les données que nous voulons garder : dans ce cas la liste de « practitioners » n'apparaît plus dans le DTO. De plus cela nous a permis de renommer « labelFR » en « label ».

Maintenant notre service renvoie une liste de SpecialtyDTO (à la place des données brut de la BDD) au controller.

La dernière chose à faire au niveau du Controller est de transformer nos données au format JSON. Pour ce faire nous utilisons Jackson.

```
24 @GetMapping(value = "/specialties")
25 public MappingJacksonValue getSpecialties() {
26     MappingJacksonValue res = new MappingJacksonValue(specialtyService.findAll());
27     return applySpecialtyFilter(res);
28 }
29
30 private MappingJacksonValue applySpecialtyFilter(MappingJacksonValue json) {
31     SimpleBeanPropertyFilter practitionerPropertyFilter =
32         SimpleBeanPropertyFilter.serializeAllExcept("appointments", "specialties");
33     SimpleBeanPropertyFilter patientPropertyFilter =
34         SimpleBeanPropertyFilter.serializeAll();
35     SimpleBeanPropertyFilter appointmentPropertyFilter =
36         SimpleBeanPropertyFilter.serializeAll();
37     SimpleBeanPropertyFilter specialtyPropertyFilter =
38         SimpleBeanPropertyFilter.serializeAllExcept("practitioners");
39
40     FilterProvider fp =
41         new SimpleFilterProvider()
42             .addFilter("filter-Practitioner", practitionerPropertyFilter)
43             .addFilter("filter-Appointment", appointmentPropertyFilter)
44             .addFilter("filter-Patient", patientPropertyFilter)
45             .addFilter("filter-Specialty", specialtyPropertyFilter);
46
47     json.setFilters(fp);
48     return json;
49 }
```

Avant que le Json soit ajouté au body de la réponse nous appliquons un filtre qui fonctionne grâce à des annotations fournies par Jackson. Avant l'implémentation de DTO, nous avions utilisé différentes stratégies offertes par Jackson pour filtrer les données à transmettre à l'IHM. Grâce à ces filtres nous avions pu à un moment gérer des problèmes de récursions des données. L'implémentation de DTO a été réalisée dans la dernière partie du stage donc le temps a manqué pour enlever tous les filtres et s'assurer que l'API fonctionnerait correctement.

Tout le processus décrit précédemment est le mécanisme de base du modèle en couche mis en oeuvre dans chaque package du serveur backend : un Controller offre une ou plusieurs méthodes à la vue ; ce Controller communique via un Service avec un DAO qui, lui, s'occupe de l'accès à la base de données. Si des données doivent être retournées au client, le service passe les données dans un DTO puis les retourne au Controller, lequel les sérialisent en Json. Enfin une réponse est retournée au client.

## Utilisation de Postman

Postman est un logiciel qui se focalise sur les tests des API. Ainsi il nous permet de tester précisément un service de l'API sans avoir à implémenter une IHM particulière ou effectuer une série de requêtes pour atteindre la fonctionnalité ciblée. Dans Postman nous allons pouvoir, entre autres, déterminer quelle méthode du protocole HTTP nous voulons, paramétrier le Header de la requête, ajouter un body à la requête, vérifier et afficher le body de la réponse d'une requête, sauvegarder une requête pour des tests ultérieurs...

Voici un test de du service pour récupérer l'ensemble des spécialités :

The screenshot shows the Postman application interface. At the top, there are several tabs: 'New', 'Import', 'Runner', 'votreredvsante-api' (selected), 'Invite', and 'Upgrade'. Below the tabs, there are three requests listed: 'GET get cookie', 'POST log in', and 'GET get all specialties'. The third request is highlighted with a red circle and labeled '1'. The URL 'http://localhost:9092/specialties' is shown in the 'Send' button area and is also circled in green and labeled '2'. In the 'Headers' tab, there is a table with one row selected, circled in blue and labeled '3'. The table has columns 'KEY', 'VALUE', and 'DESCRIPTION'. The key 'X-XSRF-TOKEN' has a value of '407c0efd-1af0-4ed5-b336-d32466b56185'. The response section at the bottom shows a JSON array of three objects, each representing a specialty with fields 'id' and 'label'. A large pink rectangle labeled '4' encloses the entire response body table.

| KEY          | VALUE                                | DESCRIPTION |
|--------------|--------------------------------------|-------------|
| X-XSRF-TOKEN | 407c0efd-1af0-4ed5-b336-d32466b56185 |             |
| Key          | Value                                | Description |

```
1 [ { "id": 1, "label": "micronutritionniste" }, { "id": 2, "label": "nutritionniste" }, { "id": 3, "label": "kinésithérapeute" } ]
```

**1 - la méthode de la requête = GET**

**2 - l'URL = <http://localhost:9092/specialties>**

**3 - le header**

**4 - la réponse retourné par notre webservice avec la vue du body**

Nous pouvons voir sur cette illustration que notre requête a fonctionné selon nos attentes grâce au code d'état ( status: 200 OK) et que nous obtenons dans le body les données attendues au format Json. L'analyse du Json nous montre que nous avons un tableau composé de trois objets composés eux-mêmes d'un ensemble de « clé / valeur ».

On peut également vérifier rapidement avec une vue de phpMyAdmin de la table « specialty » que nous avons récupéré tous les tuples, c'est à dire 3 tuples ; notons aussi que le découplage « modèle / vue » s'est passé comme voulu.

The screenshot shows the phpMyAdmin interface for a MySQL database named 'votrerdvdsante'. The current table is 'specialty'. The top navigation bar includes 'Browse', 'Structure', 'SQL', 'Search', 'Insert', 'Export', and 'InnoDB status'. Below the navigation is a green success message: 'Showing rows 0 - 2 (3 total, Query took 0.0008 seconds.)'. The SQL query 'SELECT \* FROM `specialty`' is displayed. A table view shows three rows with columns 'id' and 'labelfr':

|                          |                    | <b>id</b> | <b>labelfr</b>      |
|--------------------------|--------------------|-----------|---------------------|
| <input type="checkbox"/> | Edit  Copy  Delete | 1         | micronutritionniste |
| <input type="checkbox"/> | Edit  Copy  Delete | 2         | nutritionniste      |
| <input type="checkbox"/> | Edit  Copy  Delete | 3         | kinésithérapeute    |

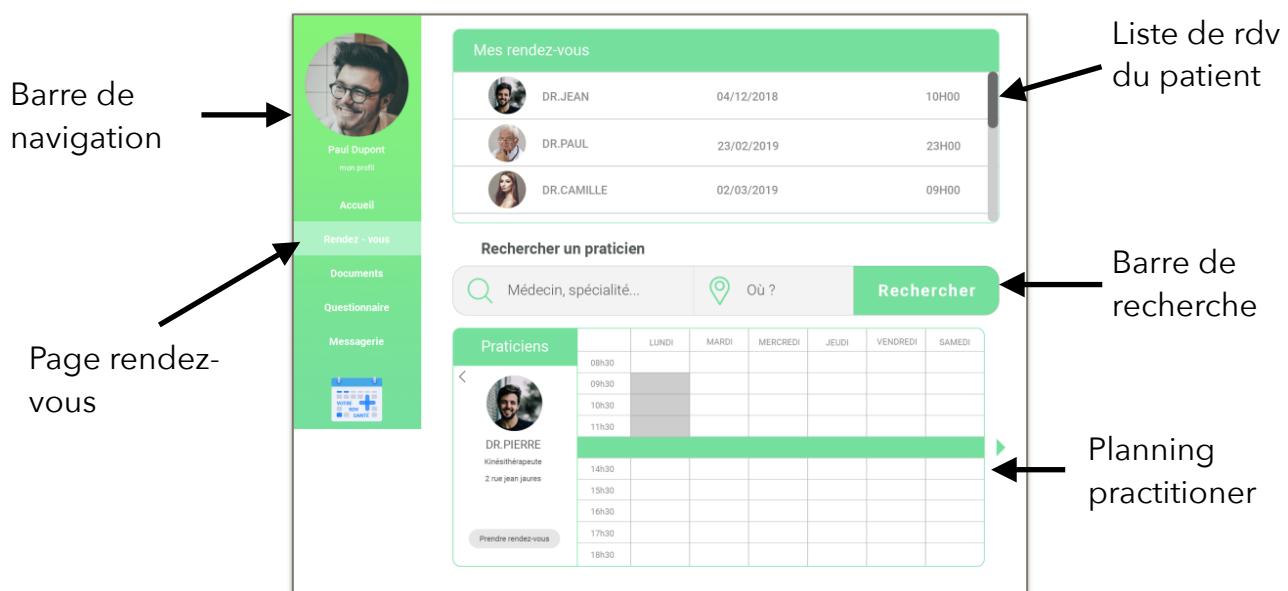
Below the table are buttons for 'Check all', 'With selected:', and various operations like 'Edit', 'Copy', 'Delete', and 'Export'. The bottom section contains links for 'Print', 'Copy to clipboard', 'Export', 'Display chart', and 'Create view'.

## Affichage des données côté client

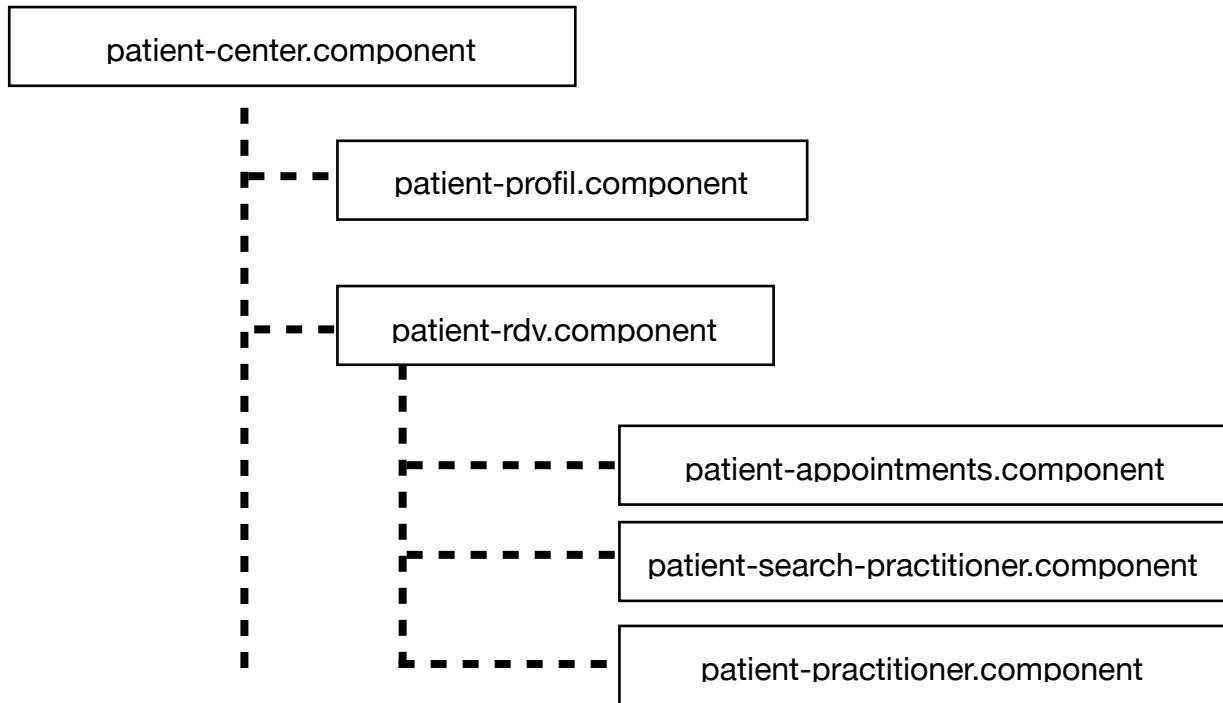
Grâce au webservice nous avons récupéré une liste de spécialités médicales. Chaque élément de cette liste comprend strictement les données qui vont nous être nécessaires : un label compréhensible pour l'utilisateur ainsi que un id. Cet id sera utilisé pour une requête ultérieure vers le serveur afin de récupérer cette fois-ci des localités dans lesquelles un praticien pratique la spécialité sélectionnée.

Pour l'organisation du code côté client je me suis servi des maquettes proposées pour constituer des composants logiques.

D'une part nous avons une page qui contient tout ce qui à trait aux rendez-vous. Cette page est accessible lorsque qu'un patient authentifié se trouve sur son espace personnel à partir d'une barre de navigation donc cette page se trouve au même niveau que d'autres : nous pouvons alors concevoir que chacune de ces pages soit un component à part entière.



D'autre part la vue représentant la page rendez-vous contient une liste de rendez-vous déjà pris, une barre de recherche de praticien et un espace d'affichage de la recherche. Donc nous pouvons concevoir que le component RDV contienne à son tour trois autres components. Ce qui nous donne l'arborescence de components suivant :



C'est dans le composant patient-search.component que nous allons programmer notre recherche.

En premier, nous devons nous occuper de récupérer les informations auprès du webservice. Pour cela nous ajoutons une classe service dans Angular. Toutes les requêtes que nous effecturons se trouveront dans cette classe. Nous avons nommé ce service : *PatientCenterService*

```

16  @Injectable({
17    providedIn: 'root'
18  })
19  export class PatientCenterService { ... }
193
  
```

Le décorateur `@Injectable` déclare le service comme pouvant être injecté. Mais il est nécessaire de configurer en plus un fournisseur de ce service, ici c'est `root`. Grâce à ce décorateur Angular peut injecter la dépendance dans un constructeur d'un autre service ou component. Comme notre component a besoin de ce service nous l'injectons dans le constructeur :

```

7   @Component({
8     selector: 'app-patient-search-practitioner',
9     templateUrl: './patient-search-practitioner.component.html',
10    styleUrls: ['./patient-search-practitioner.component.css']
11  })
12  export class PatientSearchPractitionerComponent implements OnInit {
13
14    constructor(
15      private patientCenterService: PatientCenterService,
16      private patientMessageservice: PatientMessageService
17    ) { }
18

```

Maintenant, dans le component PractitionerSearchComponent, nous pouvons utiliser les méthodes fournies par notre service, notamment la méthode pour récupérer toutes les spécialités médicales :

```

64 /**
65  * get all medical specialties
66 */
67 getMedicalSpecialties(): Observable<MedicalSpecialty[]> {
68   return this.http.get<MedicalSpecialty[]>(this.medicalSpecialtyUrl)
69     .pipe(
70       catchError(this.patientMessageService.handleError<any>())
71     );
72

```

this.http est un attribut obtenu également par injection de dépendance dans notre service même puisque nous l'avons déclaré dans le constructeur : *private http: HttpClient*. La variable http est du type *HttpClient* et sa visibilité est *private*. *HttpClient* est une API d'Angular qui va nous permettre de faire facilement des requêtes vers notre serveur Back-end.

Au final la méthode getMedicalSpecialties() de notre service va retourner un Observable d'un tableau de MedicalSpecialty, classe que nous avons au préalable déclarée :

```

1  export class MedicalSpecialty {
2    id: number;
3    label: string;
4  }
5

```

Cette classe possède deux attributs: *id* et *label*, correspondant exactement au format du Json attendu.

Par défaut la méthode this.http.get<MedicalSpecialty[]> va transformer notre Json en tableau d'objets MedicalSpecialty. L'idée d'un Observable est de pouvoir facilement (dans notre exemple) faire une requête asynchrone : nous ne sommes pas obligés

d'attendre la réponse du serveur pour faire autre chose. Et puisque `HttpClient` retourne un `Observable`, la méthode `getMedicalSpecialties` doit retourner aussi un `Observable`.

Enfin notre component peut appeler et recevoir les données de notre service :

```
27 ① ↗  ngOnInit() {
28      this.getMedicalSpecialties();
29  }
30
31  getMedicalSpecialties(): void {
32      this.patientCenterService.getMedicalSpecialties()
33          .subscribe( next: specialties => this.medicalSpecialties = specialties);
34 }
```

ngOnInit() est une méthode qui fait partie du cycle de vie des components : cette méthode est appelée automatiquement après le constructeur et permet à ce moment de faire des initialisations complexes. Dans notre exemple, à l'initialisation, la méthode getMedicalSpecialties() du component est appelée. Ainsi lorsque l'utilisateur arrive sur la page « rendez-vous » le client va récupérer les spécialités médicales sans attendre une autre action de la part de l'utilisateur.

Par contre il nous est nécessaire de faire souscrire notre méthode à la valeur de retour. Ainsi notre callback nous permet de copier les données retournées dans un attribut de notre component (`this.medicalSpecialties`)

Ne nous manque plus qu'à afficher nos `medicalSpecialties`. Cette partie se fait dans un fichier html lié à notre component : `patient-search-practitioner.component.html`. Ce lien est visible dans la déclaration du component :

```
7  @Component({
8    selector: 'app-patient-search-practitioner',
9    templateUrl: './patient-search-practitioner.component.html',
10   styleUrls: ['./patient-search-practitioner.component.css']
11 })
12 export class PatientSearchPractitionerComponent implements OnInit {
13
14   constructor(
15     private patientCenterService: PatientCenterService,
16     private patientMessageservice: PatientMessageService
17   ) { }
```

Sur la maquette représentant cette fonctionnalité on devine que nous allons devoir utiliser un contrôle qui fournit une liste d'options parmi lesquelles l'utilisateur pourra choisir.

Ayant installé le module Material Angular nous utilisons directement le component select de Material :

Dès lors nous pouvons profiter pleinement de l'API du component MatSelect qui devient un descendant de notre component :

```

4   <mat-form-field appearance="none"
5     class="form-specialty">
6     <mat-icon matPrefix aria-hidden="true"
7       class="material-icons md-48" color="primary">search</mat-icon>
8     <mat-select placeholder="Médecin, spécialité..." 
9       (valueChange)="onSelectMedicalSpecialty($event)" >
10    <mat-option *ngFor="let medicalSpecialty of medicalSpecialties"
11      [value]="medicalSpecialty"
12      class="not-empty-select-specialty">
13      {{medicalSpecialty.label}}
14    </mat-option>
15  </mat-select>
16 </mat-form-field>

```

- Ligne 10 : dans la balise des options, nous faisons une boucle ForEach sur `medicalSpecialties` qui est un attribut de notre component
- Ligne 11 : on passe chaque élément de notre boucle (`medicalSpecialty`) en valeur de nos options grâce au décorateur `@Input` dans `[value]`
- Ligne 9 : lorsqu'une option sera sélectionnée, `valueChange` appellera la méthode `onSelectMedicalSpecialty($event)` qui appartient au component parent, `$event` étant la valeur `medicalSpecialty` de l'option sélectionnée.

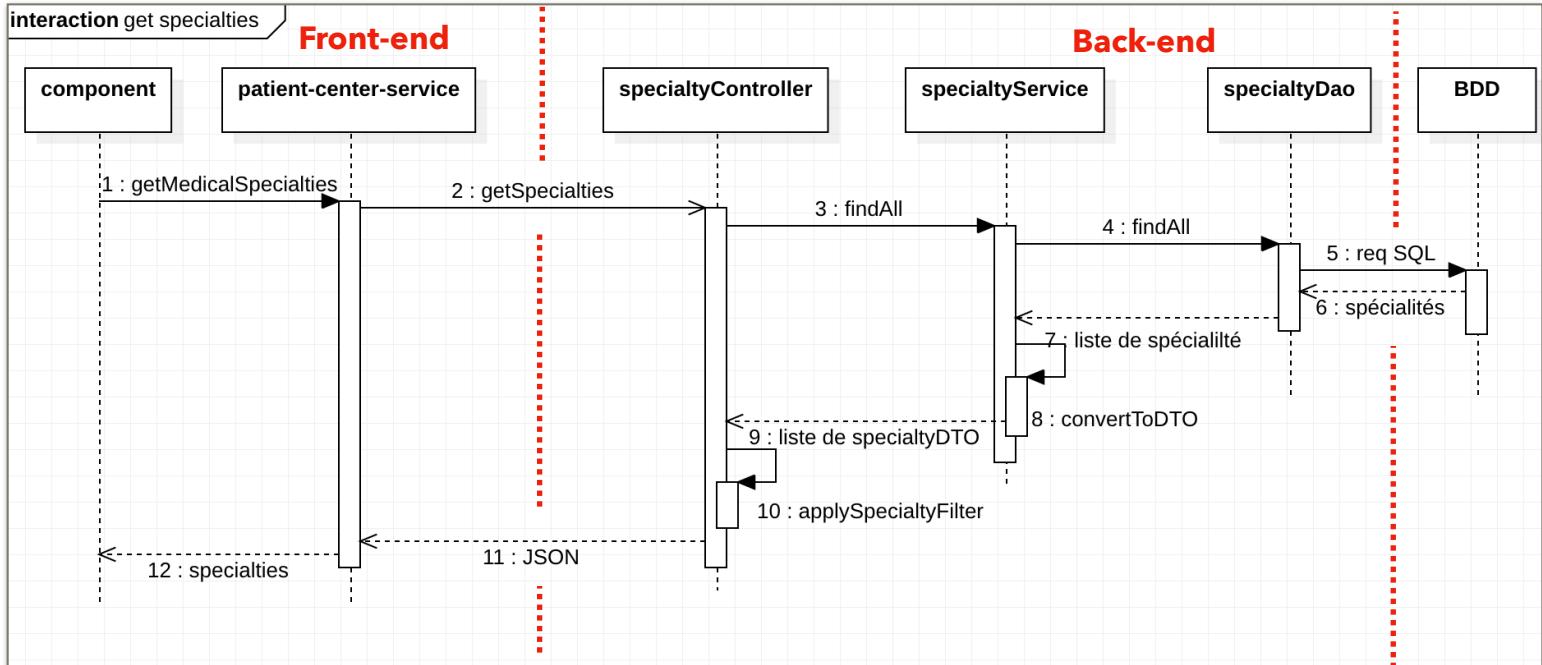
Rendu final :

The screenshot shows a search interface titled "Rechercher un praticien". It has two input fields: one for "Médecin, spécialité..." and another for "Où?". A green "Rechercher" button is located to the right of the second field. The input field for "Médecin, spécialité..." contains a placeholder icon and text.

Lorsqu'on clique dans l'input « médecin, spécialité » :

The screenshot shows the same search interface after clicking on the "Médecin, spécialité..." input field. A dropdown menu has appeared, listing three options: "micronutritionniste", "nutritionniste", and "kinésithérapeute". The "Rechercher" button is still visible to the right of the input fields.

Avec la requête vers notre API et le traitement côté Front avec Angular et Material des informations retournées par l'API, nous venons de voir les mécanismes de base mis en oeuvre sur l'ensemble de l'application. Voici le diagramme de séquence correspondant juste à la récupération des spécialités médicales, qui ne représente que la toute première partie de la fonctionnalité « recherche planning d'un praticien ».



### Fonctionnalité « réservation d'un rendez-vous »

Dans cette fonctionnalité nous allons seulement voir comment sont récupérés les rendez-vous disponibles.

Afin que le serveur puisse nous retourner les rendez-vous disponibles d'un praticien nous avons besoin de faire passer au serveur plusieurs données : quel praticien, pour quelle spécialité, quel sujet de rendez-vous et enfin la période désirée (début et fin).

Nous composons donc l'url qui consommera notre API :

```

@GetMapping(value=
        "/practitioners/{practitioner_id}/freeslots/weekly" +
        "/{from}/{to}/{specialty_id}/{appointmentssubject_id}/{direction}")
public ResponseEntity<?> displayNextFreeSlotsForPatient(
        @PathVariable Long practitioner_id,
        @PathVariable String from,
        @PathVariable String to,
        @PathVariable Long specialty_id,
        @PathVariable Long appointmentssubject_id,
        @PathVariable String direction) {

```

Côté Front, afin de préparer cette méthode GET nous avons stocké des données nécessaires dans une classe service qui gère les requêtes du patient vers le serveur au fur et à mesure du déroulement de la fonctionnalité. Le choix de garder des données dans un service nous permet de les transmettre à des components différents lorsque ceux-ci ont besoin de les afficher.

```
export class PatientCenterService {

    patient: Patient;
    selectedMedicalSpecialty: MedicalSpecialty;
    selectedCity: City;
    selectedPractitioner: Practitioner;
    selectedAppointmentSubject: AppointmentSubject;
    selectedAppointment: AppointmentToStore;
    appointmentBooked = new Event Emitter<Appointment>();
```

Le déclenchement de la requête se fait lorsque l'utilisateur a choisi un sujet de rendez-vous

```
onSelectAppointmentSubject(appointmentSubject: AppointmentSubject): void {
    this.selectedAppointmentSubject = appointmentSubject;
    this.patientCenterService.selectedAppointmentSubject = appointmentSubject;
    this.searchFreeAppointments();
}
```

Après le retour de la réponse du serveur, l'utilisateur a la possibilité de naviguer de semaine en semaine ou bien de choisir une date précise pour trouver un horaire qui lui convient.  
C'est à ce moment que la variable direction nous est utile. Nous avons fait le choix que l'API renvoie toujours un résultat comprenant au moins un rendez-vous libre. Donc il est important de donner le sens de recherche à l'API par rapport à la semaine affichée.

```
searchFreeAppointments(): void {
    this.patientCenterService.searchFreeSlots2(
        this.weekRequest.start,
        this.weekRequest.end,
        this.weekRequest.direction).subscribe(
            next: appointments => {
                this.ready = 'ready-not';
                this.freeslots = [];
                this.freeslots = appointments;
                this.freeslots.sort( compareFn: function(a, b) {
                    return a.from.localeCompare(b.from);
                });
                this.setDayDate();
                this.addAppointmentsToDays();
                this.ready = 'ready';
            }
        );
}
```

```

private getDirection(): string {
    const start = new Date(this.dayWeekRequestStart);
    const end = new Date();
    const milli = start.getTime() + (RANGEDAYSREQUEST - 1) * 24 * 60 * 60 * 1000;
    end.setTime(milli);
    if (this.selectedDay < start) {
        return BACKWARD;
    } else if (this.selectedDay > end) {
        return FORWARD;
    } else {
        return IN;
    }
}

```

Côté serveur, puisque les rendez-vous disponibles ne sont pas stockés en base de données, nous devons d'abord récupérer les données qui vont nous permettre de construire nos rendez-vous.

D'abord on vérifie la validité de la variable direction :

```

/* check if direction is correct value */
if(!(direction.equals(BACKWARD) || direction.equals(FORWARD))) {
    return new ResponseEntity<>("direction", HttpStatus.BAD_REQUEST);
}

```

On récupère auprès de la base un Praticien par l'id grâce à une méthode du repository

```

try {
    practitioner = practitionerService.findById(practitioner_id);
} catch (NoPractitionerFoundException e) {
    return e.getResponseEntity();
}

```

**SQL ->**

```
SELECT * FROM practitioner p WHERE p.id = :practitionerId
```

On récupère auprès de la base les préférences spécialités d'un praticien:

```

rules = paps.getAppointmentPreferences(
    practitioner_id,
    specialty_id,
    appointmentsubject_id);

```

**SQL ->**

```

SELECT *
FROM appointmentsubject_practitioner_specialty aps
WHERE aps.appointment_subject_id = 1
AND aps.practitioner_id = 1
AND aps.specialty_id = 1

```

On récupère auprès de la base les rendez-vous déjà pris du praticien :

```
List<AppointmentPractitionerViewDTO> appointments =  
    appointmentService.findBypractitioner(practitioner_id);
```

On récupère auprès de la base les indisponibilités du praticien après avoir formaté les dates pour la recherche :

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy");  
LocalDate d1 = LocalDate.parse(from, formatter);  
LocalDate d2 = LocalDate.parse(to, formatter);  
List<Unavailability> unavailabilities =  
    unavailabilityService.findByPractitionerIdAndRangeDate(  
        practitioner_id,  
        d1.toString(),  
        d2.toString());
```

Pour gérer cette requête SQL nous avons utilisé une native query. Pour cela on a ajouté une nouvelle méthode au repository dans UnavailabilityDao et déclaré la native query dans la classe Unavailability

```
@Query( nativeQuery = true,  
        name = "searchUnavailabilitiesByPractitionerAndRangeDate")  
List<Unavailability> findUnavailabilitiesByPractitionerIdAndRangeDate(  
    @Param("practitionerId") Long practitioner_id,  
    @Param("startTime") String startTime,  
    @Param("endTime") String endTime  
>;
```

```
@NamedNativeQuery(  
    name = "searchUnavailabilitiesByPractitionerAndRangeDate",  
    resultClass = com.votrerdvsante.api.unavailability.Unavailability.class,  
    query = " SELECT "  
    + " * "  
    + " FROM "  
    + " unavailability u "  
    + " WHERE "  
    + " u.practitioner_id = :practitionerId "  
    + " AND "  
    + " u.start_time >= :startTime "  
    + " AND "  
    + " u.end_time <= :endTime "  
    + " ORDER BY u.start_time ASC"  
)
```

À ce stade nous pouvons chercher le premier rendez-vous libre possible : si entre les deux dates fournies il n'y a pas de rendez-vous trouvé (possible) c'est à ce moment que nous utilisons le sens de la recherche ( défini côté Front ) pour déterminer un nouveau laps de temps à inspecter.

```

Boolean firstFreeSlotFound = false;
LocalDate goodDay = null;

while(!firstFreeSlotFound) {
    LocalDate dtmp1 = d1;
    LocalDate dtmp2 = d2;
    while ((dtmp1.isBefore(dtmp2) || dtmp1.isEqual(dtmp2)) && !firstFreeSlotFound) {
        Appointment firstFreeSlot = findFirstFreeSlotByPractitionersForAppointmentSubject(
            dtmp1, rules, weeklyHours, appointments);
        if (firstFreeSlot.getStartTime() != null) {
            firstFreeSlotFound = true;
            goodDay = LocalDate.of(
                firstFreeSlot.getStartTime().getYear(),
                firstFreeSlot.getStartTime().getMonth(),
                firstFreeSlot.getStartTime().getDayOfMonth());
        }
        dtmp1 = dtmp1.plusDays(1);
    }
    if(direction.equals(FORWARD)) {
        d1 = d1.plusDays(7);
        d2 = d2.plusDays(7);
    } else {
        d1 = d1.minusDays(7);
        d2 = d2.minusDays(7);
    }
}

```

Dès qu'un résultat est trouvé, on sort de la boucle et l'on peut chercher tous les rendez-vous libres possibles en fonction de la date du résultat précédent et retourner la réponse.

Côté Front, voici les boutons qui nous permettent de se déplacer dans l'agenda :

The image displays two side-by-side screenshots of a mobile application interface for scheduling appointments. Both screens have a light green header bar with the title "Praticiens".

**Left Screen (Landscape View):**

- Header:** "Praticiens" and "Dr. Docteur Knock1, 10 rue de Grenoble TOULOUSE".
- Section:** "1er rendez-vous" with a dropdown arrow icon.
- Calendar:** A grid showing days from Monday, February 25 to Sunday, March 3. Each day has two time slots: 08:00 and 08:35.
- Navigation:** Buttons for navigating between weeks, with a central calendar icon.

**Right Screen (Portrait View):**

- Header:** "Praticiens" and "Dr. Docteur Knock1, 10 rue de Grenoble TOULOUSE".
- Section:** "1er rendez-vous" with a dropdown arrow icon.
- Calendar:** A grid showing days from Monday, February 25 to Sunday, March 3. Each day has three time slots: 08:00, 08:35, and 09:10.
- Navigation:** Buttons for navigating between weeks, with a central calendar icon.

On voit que les boutons « avant / après », en fonction de la vue ne peuvent pas avoir exactement le même comportement

- Vue mobile : les boutons font se déplacer de jour en jour
- Vue tablette et PC : les boutons font se déplacer de semaine en semaine

Pour pouvoir déterminer le comportement à avoir on regarde la taille de l'écran à l'initialisation du composant :

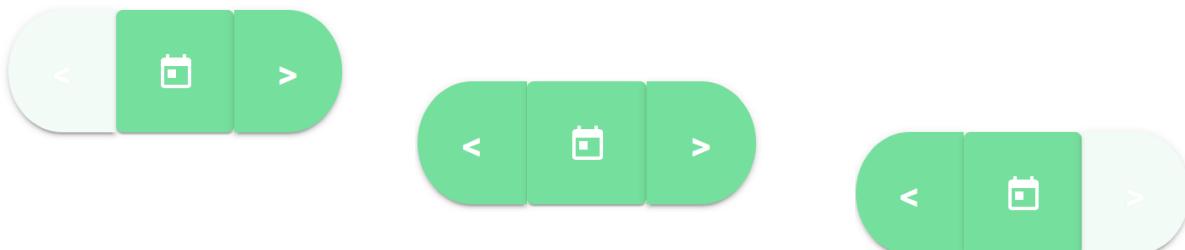
```
this.range = window.innerWidth > MOBILEMAXSIZEDATE ? RANGEDAYSDISPLAY : DAY;
```

La variable range prend alors le nombre de jour à afficher (7 ou 1).

Lorsqu'un utilisateur appuie sur un des boutons, on détermine quel jour est sélectionné et s'il est nécessaire de faire une nouvelle requête vers le serveur :

```
onChangeDayOrWeek(event): void {
    if (event === MINUS) {
        this.selectedDay =
            this.addOrSubtractRangeDate(this.selectedDay, -this.range);
        if (this.selectedDay <= this.minDate) {
            this.btnMinus = false;
        }
    } else if (event === PLUS) {
        this.selectedDay =
            this.addOrSubtractRangeDate(this.selectedDay, this.range);
    } else {
        this.selectedDay = new Date(event);
    }
    this.direction = this.getDirection();
    if (this.direction !== IN) {
        this.setRequest(this.selectedDay, this.direction);
        this.searchFreeAppointments();
    }
    this.checkBtnSelectorWeekActive();
}
```

Ce bout de code nous permet également de fixer l'apparence des boutons afin de signaler à l'utilisateur qu'il ne lui est pas possible de visionner l'agenda au delà de certaines dates.



On fait cela grâce à l'ajout de classes dans la balise HTML et la feuille de style liée au composant en question.

```
<button (click)="getMinus($event)"  
[ngClass]="'btn-week-selector-active': btnMinus,  
'btn-week-selector-inactive': !btnMinus'"  
mat-raised-button  
[disableRipple]!"btnMinus"></button>
```

```
.btn-week-selector-active {  
    background: rgba(117, 223, 157, 1);  
}  
  
.btn-week-selector-inactive {  
    background: rgba(117, 223, 157, 0.1);  
}
```

Si l'utilisateur clique sur un horaire proposé on ouvre une boîte de dialogue dans laquelle on passe l'instance du slot en question :

```
openDialog(): void {  
    const dialogRef = this.dialog.open(DialogOverviewDialogComponent, { config: {  
        panelClass: 'myapp-no-padding-dialog',  
        width: '250px',  
        data: {appointment: this.slot}  
    }});  
    dialogRef.afterClosed().subscribe( next: result => {  
        if (result) {  
            this.slotEvent.emit(this.slot);  
        }  
    });  
}
```

Si l'utilisateur décide de réserver un rendez-vous alors on déclenche une méthode du service qui fera une requête POST vers l'API pour stocker ce rendez-vous :

```
confirmSelectedSlot(event): void {  
    this.selectedAppointment = event;  
    this.patientCenterService.selectedAppointment = event;  
    this.patientCenterService.bookAppointment().subscribe( observer: {  
        next: appointment => {  
            this.freeslots = this.freeslots.filter( callbackfn: slot => slot !== event);  
            this.freeslots.sort( compareFn: function(a, b) {  
                return a.from.localeCompare(b.from);  
            });  
            this.setDayDate();  
            this.addAppointmentsToDays();  
        }  
    }  
}  
  
try {  
    ret = findFreeslotsByPractitionerIdAndSpecialtyIdAndAppointmentssubjectIdAndRangeDate(  
        practitioner_id, specialty_id, appointmentssubject_id, fromDate, toDate);  
} catch (NoAppointmentPreferenceFoundException e) {  
    return e.getResponseEntity();  
} catch (NoWeeklyHoursFoundException e) {  
    return e.getResponseEntity();  
}
```

Côté backend, après la vérification de la validité des données passées dans le body de la requête on vérifie si le rendez-vous n'est pas déjà pris

Lorsque ce rendez-vous est disponible on le stocke et on le récupère de suite pour le retourner à l'utilisateur :

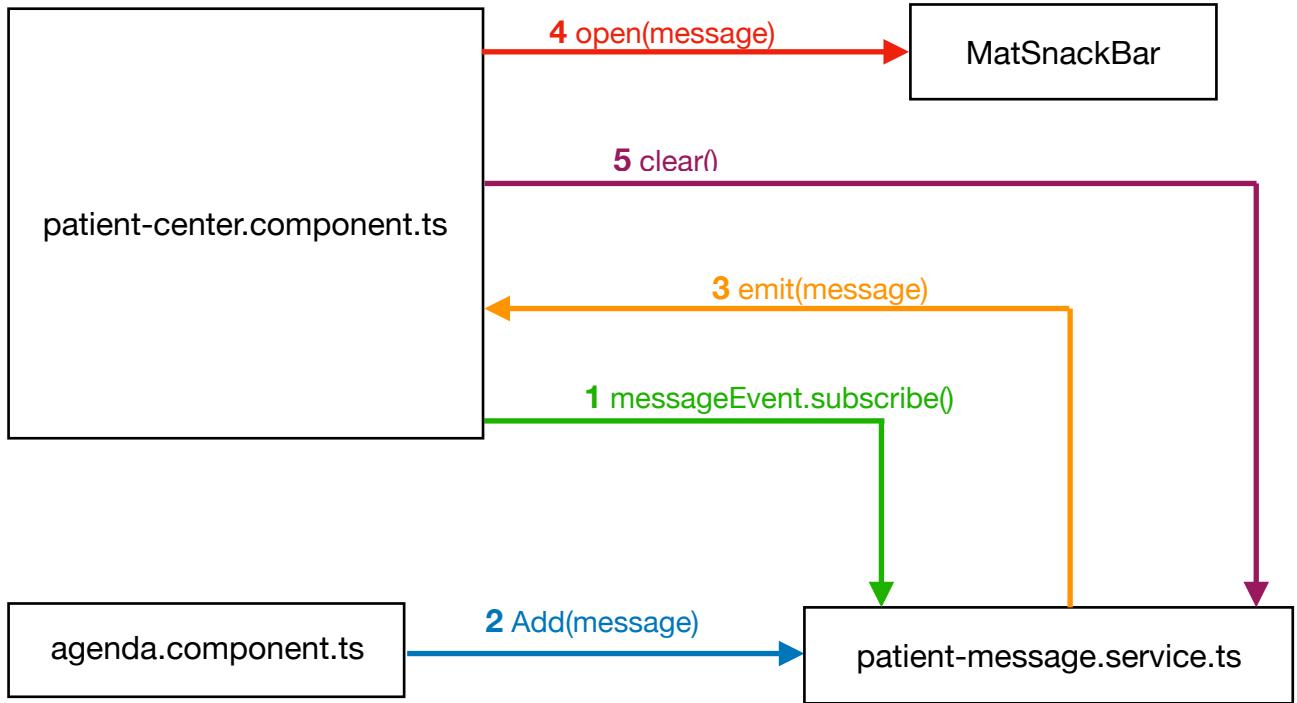
```
// add the booked appointment
Appointment appointmentAdded = appointmentService.save(appointment);
try {
    appointmentAdded = appointmentService.findById(appointmentAdded.getId());
} catch (NoAppointmentFoundException e) {
    // TODO how to deal with this error ?
    e.printStackTrace();
}
```

On utilise la méthode save du Repository dont l'équivalent SQL est :

```
INSERT INTO appointment
(end_time, start_time, practitioner_id, appointmentsubject_id, patient_id, specialty_id)
VALUES ('2019-03-05T09:40:00', '2019-03-05T09:10:00', '1', '1', '1', '1');
```

Retourner à l'utilisateur le nouveau rendez-vous est important afin de l'ajouter à la liste des rendez-vous qu'il a déjà pris, mais également pour informer l'utilisateur du résultat de la requête.

Nous avons mis en place un service spécifique pour gérer les messages destinés à l'utilisateur. Ce service peut-être utilisé par tous les components implémentés dans l'espace personnel. Voici un schéma illustrant ce mécanisme de messagerie lorsqu'un rdv a été stocké :



1 - messageEvent.subscribe()

```
this.patientMessageService.messageEvent.subscribe(
  generatorOrNext: () => {...}
);
```

2 - add(message)

```
this.patientMessageService.add('rendez-vous pris');
```

3 - emit(message)

```
add(message: any) {
  this.messages.push(message);
  this.messageEvent.emit(this.messages);
}
```

4 et 5 - open(message) et clear()

```
this.patientMessageService.messageEvent.subscribe(
  generatorOrNext: () => {
    this.openSnackBar(this.patientMessageService.messages.shift(), { action: 'fermer' });
    this.patientMessageService.clear();
  }
);
```

## **G. Conclusion**

Avant de faire ce stage je n'avais jamais travaillé sur un projet aussi important sur deux points :

- un projet commercial dans le domaine de la santé, un secteur qui nécessite une attention cruciale sur les problématiques de sécurité et de confidentialité des données
- le développement d'une application plus complexe que ce que j'avais vu auparavant et simultanément l'apprentissage de deux nouvelles technologies, Angular et Spring.

Ce furent des pentes raides à gravir par moment et il a fallu investir beaucoup de temps et d'énergie pour arriver au résultat escompté. Mais je suis assez fier du résultat.

Je n'ai pas vraiment eu l'occasion de travailler en équipe au cours du stage ce qui, d'une certaine façon, m'a manqué. Cependant cette expérience m'a amené à analyser petit à petit mes erreurs et j'en retire un grand profit.

Au final, grâce à tout ça, j'ai eu plaisir à travailler sur un projet riche, de la conception au développement pour le livrer dans les délais.

J'espère dans l'avenir continuer dans cette voie.

## H. Annexes

exemple de maquettes « patient » liées aux rendez-vous :

The left screenshot shows a sidebar with navigation options: Accueil, Rendez-vous, Documents, Questionnaire, and Messagerie. The main area displays "Mes rendez-vous" with three entries: DR.JEAN (04/12/2018, 10H00), DR.PAUL (23/02/2019, 23H00), and DR.CAMILLE (02/03/2019, 09H00). Below this is a "Rechercher un praticien" section with a map of a city and a list of practitioners. The right screenshot shows a similar layout but highlights a specific appointment for DR.PIERRE on 04/12/2018 at 10H00, with a detailed view of his availability from 08h30 to 18h30.

Versions mobile:

The four screenshots show the mobile application's interface across different screens:

- Mes rendez-vous:** Shows a list of upcoming appointments (DR.JEAN, DR.PAUL, DR.CAMILLE) and a "Rechercher un praticien" section with search fields for doctor and location.
- Prochain rendez-vous:** Displays the next appointment (DR.JEAN, 15/12/2018) with icons for phone, message, and location.
- Notifications:** Lists notifications from DR.JEAN, DR.PAUL, and DR.CAMILLE regarding messages and bills.
- Praticiens:** Shows a map of the city with practitioner locations and a detailed list of practitioners (DR.PIERRE) with their availability from 08h30 to 18h30, and a "Confirmer le rendez-vous" button.

exemple de maquettes « praticien » liées aux rendez-vous :

The first screen shows a profile picture of Paul Dupont and a sidebar with navigation icons: Dossier patient, Mes rendez-vous, Messages, Documents types, Prescriptions, Bibliothèque produit, and Bilan financier.

The second screen displays the "Rendez-vous de la journée" (Daily Appointments) with a list of slots from 08h30 to 18h30. Appointments are shown for Nathan M. at 08h30 and Véronique S. at 09h30.

The third screen shows "Notifications" with two messages: one from Nathan M. on 13/04/2017 and one from Véronique S. on 13/04/2017. It also includes a section for "HIER".

The interface includes a profile picture of Paul Dupont and a sidebar with the same navigation icons as the top screen.

The main area is titled "Mes rendez-vous" and shows a weekly grid from Monday to Saturday. Appointments are listed for Nathan M. at 08h30 on Monday and Véronique S. at 09h30 on Tuesday.

Two buttons at the top right allow defining availability "sur l'année" (year-round) or "sur la semaine" (week).

At the bottom, it says "Semaine du 11/04/2018" and "Ajouter un nouveau patient" with a plus icon.