Fic'Tif – Centre médical

Application web de prise de rendez-vous médicaux

par Nicolas BERNARD

Projet Réalisé dans le cadre de la formation Développeur Web et Web Mobile
réalisée à l'organisme de formation SOFIP Rouvignies

Table des matières

Présentation personnelle

Mon parcours

Ayant toujours baigné dans le monde du numérique et de l'informatique, j'ai toujours eu une appétence pour le développement. Lors de mes années d'études, j'ai d'ailleurs entrepris un cursus informatique à l'université du Mont Houy à Valenciennes (DEUST IOSI). J'ai alors découvert de nombreux domaines liés à l'informatique comme le réseau, la programmation et aussi le développement web. Au cours de ce cursus, c'est d'ailleurs ce dernier point qui m'a le plus inspiré quand il été question de réalisé un projet de fin d'étude au technologies et thèmes ouverts.

Mon entrée en formation

Après une longue période dans le milieu du help desk, j'ai décidé, à l'âge de 34 ans, d'effectuer une reconversion professionnelle. Le milieu dans lequel le réalisé était clair, le développement web ! C'est alors que j'ai appris l'existence de la formation DWWM dispensée par la SOFIP, je m'y suis directement inscris et j'ai eu la chance d'y participer. Depuis le 29 Mai, je la suis avec la plus grande assiduité et tout le sérieux dont je puisse faire preuve.

Le projet Fic'Tif- Concept

Introduction

Afin de mettre en avant les compétences que j'ai acquises depuis le début ma formation, j'ai voulu les illustrées dans un projet complet, ambitieux et passionnant. Je devais alors trouver un concept qui puisse cocher toutes les cases tout en symbolisant un challenge certain et être un prétexte à l'apprentissage.

Pour l'aspect complet, il me fallait un sujet qui introduise la gestion de données et leurs manipulations, un affichage dynamique qui soit simple à comprendre, en somme : une interface destinée à n'importe quel type de publique. Pour l'aspect ambitieux je souhaitais implémenter de nombreuses fonctionnalités et essayer de créer une structure conséquente. Et enfin, pour le passionnant, un sujet dans lequel je puisse m'y retrouver.

A l'intersection de ses trois prérequis que je me suis fixé, j'ai trouvé le domaine médical. Je me devais alors de réaliser un site pour tout le monde tout en prenant en compte les règles d'accessibilité. Le sujet est suffisamment vaste pour que je puisse y trouver de nombreux besoins auxquels il faudrait répondre. Et bien que le sujet ne fasse pas partie de mes passions ou que mon quotidien n'est pas réellement attaché au milieu médical, j'aimais l'idée de créer quelque chose que tout le monde pourrait utiliser un jour ou l'autre. Et puis ça me permet de garder à l'esprit que l'application que je développe ne doit pas que me convenir, mais dois aussi être pensée pour convenir au plus grand nombre.

C'est alors que l'idée d'un site de prise de rendez-vous médical m'est venue. Mon premier projet personnel de développement web et web mobile (Hors projet étudiant) sera alors un site de prise de rendez-vous médical !

Services délivrés par l'application

- Prise de rendez-vous

La principale fonctionnalité de mon application serait alors de prendre rendez-vous avec un médecin particulier en fonction de ses disponibilités. Etant la fonctionnalité principale, je voulais qu'elle soit facilement accessible et simple d'usage.

- Suivi des rendez-vous

Une fois un ou plusieurs rendez-vous effectué/s. Je voulais qu'ils soient visibles dans une page dédiée afin que l'utilisateur puisse voire l'état de son/ses rendez-vous et également avec qui, ou et quand ce/ces rendez-vous aura/auront lieu.

- Gestion du compte utilisateur

Je me suis rendu compte très rapidement que j'aurais besoin d'un système d'identifiant. J'aurais besoin d'implémenter un système de connexion utilisateur avec la possibilité de se déconnecter. Mais je voulais aussi que l'utilisateur puisse avoir le contrôle sur ses informations de compte, qu'il puisse en changer ou encore supprimer son compte.

D'autres fonctionnalités ont germé dans mon esprit au cours du développement de l'application mais ces trois-là représentent la base de ma réflexion initiale.

Besoins fonctionnels

Ces différents services génèrent déjà plusieurs besoins avant d'être pleinement conçu. J'ai appris, après le début de mon projet, que ces réflexions conceptuelles sont à l'origine de la rédaction d'une série de besoins métier qu'un Product Owner se doit de centraliser et verbaliser afin de les distribuer dans des User Story.

En ce sens, j'ai divisé mes besoins fonctionnels entre le profil patient et le profil docteur.

- Quand je suis patient
  - ❖ Je dois être capable de me connecter POUR accéder aux services requérant une connexion
  - ❖ Je dois être capable de m'inscrire POUR obtenir des informations de connexion
  - ❖ Je dois être capable de me déconnecter POUR des raisons de sécurité ET POUR permettre à d'autres utilisateurs d'utiliser l'application
  - ❖ Je dois être capable de reprendre ma session en cours POUR ne pas avoir à me reconnecter si la période de session n'est pas dépassée
  - ❖ Je dois être capable de consulter mes informations de compte POUR vérifier qu'elles sont à jour
  - ❖ Je dois être capable de modifier mes informations de compte POUR les mettre à jour
  - ❖ Je dois être capable de supprimer mon compte POUR retirer mes informations du site
  - ❖ Je dois être capable de voire la liste des médecins POUR connaitre la totalité des services proposés
  - ❖ Je dois être capable, si je suis connecté, d'atteindre la page de rendez-vous POUR prendre rendez-vous avec un médecin
  - ❖ Je dois être capable, sur la page de rendez-vous, de filtrer les médecins par Spécialité POUR obtenir un rendez-vous qui correspond à mes besoins
  - ❖ Je dois être capable, sur la page de rendez-vous, de sélectionner un créneau horaire POUR réserver ce créneau pour mon rendez-vous

- ❖ Je dois être capable, sur la page de rendez-vous, d'être confronter à une étape de confirmation POUR ne pas prendre rendez-vous par accident
- ❖ Je dois être capable de consulter les rendez-vous en cours qui me concerne POUR ne pas les oublier

- Quand je suis docteur
  - ❖ Je dois être capable de me connecter POUR accéder aux services requérant une connexion
  - ❖ Je dois être capable de me déconnecter POUR des raisons de sécurité ET POUR permettre à d'autres utilisateurs d'utiliser l'application
  - ❖ Je dois être capable de reprendre ma session en cours POUR ne pas avoir à me reconnecter si la période de session n'est pas dépassée
  - ❖ Je dois être capable de consulter mes informations de compte POUR vérifier qu'elles sont à jour
  - ❖ Je dois être capable de modifier mes informations de compte POUR les mettre à jour
  - ❖ Je dois être capable de supprimer mon compte POUR retirer mes informations du site
  - ❖ Je dois être capable de consulter les rendez-vous en cours qui me concerne POUR ne pas les oublier

Problématiques techniques

Afin de répondre efficacement à ses besoins, j'ai des impératifs techniques :

- ❖ J'ai besoin d'un front end pour le visuel de mon application tout en veillant à l'accessibilité et le backend pour gérer la communication avec les données stockées.
- ❖ J'ai besoin d'une base de données pour stocker les données.
- ❖ Ma page de prise de rendez-vous doit accueillir un calendrier à l'affichage dynamique pour organiser de façon claire et visuelle les créneaux disponibles pour que le patient puisse prendre son rendez-vous.

Ces points ont constitué le constat initial de mon application.

Solutions envisagées

Afin de surmonter ces problématiques, j'ai opté pour l'usage de Node.js et React pour le front end, cela me permettra une flexibilité pour m'adapter aux problématiques que je rencontrerais au cours du développement et me permettra plus facilement de mettre en place une structure conséquente. J'ai décidé d'utiliser Express en complément de Node.js pour effectuer le back-end. Ce choix s'est effectué de par la simplicité d'usage, la quantité d'informations présente sur internet concernant Express et surtout par sa capacité à fournir un backend robuste.

Fic'Tif- Conception

Maquettage de l'application

- La théorie des couleurs

Lors de la conception de ma maquette, j'ai été préoccupé par l'identité de mon application. La couleur y jouant un rôle capital, je me suis intéressé aux différentes nuances disponibles et également à leur signification.

D'après le site Adobe.com, *(https://www.adobe.com/fr/creativecloud/design/discover/color-symbolism.html)*

« Le Bleu

***Associé à** : la mer et le ciel*

***Effet psychologique** : stabilité, harmonie, paix, calme et confiance*

***Utilisation par les marques et dans le design** : le bleu peut aider à positionner la marque comme digne de confiance, fiable et relaxante. C'est le cas d'enseignes de grande distribution où l'on peut faire ses courses en un seul endroit pratique. Le secteur de la santé utilise généralement le bleu dans leur image de marque pour aider les gens à associer la marque à un produit de qualité, fiable et sûr.*

Le Orange

***Associé à** : la créativité, l'aventure, l'enthousiasme, le succès et l'équilibre.*

***Effet psychologique** : ajoute un peu de plaisir.*

***Utilisation par les marques et dans le design** : dans les chaînes de télévision pour enfants pour représenter la créativité, l'enthousiasme et le côté ludique. De nombreuses enseignes de bricolage utilisent la couleur orange pour représenter la créativité et la rénovation de la maison. L'orange est une couleur forte qui convient le mieux aux marques tout aussi bruyantes et énergiques. Si votre marque est quelque chose d'un peu plus traditionnel, l'orange n'est probablement pas la voie à suivre. »*

Pour le domaine médical, le bleu est très régulièrement utilisé. Afin que l'utilisateur ne soit pas perturbé par une charte de couleur différente de celle qu'il a l'habitude de voir dans ce domaine, j'ai opté pour le bleu. Aussi, je voulais obtenir un visuel attrayant et dynamique, de ce fait j'ai sélectionné la couleur orange, d'autant plus qu'elle s'associe bien au bleu comme le montre ce second article

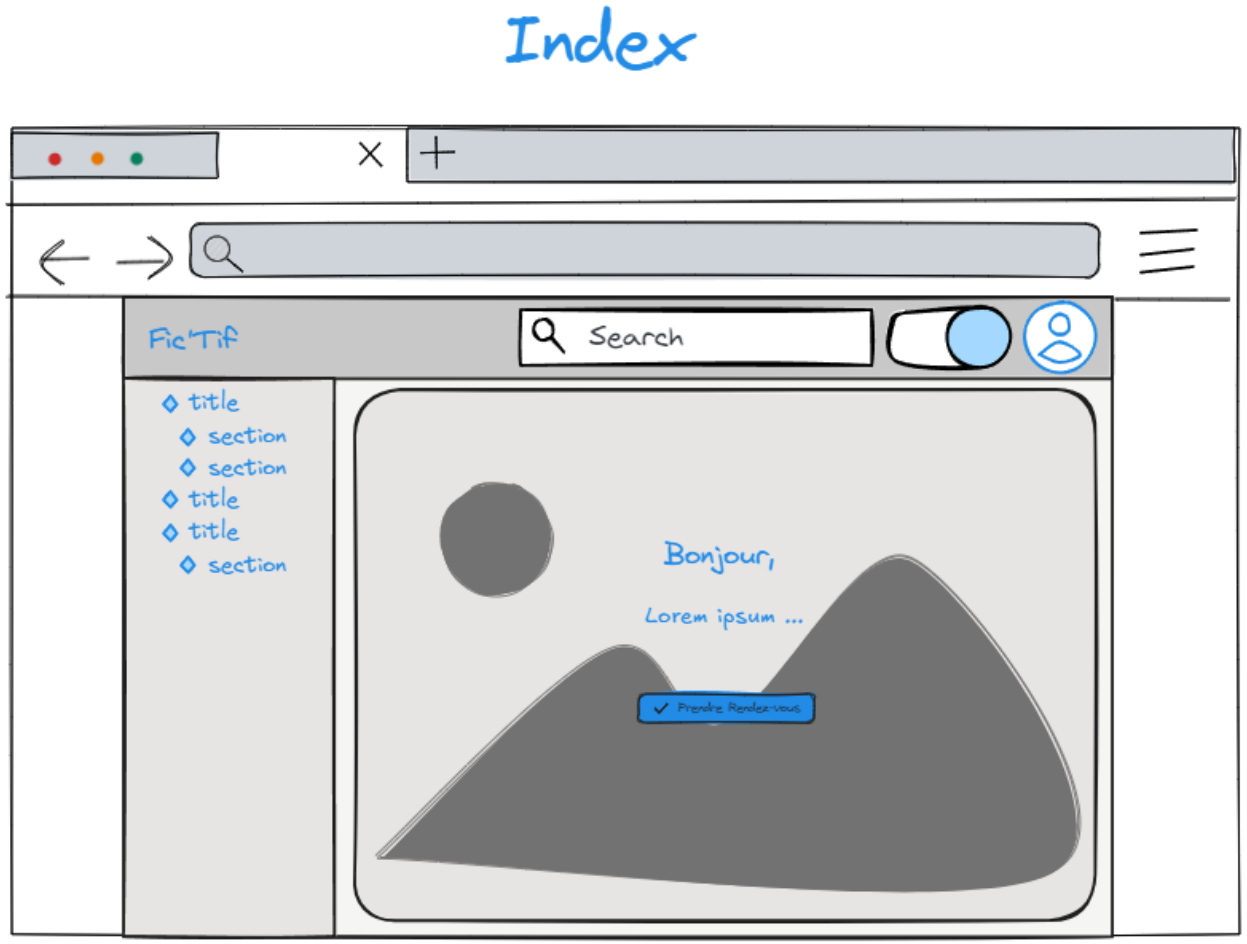
D'après le site Adobe.com, *(https://www.adobe.com/fr/creativecloud/design/discover/complementary-colors.html)*

« Couleurs complémentaires avec une couleur primaire

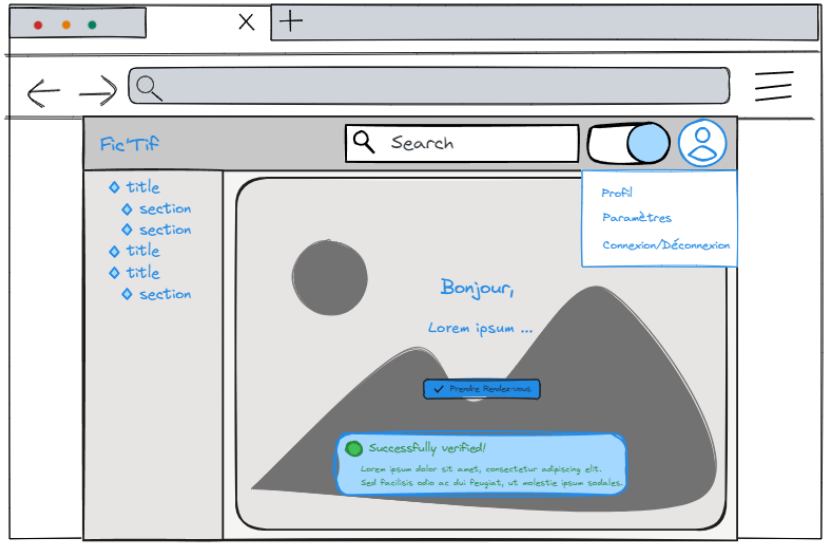
***Le bleu et l'orange**. En associant ces deux couleurs dans vos réalisations graphiques, vos peintures, vos photos ou encore votre logo, vous vous assurez d'avoir un rendu contrasté et intemporel.*

- Wireframes

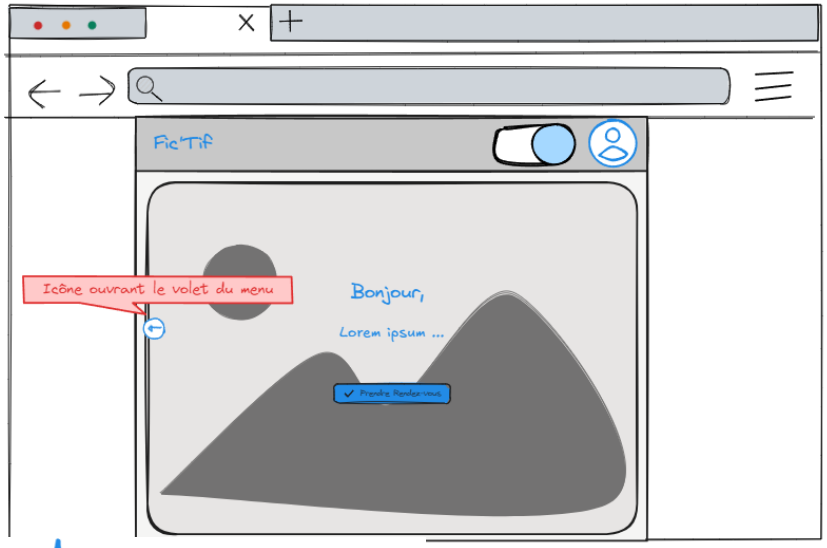
Pour le design de mon application, je me suis contraint à éviter au maximum le scrolling vertical ou horizontal. J'ai donc pensé la vue mobile et desktop en fonction au sein d'un wireframe que j'ai réalisé avec Excalidraw (*https://excalidraw.com*).



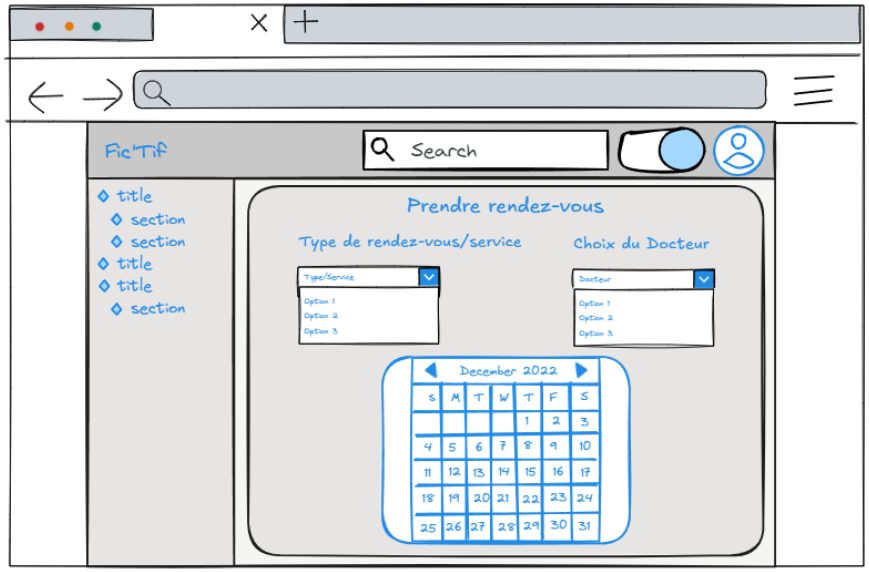
Index avec fonctionnalités



Vue mobile du layout

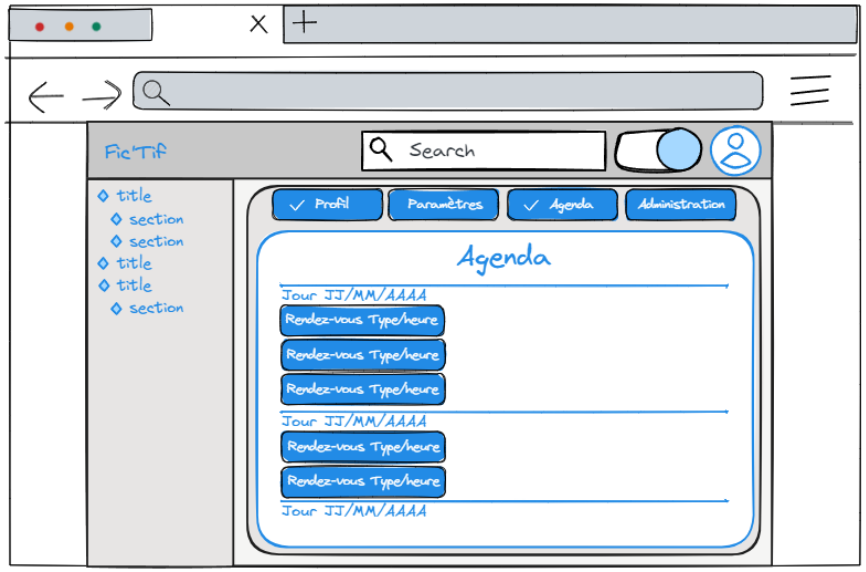


Prise de rendez-vous



Profil utilisateur

Agenda



Mise en place de l'environnement

- Les repos git

Dès le début du projet, j'ai imaginé le frontend séparé du backend. Cependant pour une raison de praticité j'ai travaillé avec un seul repo git qui contenait deux dossiers accueillant les deux parties de mon projet.

Dans un premier temps, j'ai effectué des opérations simples directement sur la branche main. Mais comme le projet demandé un développement plus ambitieux que les projets effectués dans le cadre de l'apprentissage au sein de la formation, il me fallait adopter une autre méthode de travail. J'ai alors commencé à utiliser des branches différentes pour l'implémentation de features, assurant une version stable sur la branche main et me servant des autres branches pour travailler. De cette façon j'ai appris l'usage des rebases et la gestion des conflits.

J'ai également adopté une convention de nommage pour mieux structurer le versionnage sur le repo distant. (Exemples de commit : ' git commit -m « fix: feature login »' 'git commit -m « feature: login »)

Le projet m'a permis de mettre en relief l'importance de maitriser git et la rigueur qui est nécessaire pour obtenir un dossier de projet propre.

- Installation de l'IDE

Mon choix d'IDE (Integrated Development Environment) s'est porté sur Visual studio code (*https://code.visualstudio.com/*) pour son large choix d'extensions, sa facilité d'installation et la quantité de documentation accessible sur internet. Je l'ai paramétré pour qu'il convienne à mes préférences. L'usage de linter grâce à ESlint, m'a permis tout au long de mon développement de visualiser plus facilement les erreurs présentes dans le code.

Mise en place de la base de données

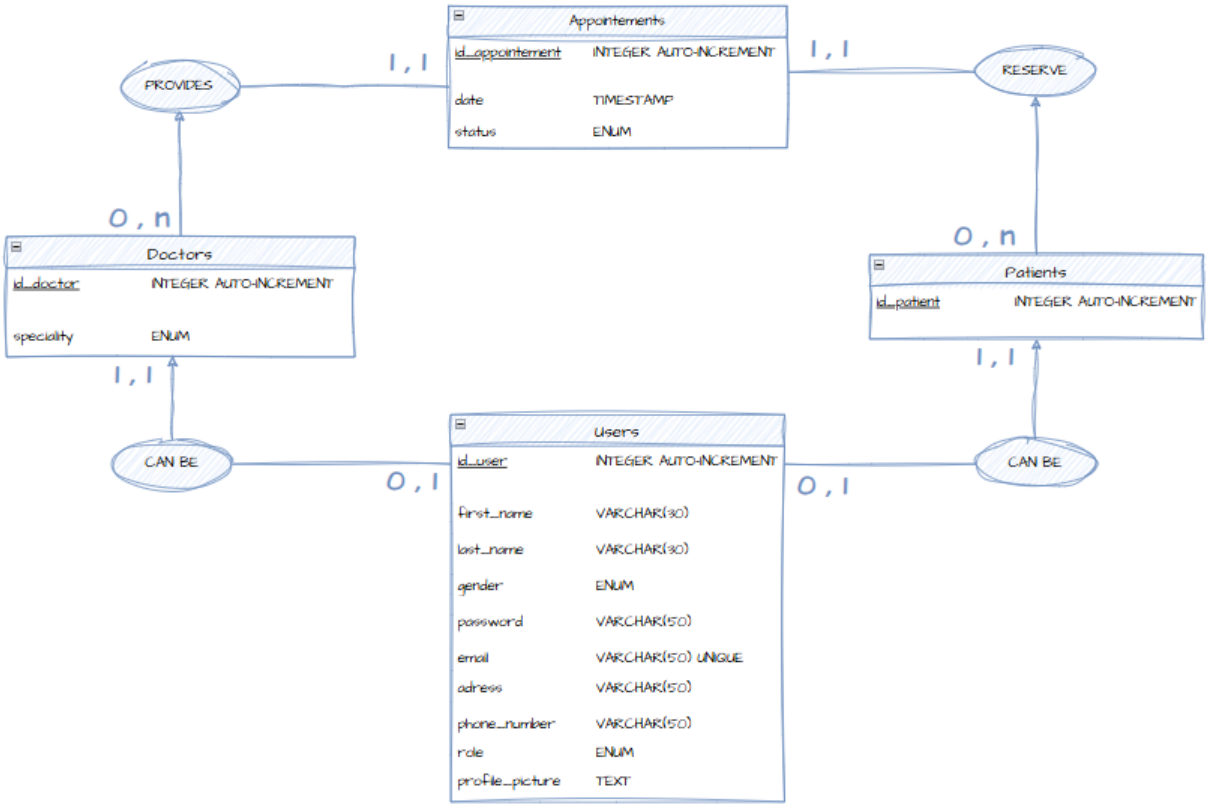
La création d'une base de données est une étape importante. Cette dernière influencera directement la façon dont l'application s'organisera pour mener à bien les actions attendues, de ce fait j'ai créer un MCD pour mieux organiser mes idées et concevoir la logique autour de mes données et leurs interactions, puis j'ai établi seulement après la base de données.

- MCD

J'ai opté pour une organisation en losange de mes données entre les utilisateurs (Users) et les rendez-vous (Appointments) qui passerait alors par les docteurs (Doctors) et par les patients (Patients). En effet, j'ai considéré qu'un rendez-vous a lieu entre un patient et un docteur et que chacun d'entre eux partage le fait d'être un utilisateur du site, ayant des données communes comme un nom, prénom, genre, email, mot de passe, etc…

Avec cette organisation, j'évite d'avoir deux collections de données qui sont jumelles à la différence des docteurs qui aurait une spécialité (speciality) en plus par rapport aux patients.

De son côté, la collection rendez-vous n'ajoute qu'un horaire de début de consultation et un status. J'ai en effet opté pour des consultations à durée unique de 30 minutes, de ce fait je n'ai besoin que de l'horaire de début. Le status permet de notifier si le rendez-vous est en attente, annulé, ou effectué (« pending », « canceled » ou « done »).



Pour répondre à cette problématique, j'ai alors imaginé un second portail web qui accueillerait le formulaire d'inscription des docteurs avec des vérifications supplémentaires (Code reçu par mail par exemple) et la partie administration de Fic'Tif. Cependant, faute de temps, ce second portail n'est pas présent dans ce projet pour le moment.

Les dépendances indispensables

Pour réaliser correctement le projet, je connaissais déjà les problématiques techniques grâce au travail sur le concept du projet. De ce fait, souhaitant utiliser React et Node.js pour le frontend et Express et Node.js pour le backend, j'avais besoin d'une liste de module qui se montreraient indispensable pour mon projet, couvrant des aspects de sécurité ou répondant aux problématiques techniques propres au projet.

- *Front-end*

❖ React Router - Permet de gérer la navigation entre les différentes vues/pages de mon application. (*https://www.npmjs.com/package/react-router-dom*)

❖ React big calendar - Permet d'ajouter un calendrier interactif et personnalisable dans mon application. (*https://www.npmjs.com/package/@types/react-big-calendar*)

❖ React Toastify - Permet d'afficher des notifications toast (notifications temporaires) de manière simple et élégante tout en gardant le comportement de la page souhaité (Pas de scrolling) (*https://www.npmjs.com/package/react-toastify*)

Ici, les modules convoités par défaut se devaient de répondre à des problématiques techniques. J'ai alors commencé à parcourir leurs différentes documentations pour veillez à ce que leur emploi soit justifié et corresponde aux besoins de mon projet.

- *Back-end*

❖ Bcryptjs - Permet de hacher les mots de passe de manière sécurisée. (*https://www.npmjs.com/package/bcryptjs*)

❖ cookie parser – Permet de parser les cookies envoyés dans les requêtes HTTP. (*https://www.npmjs.com/package/cookie-parser*)

❖ Cors - Permet de gérer le **Cross-Origin Resource Sharing (CORS)** pour autoriser ou restreindre l'accès à l'API depuis différents domaines. (*https://www.npmjs.com/package/cors)*

❖ Express async errors - Permet de gérer les erreurs asynchrones dans Express. (*https://www.npmjs.com/package/express-async-errors*)

❖ Express mongo sanitize - Protection contre les attaques d'injection MongoDB en nettoyant les requêtes entrantes. (*https://www.npmjs.com/package/express-mongo-sanitize*)

❖ Express rate limit - Limitation du nombre de requêtes d'un utilisateur pour prévenir les attaques par déni de service (DDoS) ou le brute-force. (*https://www.npmjs.com/package/express-rate-limit*)

❖ Helmet - Middleware de sécurité pour Express qui aide à protéger l'application contre des vulnérabilités web courantes. (*https://www.npmjs.com/package/helmet*)

❖ http status codes - Fournit des constantes pour les codes de statut HTTP afin de rendre le code plus lisible et éviter les erreurs de typographie. (*https://www.npmjs.com/package/http-status-codes*)

❖ Jsonwebtoken - Permet de créer et vérifier des JSON Web Tokens (JWT) pour l'authentification. (*https://www.npmjs.com/package/jsonwebtoken*)

❖ Mongoose - ODM (Object Data Modeling) pour MongoDB, facilite l'interaction avec la base de données en utilisant des modèles. (*https://www.npmjs.com/package/mongoose*)

❖ Zod - Bibliothèque de validation de schémas JavaScript pour valider les entrées. (*https://www.npmjs.com/package/zod*)

Avec l'usages de ses modules je m'assure d'obtenir un backend sécurisé, capable de gérer les sessions des utilisateurs et les erreurs éventuelles ainsi que la communication avec une base de donnée MongoDB.

Remarque : Ce projet m'a permis d'apprendre comment se comporte un backend et comment développer une API (Gestion des erreurs, communication entre le front et le back etc etc). J'ai donc suivi des recommandations pour m'assurer d'aborder chaque point de vigilance que l'on se doit de connaitre en tant que développeur.

Création du backend

- Technologies employées

J'ai utilisé Express et Node.js pour le backend, j'avais besoin d'une liste de module qui se montreraient indispensable pour mon projet, couvrant des aspects de sécurité ou répondant aux problématiques techniques propres au projet.

- Création de la base MongoDB

J'ai alors créé ma base MongoDB et je l'ai associé à mon backend naissant. J'ai pour cela eu recours a un fichier .env qui me permet de stocker les variables d'environnement nécessaires au bon fonctionnement de mon application. La première de ses variables fut donc celle requise pour me connecter à ma base de données MongoDB.

Par la suite, ce fichier se verra également attribuer des variables qui serviront à paramétrer mes Tokens, mes cookies et le port de déploiement lors de la phase de développement.

- Modules et dépendances

❖ Bcryptjs - Permet de hacher les mots de passe de manière sécurisée. (*https://www.npmjs.com/package/bcryptjs*)

❖ cookie parser – Permet de parser les cookies envoyés dans les requêtes HTTP. (*https://www.npmjs.com/package/cookie-parser*)

❖ Cors - Permet de gérer le **Cross-Origin Resource Sharing (CORS)** pour autoriser ou restreindre l'accès à l'API depuis différents domaines. (*https://www.npmjs.com/package/cors)*

❖ Express async errors - Permet de gérer les erreurs asynchrones dans Express. (*https://www.npmjs.com/package/express-async-errors*)

❖ Express mongo sanitize - Protection contre les attaques d'injection MongoDB en nettoyant les requêtes entrantes. (*https://www.npmjs.com/package/express-mongo-sanitize*)

❖ Express rate limit - Limitation du nombre de requêtes d'un utilisateur pour prévenir les attaques par déni de service (DDoS) ou le brute-force. (*https://www.npmjs.com/package/express-rate-limit*)

❖ Helmet - Middleware de sécurité pour Express qui aide à protéger l'application contre des vulnérabilités web courantes. (*https://www.npmjs.com/package/helmet*)

❖ http status codes - Fournit des constantes pour les codes de statut HTTP afin de rendre le code plus lisible et éviter les erreurs de typographie. (*https://www.npmjs.com/package/http-status-codes*)

❖ Jsonwebtoken - Permet de créer et vérifier des JSON Web Tokens (JWT) pour l'authentification. (*https://www.npmjs.com/package/jsonwebtoken*)

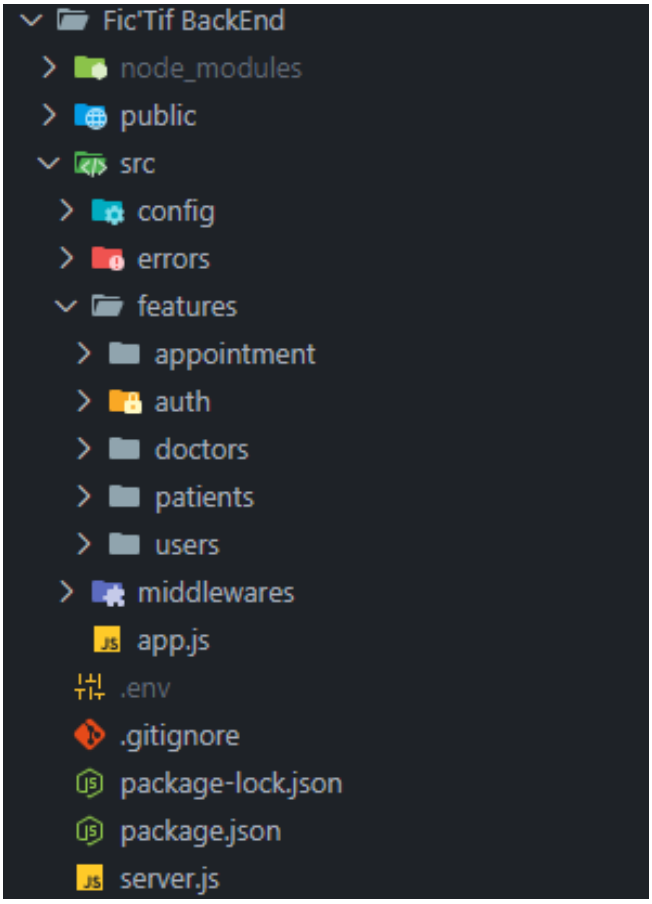
❖ Mongoose - ODM (Object Data Modeling) pour MongoDB, facilite l'interaction avec la base de données en utilisant des modèles. (*https://www.npmjs.com/package/mongoose*)

❖ Zod - Bibliothèque de validation de schémas JavaScript pour valider les entrées. (*https://www.npmjs.com/package/zod*)

Avec l'usages de ses modules je m'assure d'obtenir un backend sécurisé, capable de gérer les sessions des utilisateurs et les erreurs éventuelles ainsi que la communication avec une base de données MongoDB.

- Structure

La structure de mon backend veille à regrouper chaque fonctionnalité par types, permettant une navigation fluide et un suivi efficace de ces mêmes fonctionnalités.



Les noms de dossiers config et errors sont relativement pertinents. Ils me permettent de d'accueillir le fichier de configuration de la connexion vers la base de données et mes gabarits d'erreur comme pour l'erreur d'authentification.

Le dossier features contient les controllers, services et potentiel shémas/models nécessaire pour dispenser les services dont mon application a besoin.

auth par exemple, gère tout ce qui concerne la connexion des utilisateurs, de la création de compte à la vérification de validité de session.

Le dossier middlwares contient les différents middlewares comme ceux nécessaires à la validation de Schema à l'aide de zod, les middlewares de gestion d'erreur, ou encore d'erreurs d'authentification.

- Création des premières routes

J'ai alors mis en place mes premières routes pour tester le bon fonctionnement de mon backend.

```
import { auth } from "./features/auth/index.js";
app.use("/api/v1/auth", auth);

import { users } from "./features/users/index.js";
app.use("/api/v1/users", users);

import { appointment } from "./features/appointment/index.js";
app.use("/api/v1/appointments", appointment);

import { doctors } from "./features/doctors/index.js";
app.use("/api/v1/doctors", doctors);

app.use(notFound);

app.use(errorHandler);
```

Organisée de cette manière, l'application est donc structurée en différentes fonctionnalités (authentification, utilisateurs, rendez-vous, médecins) avec des points d'API distincts, ce qui facilite la gestion et la maintenance de chaque fonctionnalité.

Les middlewares notFound et errorHandler permettent de gérer les cas où les routes sont introuvables ou où des erreurs surviennent pendant le traitement de la requête.

Cette séparation assure une clarté sur les différentes fonctionnalités de l'application, qui sont ensuite structurées de la façon suivante :

ROUTES -> MIDDLEWARES -> CONTROLLERS -> SERVICES

- *Routes*

Chaque features a un fichier .route qui lui est associé, permettant de rediriger vers les différents controllers des services qui sont englobés dans la fonctionnalité.

Par exemple le système d'authentification :

```
import express from "express";
const router = express.Router();
import validate from "../../middlewares/validation.middleware.js";
import { LoginUserSchema, RegisterUserSchema } from "../users/users.schema.js";
import { RegisterDoctorSchema } from "../doctors/doctors.schema.js";
import * as authController from "./auth.controller.js";
import authenticateUser from "../../middlewares/auth.middleware.js";

//* Patients
router.post(
  "/register/patient",
  validate(RegisterUserSchema),
  authController.registerPatient
);

//* Doctors
router.post(
  "/register/doctor",
  validate(RegisterDoctorSchema),
  authController.registerDoctor
);

router.post("/login", validate(LoginUserSchema), authController.login);
router.post("/logout", authController.logout);

router.get("/isLogged", authenticateUser, authController.isLogged);

export default router;
```

De cette manière j'obtiens un suivi clair de ce que chaque route effectue et intégrer un middleware comme ceux de validation de Schema ou de validation de session courante est facile sans perturber la clarté des routes ainsi créées.

- *Controllers*

```
const login = async (req, res) => {
  const user = await usersService.get({ email: req.body.email });

  if (!user) {
    throw new UnauthenticatedError("Identifiants invalides.");
  }

  const id = user._id;
  const email = user.email;

  const isPasswordCorrect = await user.comparePasswords(req.body.password);
  if (!isPasswordCorrect) {
    throw new UnauthenticatedError("Identifiants invalides.");
  }

  const token = jwt.sign({ id, email }, process.env.JWT_SECRET, {
    expiresIn: process.env.JWT_LIFETIME,
  });

  const oneDay = 24 * 60 * 60 * 1000;

  res.cookie("accessToken", token, {
    HttpOnly: true,
    secure: process.env.NODE_ENV === "production",
    signed: true,
    expires: new Date(Date.now() + oneDay),
  });
```

Ce controller a pour rôle de vérifier que la requête reçue est valide, sinon elle retourne une erreur d'authentification si la requête ne possède pas un email existant dans la base de données (Cette vérification s'effectue à l'aide d'un service dédié : usersService.get) ou si le mot de passe récupéré ne correspond pas à celui associé à l'adresse mail (Cette vérification s'effectue à l'aide du model de user qui a une méthode intégrée).

- *Services*

```
import User from "./users.model.js";

const get = (email) => {
  return User.findOne(email);
};
```

Ce service est enfin appelé pour effectuer les vérifications nécessaires afin de logger un utilisateur.

Au préalable, l'on doit utiliser un ODM (Ici mongoose) pour définir des modèles d'objets afin de correspondre à la structure de nos documents et interagir avec eux à l'aide d'opérations CRUD (Create, Read, Update, Delete), dans cet exemple ce serait READ.

Il va rechercher dans la collection users un document avec comme valeur l'adresse email reçu dans la requête pour le champ email. (Les documents stockés dans la base MongoDB étant dans un format très similaire au JSON). Si un utilisateur possède cette adresse email, le service renvoi alors cet utilisateur au controller qui continue ses vérifications.

- *Models*

Pour réaliser cette action, on a donc besoin du model d'objet user.

```
const UserSchema = new Schema({
  firstName: {
    type: String,
    required: [true, "Veuillez fournir un prénom"],
    maxlength: 50,
    minlength: 3,
  },
  lastName: {
    type: String,
    required: [true, "Veuillez fournir un nom de famille"],
    maxlength: 50,
    minlength: 3,
  },
  birthDate: {
    type: Date,
    required: [true, "Veuillez fournir une date de naissance"],
    validate: {
      validator: function (value) {
        return value < new Date();
      },
      message: "Veuillez fournir une date de naissance valide",
    },
  },
  email: {
    type: String,
    required: [true, "Veuillez fournir un email"],
    unique: true,
    match: [
      /^(([^<>()[\]\\.,;:\s@"]+(\.[^<>()[\]\\.,;:\s@"]+)*)|(".+")
      "Veuillez fournir un email valide",
    ],
  },
  password: {
    type: String,
    required: [true, "Veuillez fournir un mot de passe"],
    minlength: 6,
  },
```

Ici, les propriétés du modèle sont définies afin qu'elles servent pour peupler les documents dans la base de données. Elles peuvent être définies avec des critères, par exemple, le nom et le prénom son ici requis et doivent être compris entre 3 et 50 caractères. D'autres propriétés sont présentes dans le modèle user, comme l'adresse, le numéro de téléphone et le rôle. Le rôle d'un user ne peut avoir que deux valeurs « patient » ou « doctor », c'est cette propriété qui permet de différencier un patient d'un médecin dans certains usages.

Ensuite, dans le cas du modèle users, on a également besoin de méthodes. Les méthodes sont des fonctions qui sont définies directement sur le modèle afin de manipuler des données ou d'effectuer des opérations spécifiques sur des documents de la base de données.

```
// Hachage du mot de passe avant la sauvegarde
UserSchema.pre("save", async function () {
  if (this.isModified("password")) {
    const salt = await bcrypt.genSalt();
    this.password = await bcrypt.hash(this.password, salt);
  }
});

// Hachage du mot de passe avant la mise à jour
UserSchema.pre("findOneAndUpdate", async function () {
  if (this._update.password) {
    const salt = await bcrypt.genSalt();
    this._update.password = await bcrypt.hash(this._update.password, salt);
  }
});

// Méthode pour exclure le mot de passe lors de la conversion en JSON
UserSchema.methods.toJSON = function () {
  let userObject = this.toObject();
  delete userObject.password;
  return userObject;
};

// Méthode pour comparer les mots de passe
UserSchema.methods.comparePasswords = async function (candidatePassword) {
  const isMatch = await bcrypt.compare(candidatePassword, this.password);
  return isMatch;
};
```

- *Schemas*

Les Schemas sont utiles pour vérifier les informations provenant du front et donc de l'utilisateur. C'est une bonne pratique d'effectuer cette action avant de contacter la base de données pour ne pas la compromettre en cas d'informations vérolées. Si nous reprenons l'exemple des routes précédemment exposées, la route /register/patient/ a un middleware de validation de Schema (Celui de « RegisterUserSchema ») qui compare les données reçues avec celles du Schema suivant :

```
const RegisterUserSchema = z.object({
  firstName: z
    .string()
    .trim()
    .min(3, { message: "Doit avoir au minimum 3 caractères" })
    .max(50, { message: "Doit avoir au maximum 50 caractères" }),
  lastName: z
    .string()
    .trim()
    .min(3, { message: "Doit avoir au minimum 3 caractères" })
    .max(50, { message: "Doit avoir au maximum 50 caractères" }),
  birthDate: z.string().refine(
    (dateString) => {
      const date = new Date(dateString);
      return date < new Date();
    },
    { message: "Date de naissance invalide" }
  ),
  email: z.string().email({ message: "Email invalide" }),
  password: z
    .string()
    .trim()
    .min(6, { message: "Doit avoir au minimum 6 caractères" }),
  address: z
    .string()
    .trim()
    .min(8, { message: "Doit avoir au minimum 10 caractères" })
    .max(120, { message: "Doit avoir au maximum 120 caractères" }),
  phoneNumber: z
    .string()
    .trim()
    .length(10, {
      message: "Le numéro de téléphone doit être composé de 10 chiffres",
    })
    .startsWith("0", { message: "Le numéro de téléphone doit commencer par 0" })
    .regex(/^\d{10}$/, {
      message: "Le numéro de téléphone doit contenir uniquement des chiffres.",
    }),
  gender: z.enum(["man", "woman"]),
});
```

Cela ressemble dans un certain sens au modèle, puisque le Schema vérifie que les informations reçues respectent bien des critères. Mais son objectif est bien différent ! Mongoose et ses modèles permettent d'effectuer des opérations en vue de communiquer avec la base de données. Zod et ses Schemas quant à eux, vérifient que les informations saisies correspondent au format attendu pour éviter de potentiels problèmes par la suite et pour renforcer la sécurité liée aux requêtes que l'API reçoit.

- *Middlewares*

Les middlewares sont donc des fonctions qui vont effectuer des vérifications ou des modifications sur la requête. La validation des schemas se comportent comme des middlewares et l'authenticateUser est également un middleware que j'appelle pour vérifier si le JWT token est toujours en cours de validité à chaque fois que là route /isLogged est appelée.

```
import { UnauthenticatedError } from "../errors/index.js";
import jwt from "jsonwebtoken";

const authenticateUser = (req, _res, next) => {
  const token = req.signedCookies.accessToken || null;

  if (!token) {
    throw new UnauthenticatedError("Pas de token fournit");
  }

  try {
    const decodedToken = jwt.verify(token, process.env.JWT_SECRET);
    const { id, email } = decodedToken;
    req.user = { id, email };

    next();
  } catch (error) {
    throw new UnauthenticatedError("Accès non autorisé");
  }
};

export default authenticateUser;
```

De cette façon j'obtiens une route globale qui gère mon authentification sur l'application /auth qui est ensuite divisée en plusieurs routes qui me garantissent la possibilité de réaliser les opérations nécessaires pour implémenter un gestionnaire de connexion tout en le rendant sécurisé à l'aide de middlewares et réalisant les tâches demandées à l'aide de l'ODM Mongoose. (Inscription, connexion, déconnexion, vérification de la validité de la session)

- Sécurité

C'est un sujet capital. Si je manipule des données personnelles que les utilisateurs sont prêts à me confier, je me devais de garantir une sécurité la plus optimale possible afin de les préserver de toutes fuites éventuelles, mais aussi afin de se prémunir contre les attaques malveillantes. De ce fait, j'ai entrepris plusieurs actions :

  - *Hachage du mot de passe*

Comme l'on a pu le voir dans le modèle users, le champ password renseigné par l'utilisateur est haché avant la sauvegarde et avant la mise à jour de ses informations.

```
// Hachage du mot de passe avant la sauvegarde
UserSchema.pre("save", async function () {
  if (this.isModified("password")) {
    const salt = await bcrypt.genSalt();
    this.password = await bcrypt.hash(this.password, salt);
  }
});

// Hachage du mot de passe avant la mise à jour
UserSchema.pre("findOneAndUpdate", async function () {
  if (this._update.password) {
    const salt = await bcrypt.genSalt();
    this._update.password = await bcrypt.hash(this._update.password, salt);
  }
});
```

Dans un premier temps, la fonction génère un grain de sel (salt) grâce à la fonction .gensalt de bcryptjs puis dans un second temps, à l'aide du grain de sel et de la fonction hash de bcryptjs, hash le mot de passe renseigné par l'utilisateur.

De cette façon, les mots de passe stockés dans la base de données sont une chaine de caractère qui n'a rien à voir avec le mot de passe réellement tapé par l'utilisateur.

Ensuite, à la connexion, la fonction comparePassword est en mesure de vérifier si le mot de passe saisi dans le formulaire de connexion correspond bien a celui saisi par l'utilisateur dans la base de données avant qu'il ne soit haché à l'aide de la fonction .compare de bcryptjs.

```
// Méthode pour comparer les mots de passe
UserSchema.methods.comparePasswords = async function (candidatePassword) {
  const isMatch = await bcrypt.compare(candidatePassword, this.password);
  return isMatch;
};
```

Cette fonction renvoie un booléen en fonction du résultat de la comparaison.

En implémentant le stockage de mot de passe hachés je veillé à :

- **Protéger mes utilisateurs en cas de fuite de données.** Si ma base de données est compromise, les mots de passes ne pourront pas être récupérer facilement. Garantissant une sécurité supplémentaire aux utilisateurs.
- **L'irréversibilité du hachage.** Le hachage est un processus plus sûr que le cryptage pour transformer un mot de passe. En effet, avec le cryptage, il est possible de décrypter le mot de passe si la clé de cryptage est connue, alors qu'avec un algorithme de hachage, il est difficile d'amorcer le processus inverse.
- **Minimiser les risques internes.** Si un éventuel administrateur est amené à consulter les données stockées par la suite, il ne pourra pas obtenir les mots de passe, de façon intentionnelle ou non. Ainsi, il pourra consulter les données nécessaires sans que la confiance des utilisateurs ne soit compromise tout en limitant le risque d'évènement malencontreux ou malveillant.
- **Me conformer aux réglementations RGPD.** La règlementation RGPD exige de ne pas stocker les mots de passe des utilisateurs en clair dans les bases de données (Ce qui est une faille de sécurité majeure).

JWT Token

Le JWT Token me permet de vérifier la validité d'une session de connexion d'un utilisateur.

```
const token = jwt.sign({ id, email }, process.env.JWT_SECRET, {
  expiresIn: process.env.JWT_LIFETIME,
});
```

La fonction jwt.sign de json web token permet de générer un jeton d'accès signé avec dans son playload l'id de l'utilisateur et son email. Le jeton est valable pendant une durée définie dans la propriété JWT_LIFETIME qui est présente dans mon fichier d'environnement .env et il est signé à l'aide d'une autre propriété de mon fichier d'environnement, JWT_SECRET.

Cette méthode est appelée lors de ma fonction de login comme vue précédemment après avoir vérifier que le mot de passe correspond à l'email renseigné.

Je le passe ensuite dans un cookie qui sera ajouté à la réponse de mon controller afin de me permettre plus tard de vérifier la validité du token, et donc de la session en cours de l'utilisateur.

```
res.cookie("accessToken", token, {
  HttpOnly: true,
  secure: process.env.NODE_ENV === "production",
  signed: true,
  expires: new Date(Date.now() + oneDay),
});
```

- *Protéger les headers des requêtes*

Afin de protéger mon application Express de nombreuses menaces communes, j'ai utilisé helmet pour protéger les en-têtes des requêtes http comme recommandé par Express (*https://expressjs.com/fr/advanced/best-practice-security.html*)

```
app.use(helmet());
```

Cette ligne me permet d'employé un middleware, en l'occurrence l'emploi d'helmet. Helmet intervient alors sur chaque requête entrante et sur chaque réponse envoyée, veillant a combler les failles de sécurité qui peuvent leur être liées.

- *Limiter le nombre de requêtes sur une durée donnée*

```
app.use(
  rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    limit: 1000, // Limit each IP to 1000 requests per `window` (here, per 15 minutes).
    standardHeaders: "draft-7", // draft-6: `RateLimit-*` headers; draft-7: combined `RateLimit` header
    legacyHeaders: false, // Disable the `X-RateLimit-*` headers.
  })
);
```

Un autre middleware intervient juste après. Il permet de limiter le nombre de requête envoyé par une machine sur une certaine durée à l'aide de rateLimit, (fonction provenant du module Express Rate Limit).

Cela empêche les attaques de type DDos qui vise à saturer le service délivrer par les applications ciblées afin de dégrader leurs performances voire de les pousser à l'arrêt causant alors une interruption complète du service.

- *Sanitariser les données entrantes*

Afin d'être sûr que les informations envoyées à ma base de données ne représentent pas un risque pour elle, j'ai implémenté l'usage de mongo sanitize. Ce module me permet de retirer des requêtes reçues tous les caractères utilisés par mongoDB. Ainsi il est impossible de faire de l'injection NoSQL dans ma base de données.

```
app.use(mongoSanitize());
```

Il s'agit une nouvelle fois d'un middleware intervenant juste après rateLimit.

Avec ces mesures de sécurité implémentés, mon application couvre de nombreuses failles de sécurité potentiels.

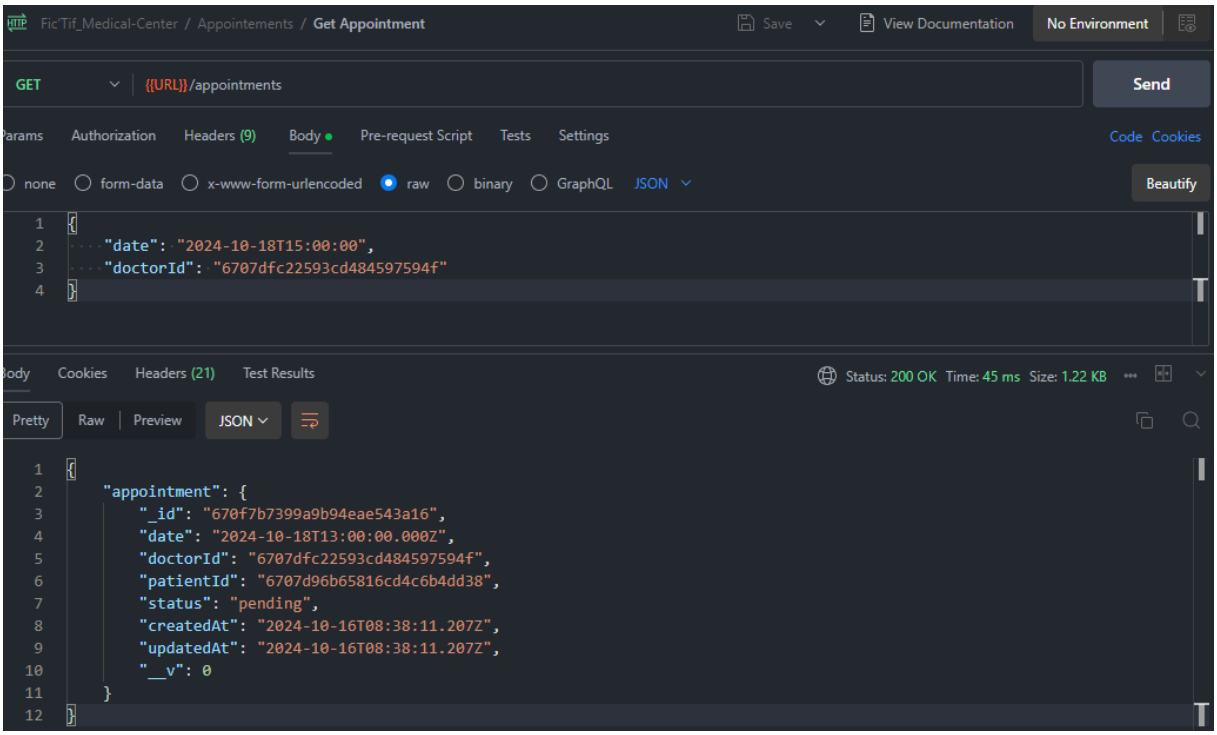
Remarque : La sécurité est un enjeu important auquel une veille permanente est nécessaire. C'est pour cette raison que je me suis penché sur ce sujet et, bien qu'il soit immense, m'a intéressé fortement.

(J'ai notamment appris la notion de White Hat et l'existence de « compétition » de hack afin que les failles soient découvertes et corrigées avant qu'une personne malveillante ne s'y engouffre)

Simulation des routes backend

- Utilisation de PostMan

Une fois mes premières routes créent de bout en bout, j'ai utilisé PostMan pour tester leurs comportements. (*https://www.postman.com/*)

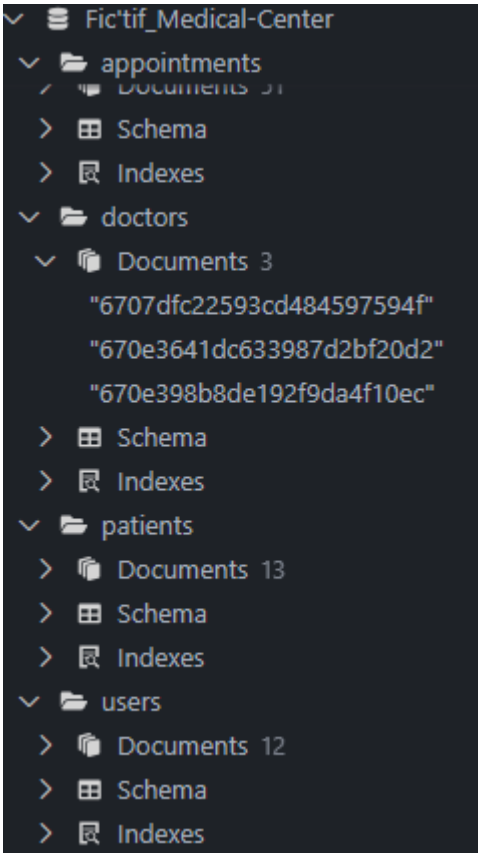


Dans l'exemple ci-dessus, j'effectue un appel à mon API sur la route appointments en méthode GET avec une date et un id de docteur dans le corps de la requête et je vérifie que mon API me retourne une réponse en status 200 avec les informations attendues.

En effectuant ces étapes pour chaque route, je m'assure que mon backend est prêt pour délivrer les services que j'ai besoin.

- Premiers peuplements de la base MongoDB

Lors de ses tests, ma base de données mongoDB a commencé à accueillir ses premiers jeux de données qui m'ont ensuite servi de support pour réaliser la suite du projet.



Remarque : L'on peut voir ici que le nombre de users ne correspond pas au nombre de patients + doctors, c'est lié au fait qu'il existe des reliquats « d'anciens » patients quand la fonction de suppression d'user ne prenait pas encore en compte la suppression des patients/doctors associé ou quand celle-ci rencontrait des problèmes.

Il est prévu que je nettoie ma base de données.

Création du frontend

Après m'être assuré que le backend été prêt, j'ai pu m'attaquer au frontend.

- Technologies employées

J'ai utilisé le framework React pour son ses fonctionnalités comme la gestion des states et l'utilisation de composants réutilisables. React, couplé à Node.js et son large choix de modules (npm), me permettent alors de créer une application fluide et adaptable. J'ai également fait le choix d'utiliser Tailwind car c'est un outil qui semble avoir le vent en poupe dans le domaine du développement web frontend et, après apprentissage et usage au sein de ce projet, offre des possibilités de stylisation de composant très rapide et propre.

- Modules et dépendances
  - *Daisy UI*

Utilisant Tailwind, j'ai également fait le choix d'opter pour un framework d'interface utilisateur. Le but étant d'avoir un outil pour m'aider à réaliser une interface modern, cohérente et élégante. J'ai donc fait le choix de Daisy UI (*https://daisyui.com/*), notamment pour sa simplicité d'utilisation. Le cadre posé par Daisy UI m'a permis de mieux me rendre compte de ce que doit être un design system. (Couleurs prédéfinies, cohérence du comportement et du design des composants, etc…)

```
<button className="btn btn-primary text-base-100 transition-all duration-500 hover:btn-secondary">
  Connection
</button>
```

- *Axios*

Afin de communiquer avec mon API, j'ai décidé d'utiliser le module Axios (*https://www.npmjs.com/package/axios*). Il me permet d'effectuer des requêtes http et prend en charge la gestion d'erreur.

- *React-router-dom*

Afin d'établir la navigation au sein de mon application, j'ai opté pour l'usage de react router dom au lieu de l'emploi de simples liens (Balise <a> en HTML avec la propriété href). En effet, mon application utilisant react, il serait plus intéressant d'employer un router qui embarque avec lui de nombreuses fonctionnalités (Comme useFetcher, useNavigate et useParams). D'autant plus que react router dom permet d'atteindre d'autres parties du site sans effectuer des rechargements complets de l'application à l'aide de la balise Link et NavLink permet de styliser un lien présent dans la page si la route qui lui est associée est active.

- *React-big-calendar*

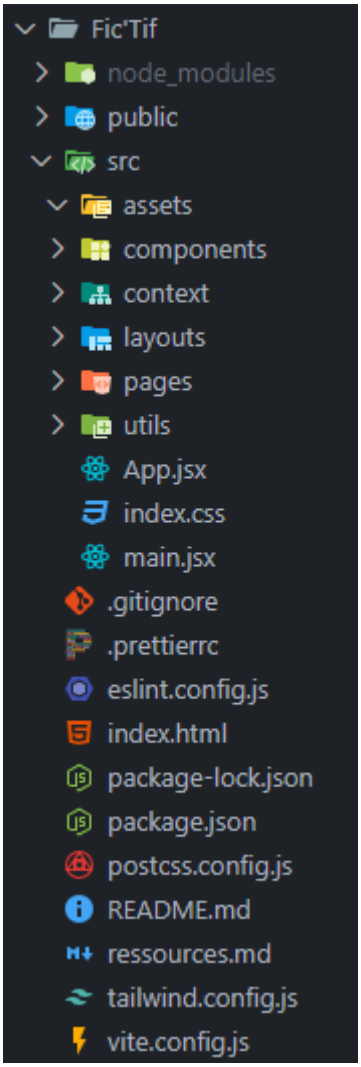
Comme évoqué plus tôt, je me devais de fournir un affichage clair pour la gestion des rendez-vous. Après quelques essais avec d'autres modules, j'ai fini par opter pour React-big-calendar. Cette librairie me semblait simple à utiliser tout en restant personnalisable. Je décris son emploi un peu plus loin.

- *React-toastify*

Afin de fournir aux utilisateurs un affichage clair et intuitif des erreurs ou une validation des actions réalisées, j'ai utilisé react-toastify. Il s'accorde parfaitement à ma volonté de fournir une interface simple, intuitive et élégante aux utilisateurs de mon site sans en déranger la structure.

```
const logout = async () => {
  try {
    await axios("/api/v1/auth/logout", {
      method: "POST",
      credentials: "include",
    });

    setUser(null);
    setIsLogged(false);

    toast.success("Vous êtes maintenant déconnecté");
  } catch (error) {
    toast.error(
      error?.msg
        ? error.msg
        : "Une erreur est survenue lors de la déconnexion",
    );
  }

  return redirect("/");
};
```

- Structure



J'ai repris la structure amorcée par le framework react suite à son installation, puis j'ai rassemblé les éléments crées pour l'application dans des catégories en fonction de leurs natures (Pages, composants, contextes, layout ou encore éléments utilitaires).

De cette façon, j'ai obtenu une structure claire et maintenable.

- Création des routes de navigations

```
const App = () => {
  const router = createBrowserRouter([
    {
      path: "/",
      element: <SharedLayout />,
      children: [
        {
          index: true,
          element: <Home />,
          errorElement: <div>Error</div>,
        },
        {
          path: "doctors",
          element: <Services />,
          errorElement: <div>Error</div>,
        },
        {
          path: "appointment/:id?",
          element: <Appointment />,
          errorElement: <div>Error</div>,
          action: appointmentAction,
        },
        {
          path: "profile",
          element: <Profile />,
          errorElement: <div>Error</div>,
        },
        {
          path: "doctorPage",
          element: <DoctorPage />,
          errorElement: <div>Error</div>,
        },
      ],
    },
    {
      path: "login",
      element: <Login />,
    },
    {
      path: "register",
      element: <Register />,
      action: registerAction,
    },
  ]);
```

J'ai conçu un router à l'aide de react-router afin de mettre en place la navigation dans mon application.

Ici, on peut voir qu'à la racine j'ai mon composant de plan commun qui s'affiche (SharedLayout) et que ce dernier possède plusieurs enfants. Ces enfants seront appelés par le plan commun selon la page que l'utilisateur consulte afin d'être affiché au sein du plan.

Par exemple, l'index affichera le composant <Home />, c'est-à-dire que lorsque la navigation sera active sur la page http://DOMAINE/ , ce sera le composant Home qui sera intégré au plan commun et sera visible sur le navigateur. Cependant s'il s'agit de la page des rendez-vous via le lien http://appointment/ , alors ce sera le composant Appointment qui sera affiché.

Ce qui implique que la page login et register ne sont pas imbriqué dans le layout commun, il bénéficie de leur propre structure.

L'ensemble est implémenté à la racine de l'application avec le rendu du composant App :

```
return (
  <main className="relative flex min-h-dvh flex-col items-center bg-base-300">
    <AuthProvider>
      <ThemeContext.Provider value={{ theme, toggleTheme }}>
        <ToastContainer
          position="bottom-center"
          autoClose={5000}
          hideProgressBar={false}
          newestOnTop={false}
          closeOnClick
          rtl={false}
          pauseOnFocusLoss
          draggable
          pauseOnHover
          theme={theme === "cupcake" ? "light" : "dark"}
          transition={Bounce}
        />
        <RouterProvider router={router} />
      </ThemeContext.Provider>
    </AuthProvider>
  </main>
);
```

Le composant RouterProvider agit alors comme un contexte dans toute l'application et gèrera à présent la navigation et assurera le rendu des pages en fonction des routes précédemment définies à l'aide la fonction createBrowserRouter de react router.

Remarque : J'ai positionné le router et l'ensemble de son rendu dans deux contextes. Ces derniers me permettent, d'une part, de gérer le thème (Clair/Sombre) des utilisateurs (ThemeContext), de l'autre gérer les sessions d'authentification (AuthProvider).

J'ai également inséré au même niveau le composant ToastContainer avec l'ensemble de ses props qui permettent de le paramétrer comme souhaité. De cette façon je m'assure que les toasts peuvent être visibles partout sur mon application.

- Création du layout

```jsx
import { Outlet, useNavigation } from "react-router-dom";
import Navbar from "../components/Navbar.jsx";
import SideMenuDrawer from "../components/SideMenuDrawer.jsx";
import Loading from "../components/Loading.jsx";
import Footer from "../components/Footer.jsx";

const SharedLayout = () => {
  const navigation = useNavigation();
  const isPageLoading = navigation.state === "loading";

  return (
    <>
      <Navbar />
      <SideMenuDrawer>
        {isPageLoading ? <Loading /> : <Outlet />}
      </SideMenuDrawer>
      <Footer />
    </>
  );
};
export default SharedLayout;
```

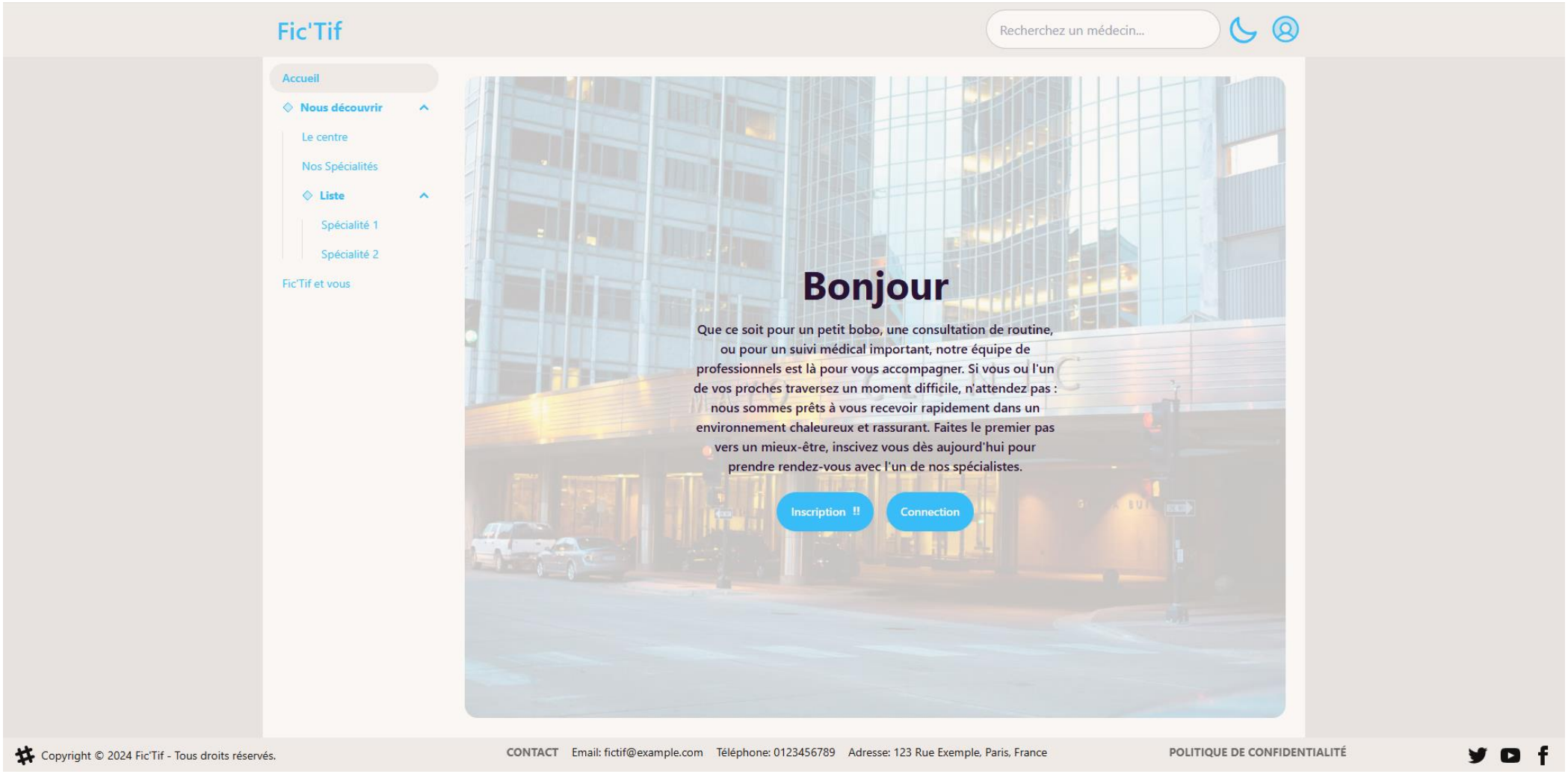
Mon layout est composé de 4 parties, une est la barre de navigation (Navbar), la seconde est le menu tiroir (SideMenuDrawer), la troisième est la zone d'affichage de contenu (Outlet) la dernière accueille le footer.

La barre de navigation accueille le titre de l'application, une barre de recherche pour trouver plus facilement le médecin que l'on souhaite, le bouton toggle pour changer de thème (Celui-ci se trouve sur la barre de navigation directement, ou dans le menu déroulement des options profil quand l'affichage est plus petit) et enfin un bouton faisant apparaître les options de profil (Connexion, inscription et le bouton toggle de changement de thème si l'écran est en format mobile ou tablette)

Le sideMenuDrawer est un composant importé du framework d'interface utilisateur Daysi UI. Il permet d'obtenir un menu rétractable selon certaine condition. De mon côté je n'ai pas souhaité offrir la possibilité de l'ouvrir ou le fermé en affichage desktop, cependant en affichage mobile ou tablette, un bouton apparait en bas à gauche de l'écran et permet de l'ouvrir et ainsi permettre la navigation à l'utilisateur. J'ai donc du le customisé pour qu'il adopte le comportement que je souhaitais et assure l'affichage recherché.

Le layout présenté ici utilise la fonction de react router pour surveiller la navigation. Si le chargement de la page a lieu, le state de isPageLoading change et passe a « true », auquel cas le composant Loading sera affiché au lieu de Outlet.

Outlet est un composant issu de react router qui schématise l'ensemble des enfants possibles et ses enfants seront rendus à l'endroit où est positionné Outlet en fonction de la page en cours.



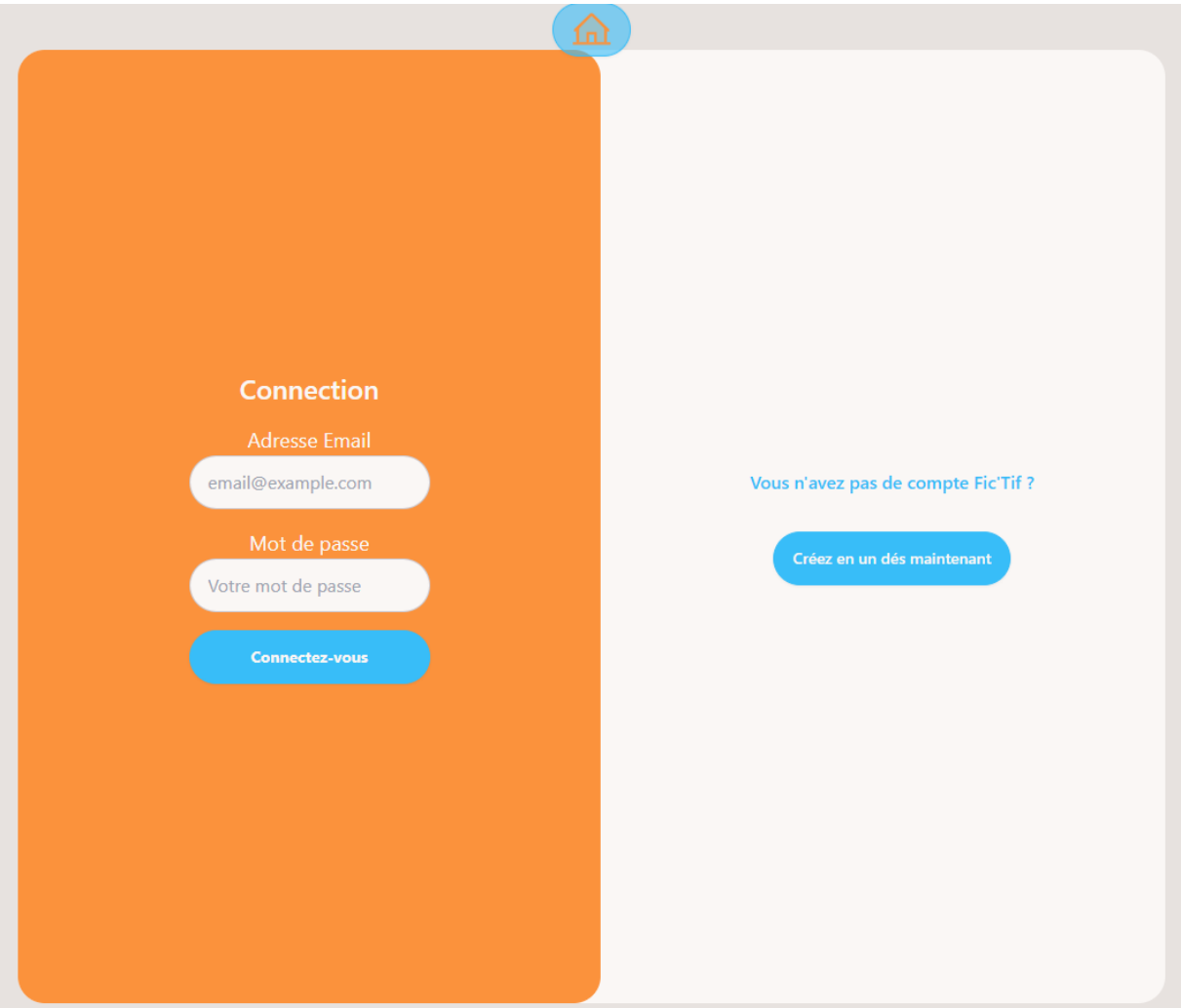
L’image représente le plan général du site

J’ai veillé à ce que le plan de base soit responsive et s’adapte à tout type d’écran. Utilisant les breakpoints prédéfini par tailwind.

Fonctionnalités

- Login/Logout
  - *Page de Login/Logout*

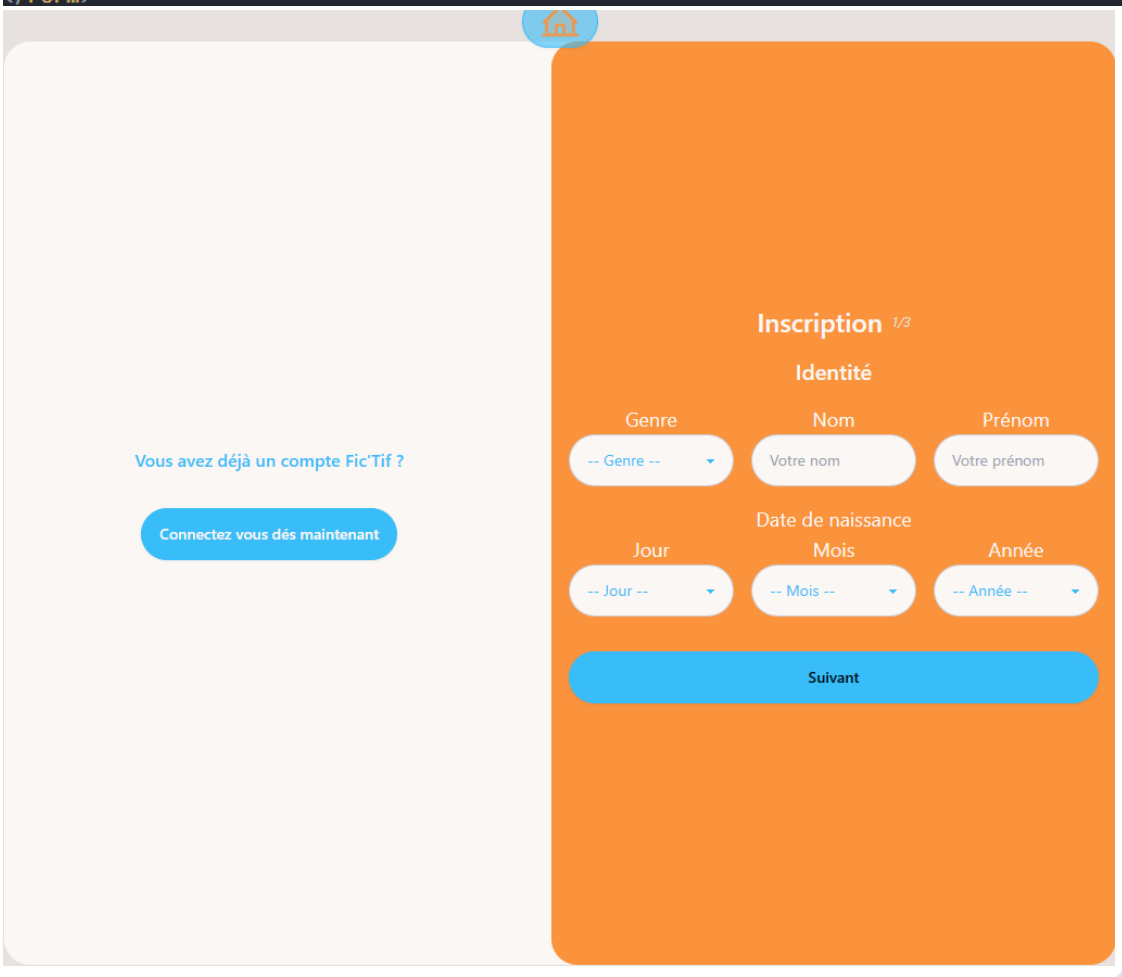
Pour prendre rendez-vous, il faut se connecter. Pour ce faire j'ai opté pour une page indépendante de mon plan qui permettrait aux utilisateurs de se connecter ou de s'inscrire dans le cas où ils n'ont pas encore de compte.

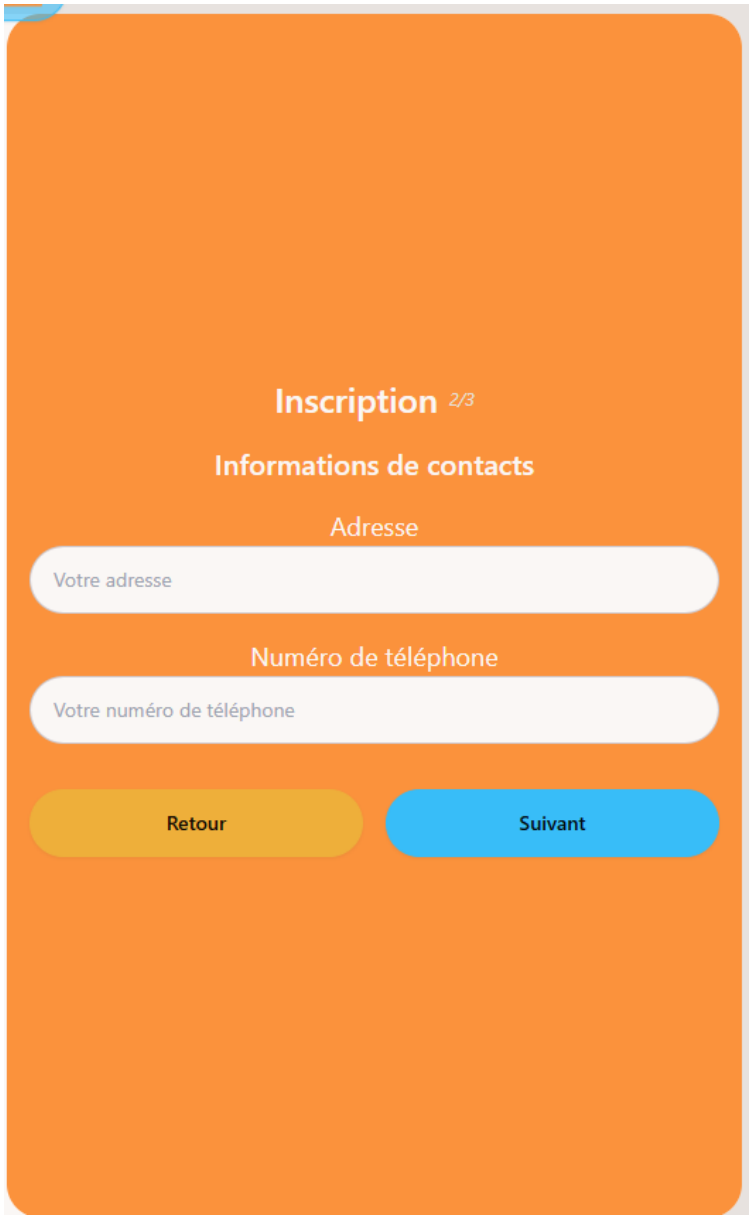


Le formulaire de Login est classique, un champs email, un champs mot de passe et un bouton d'envoi de formulaire. Lors de la soumission, la fonction handleLogin est appelée. Elle permet de récupérer les données du formulaire et d'appeler la fonction login du contexte d'authentification.

```
const handleLogin = async (event) => {
  event.preventDefault();

  const data = new FormData(event.target);
  const formData = Object.fromEntries(data.entries());

  await login(formData.email, formData.password);

  navigate("/");
};
```

```jsx
<Form className="flex flex-col gap-4" onSubmit={handleLogin}>
  <h3 className="text-2xl font-semibold">Connection</h3>
  <div className="flex flex-col">
    <label htmlFor="email" className="text-lg">
      Adresse Email
    </label>
    <input
      name="email"
      type="email"
      placeholder="email@example.com"
      className="input input-bordered w-full max-w-xs text-primary"
    />
  </div>
  <div className="flex flex-col">
    <label htmlFor="password" className="text-lg">
      Mot de passe
    </label>
    <input
      name="password"
      type="password"
      placeholder="Votre mot de passe"
      className="input input-bordered w-full max-w-xs text-primary"
    />
  </div>
  <input
    type="submit"
    value="Connectez-vous"
    className="btn btn-primary font-bold text-base-100 hover:border-accent hover:bg-accent"
  />
</Form>
```





Pour des soucis de lisibilité dans le document, je ne représente ici que le formulaire d'inscription, le bouton de connexion étant toujours accessible à chaque étape sur le volet de gauche.

```
const [currentStep, setCurrentStep] = useState(1);
const [data, setData] = useState({
  gender: "",
  lastname: "",
  firstname: "",
  day: "",
  month: "",
  year: "",
  address: "",
  phone: "",
  email: "",
  password: "",
});
```

Dans mon formulaire d'inscription, j'ai deux états définis à l'aide de useState. Le premier, currentStep, gère les étapes en cours (Par défaut 1) et le second, data, est un objet qui va acceuillir les données que l'utilisateur va saisir au fur et à mesure de son parcours d'inscritpion.

```
const handleChange = (e) => {
  const { name, value } = e.target;
  setData({ ...data, [name]: value });
};
```

La fonction handleChange sera appelée chaque fois qu'un input se verra attribué une nouvelle valeur par l'utilisateur, mettant à jour le state data, garantissant de toujours avoir les dernières données saisies par l'utilisateur à l'aide d'input contrôlés.

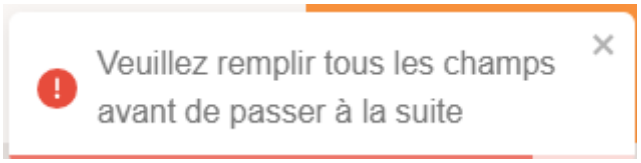
```
const validateCurrentStep = () => {
  if (currentStep === 1) {
    return (
      data.gender &&
      data.lastname &&
      data.firstname &&
      data.day &&
      data.month &&
      data.year
    );
  }
  if (currentStep === 2) {
    return data.address && data.phone;
  }
  if (currentStep === 3) {
    return data.email && data.password;
  }
  return false;
};

const nextStep = () => {
  if (validateCurrentStep()) {
    setCurrentStep((prev) => Math.min(prev + 1, 3));
  } else {
    toast.error(
      "Veuillez remplir tous les champs avant de passer à la suite",
    );
  }
};

const prevStep = () => {
  setCurrentStep((prev) => Math.max(prev - 1, 1));
};
```

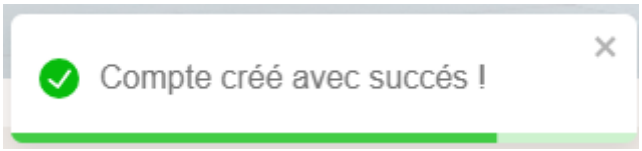
La fonction validateCurrentStep vérifie que les inputs concernées par l'étape en cours ont bien des valeurs qui leur sont associés. La fonction renvoie ensuite un booléen en fonction du résultat de ce test.

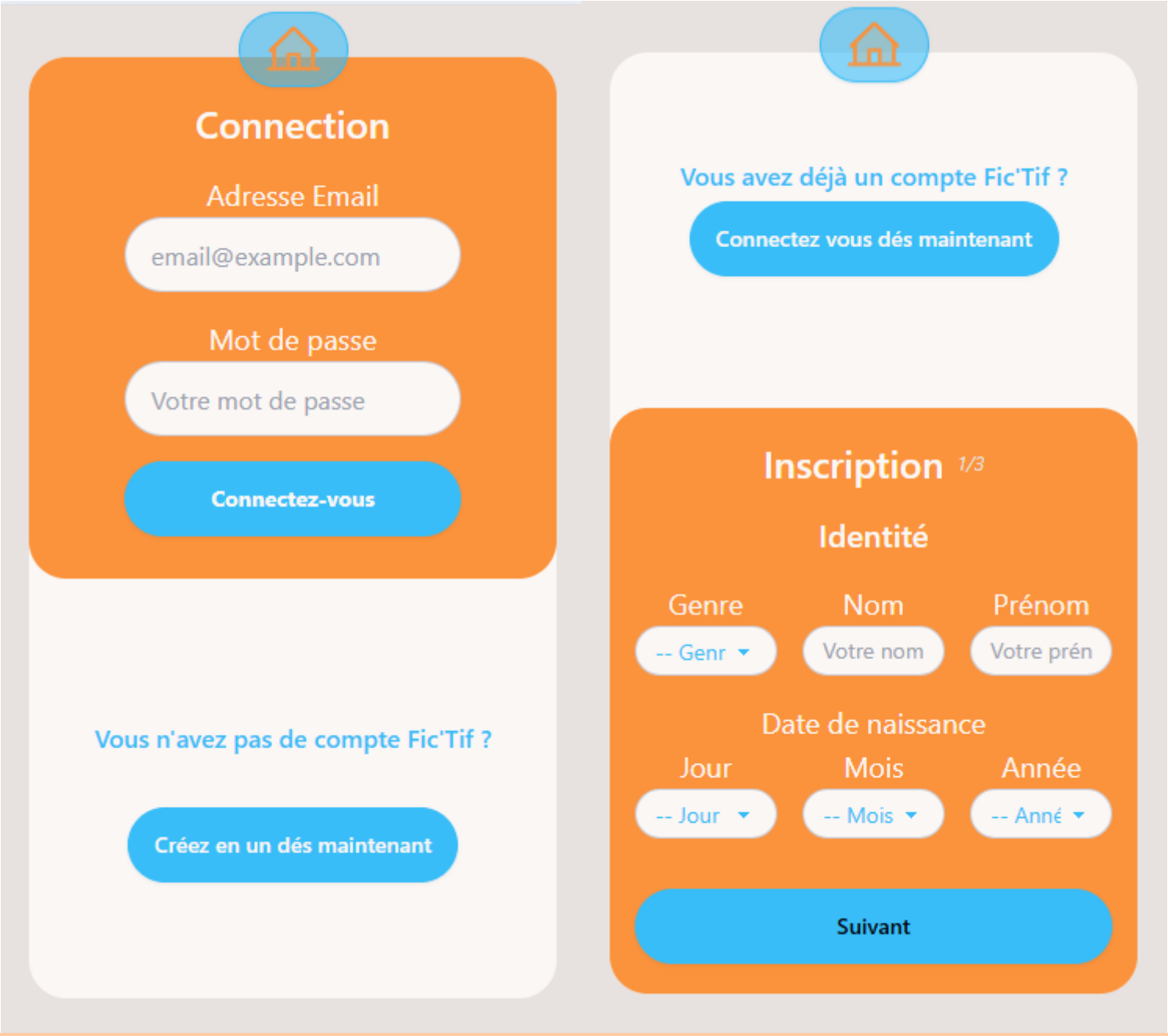
La fonction nextStep permet de passer à l'étape suivante à condition que l'appel de validateCurrentStep soit positif, ne pouvant pas aller plus haut que 3 (Pour la troisième étape), ou afficher un toast d'erreur dans le cas ou des valeurs sont manquantes.

La fonction prevStep permet de revenir à l'étape précédente en déincrémentant la valeur de currentStep de 1, ne pouvant pas aller plus bas que 1 (Pour la première étape).

De cette façon, j'ai construis un formulaire qui est capable de suivre plusieurs étapes sans perdre les informations saisies par l'utilisateur et en lui fournissant une expérience correcte à l'aide de messages présent dans un toast afin qu'il soit au courant des problématiques rencontrées.



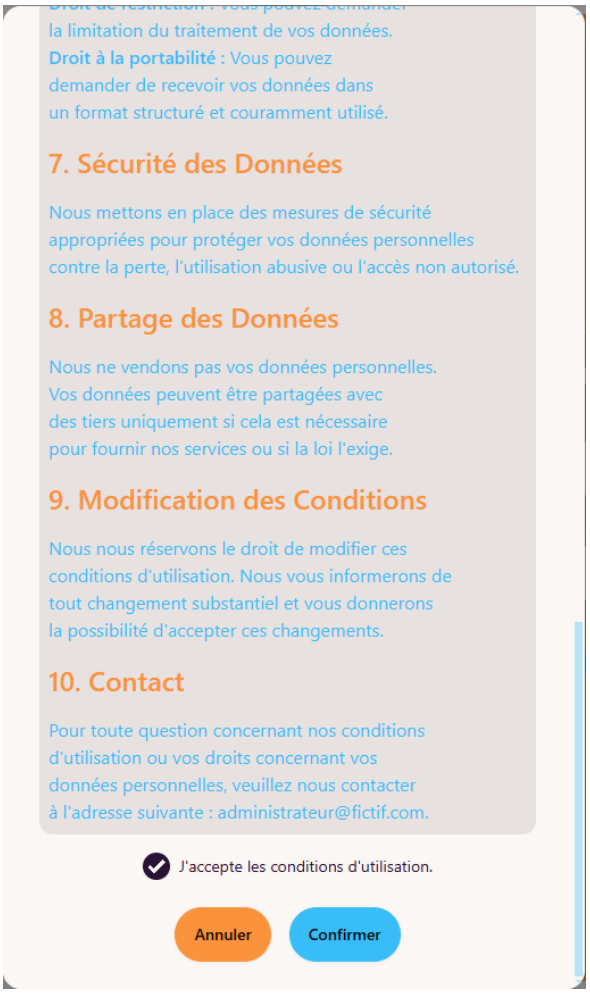




Je me suis assuré que les deux formulaires puissent s’afficher proprement dans le cas d’une navigation mobile à l’aide des breakpoints de base de tailwind. Rendant l’affichage de ces pages indépendantes responsive.

- *Modale RGPD*

A la fin du formulaire d'inscription, une modale s'affiche demandant à l'utilisateur s'il accepte les conditions d'utilisations.



Si la checkbox n'est pas cochée alors que l'utilisateur tente de finaliser son inscription via le bouton confirmer, un toast apparait lui indiquant les prérequis pour finaliser son inscription.

```
const handleSubmit = (event) => {
  event.preventDefault();

  modalRef.current.showModal();
};

const confirmSubmission = async () => {
  if (!isAgreed) {
    toast.error(
      "Vous devez accepter les conditions d'utilisation pour poursuivre l'inscription.",
    );
    modalRef.current.close();
    return;
  }

  fetcher.submit(data, {
    method: "post",
    action: "/register",
  });

  modalRef.current.close();
  return redirect("/");
};
```

Une fois les conditions acceptés et l'inscription confirmée, l'action de la page Register définie à l'aide de react router est appelée.

```
export const action = async ({ request }) => {
  const formData = await request.formData();
  const {
    email,
    password,
    firstname,
    lastname,
    day,
    month,
    year,
    gender,
    address,
    phone,
  } = Object.fromEntries(formData);

  try {
    const { data } = await apiClient.post("/api/v1/auth/register/patient",
      birthDate: `${month}/${day}/${year}`,
      firstName: firstname,
      lastName: lastname,
      address,
      phoneNumber: phone,
      email,
      password,
      gender,
    });

    toast.success(data.msg);

    return redirect("/");
  } catch (error) {
    const errorMessage =
      error.response?.data?.msg || "Erreur lors de la création du compte";
    toast.error(errorMessage);
    return null;
  }
};
```

La fonction récupère alors les informations du formulaire, les organises et les envois vers la route back /api/v1/auth/register/patient. Si la réponse reçue est correcte (Code 200), un toast de validation indiquera à l'utilisateur que tout c'est bien passé. Sinon le toast sera un toast d'erreur qui lui indiquera l'erreur envoyée par le backend.

```
const registerPatient = async (req, res) => {
  const role = "patient";
  const user = await usersService.create({ ...req.body, role });
  const userId = user._id;
  const patient = await patientService.create(userId);
  const token = user.createAccessToken();
  res
    .status(StatusCodes.CREATED)
    .json({ user, token, msg: "Compte créé avec succés !" });
};
```

- *Contexte de l'utilisateur*

```
const login = async (email, password) => {
  try {
    await apiClient.post("/api/v1/auth/login", {
      email,
      password,
    });

    setIsLogged(true);
  } catch (error) {
    if (error.response) {
      if (error.response.status === 401) {
        toast.error("Identifiants incorrects. Veuillez réessayer.");
      } else {
        toast.error(
          error.response.data.message || "Une erreur est survenue.",
        );
      }
    } else {
      toast.error("Erreur de connexion. Vérifiez votre connexion réseau.");
    }
  }
};
```

La fonction login envoie les informations saisies par l'utilisateur au Backend qui va vérifier que ses informations de connexion correspondent bien à un utilisateur présent dans la base de données. Si c'est le cas, il change l'état de isLogged à true.

IsLogged est un state qui permet de savoir si une session est en cours.

Si la tentative de connexion est en échec, il renvoi un toast d'erreur avec un message personnalisé ou le message envoyé par le serveur.

```
const checkIfLoggedIn = async () => {
  try {
    const { data } = await apiClient.get("/api/v1/auth/isLogged");

    if (data?.user) {
      if (!data.user.avatar) {
        const seed = `${data.user.firstName}${data.user.lastName}`;
        const avatarSvg = createAvatar(thumbs, {
          seed,
          backgroundColor: ["#ffffff"],
          radius: 50,
        });
        data.user.avatar = avatarSvg;
      }

      return data;
    }

    return null;
  } catch (error) {
    if (error.response && error.response.status === 401) {
      return null;
    }
    console.error(
      "Erreur lors de la vérification de l'authentification",
      error,
    );
    return error;
  }
};
```

checkIfLoggedIn est une fonction qui va contacter la route de l'API auth/isLogged. Cette route va vérifier la validité du token envoyé dans le cookie de la requête.

```
const isLogged = async (req, res) => {
  const email = req.user.email;

  const user = await usersService.isLogged(email);

  if (!user) {
    return res
      .status(StatusCodes.NOT_FOUND)
      .json({ msg: "Utilisateur non trouvé" });
  }
  res.status(StatusCodes.OK).json({ user, msg: "Utilisateur connecté" });
};
```

```
const isLogged = (email) => {
  return User.findOne({ email }).select("-password");
};
```

Puis, elle va vérifier que les informations de l'utilisateur récupéré ne comprennent pas un avatar personnalisé. S'il n'en a pas, la fonction va générer un avatar à l'aide du nom et du prénom de l'utilisateur et retourner l'ensemble des données de l'utilisateur.

```
useEffect(() => {
  let isMounted = true;

  const loadUser = async () => {
    const data = await checkIfLoggedIn();
    if (isMounted) {
      setUser(data?.user || null);
    }
  };

  loadUser();

  return () => {
    isMounted = false;
  };
}, [isLogged]);
```

L'usage du useEffect (Hook qui me permet d'effectuer un nouveau rendu du composant sous certaines conditions) me permet d'appeler checkIfLoggedIn a chaque changement d'état de isLogged.

En effet, ce useEffect va récupérer les informations retournées par CheckIfLoggedIn et les associés au state de l'utilisateur.

De cette façon, j'ai accès aux informations de l'utilisateur partout dans mon site puisqu'il s'agit ici d'un contexte qui englobe mon site. (useContexte).

La fonction logout permet de nettoyer les valeurs du contexte, mettant à null les valeurs stockées dans user et à false le state isLogged. Aussi, elle envoie avant une requête à l'API sur la route auth/logout qui veille à supprimer le JWT token existant sur le backend.

```
const logout = async (req, res) => {
  res.clearCookie("accessToken", {
    httpOnly: true,
    secure: process.env.NODE_ENV === "production",
    signed: true,
  });

  res.status(StatusCodes.OK).json({ msg: "Déconnexion réussie." });
};
```

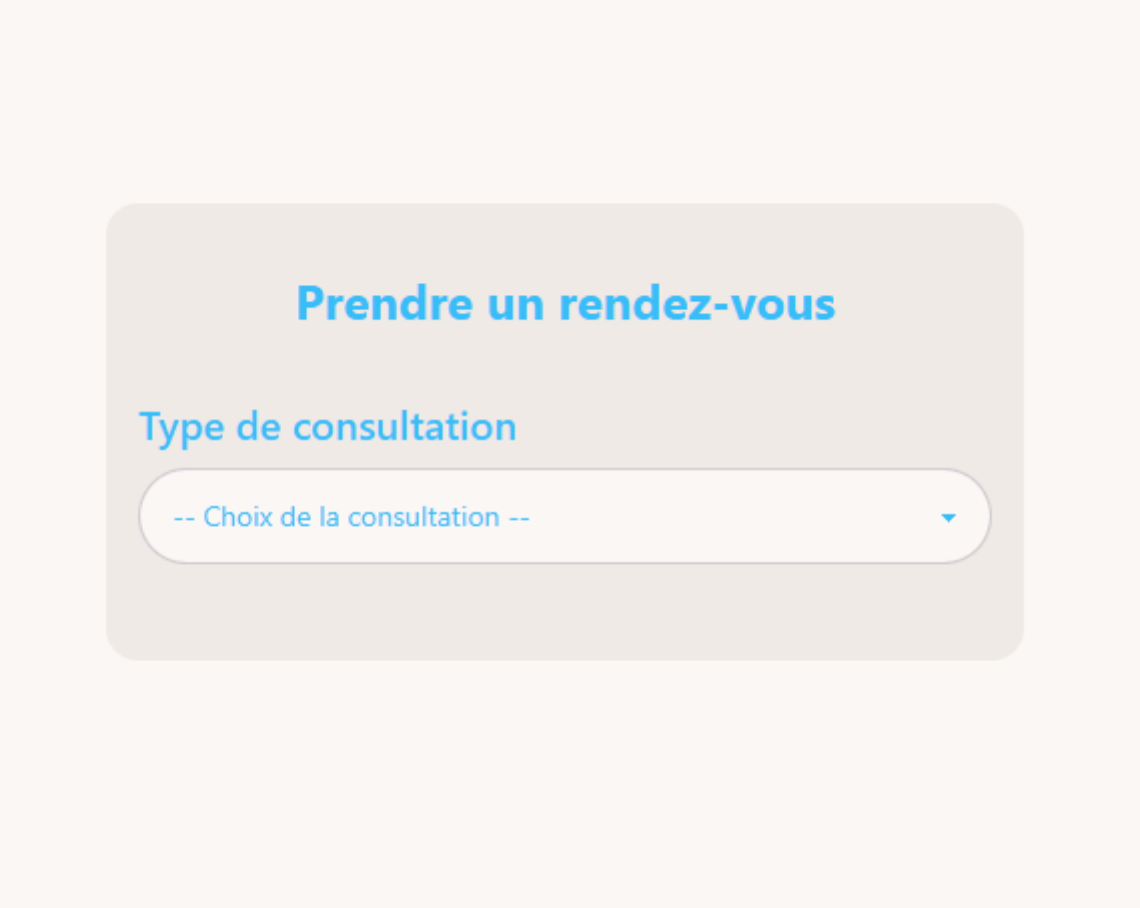
```
const logout = async () => {
  try {
    await apiClient.post("/api/v1/auth/logout", {
      credentials: "include",
    });

    setUser(null);
    setIsLogged(false);

    toast.success("Vous êtes maintenant déconnecté");
  } catch (error) {
    toast.error(
      error?.msg
        ? error.msg
        : "Une erreur est survenue lors de la déconnexion",
    );
  }

  return redirect("/");
};
```

- Prise de rendez-vous

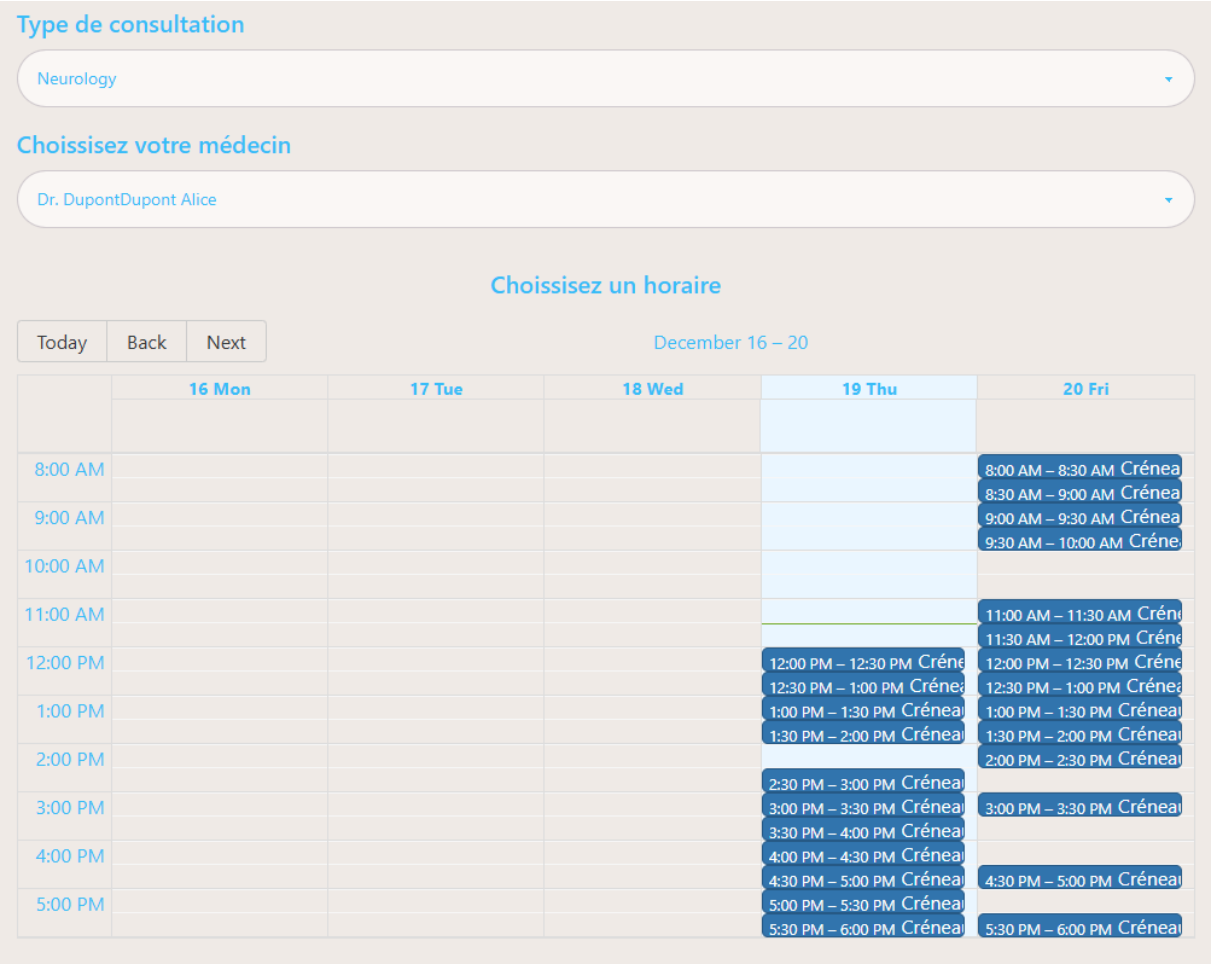
La prise de rendez-vous est la fonctionnalité primaire de l'application. Je voulais une interface simple et claire, ne laissant pas de place au doute.

J'ai donc opté pour un formulaire qui s'agrandit au fur et à mesure que l'utilisateur effectue ses choix pour avoir son rendez-vous. Comme pour le formulaire d'inscription, j'ai des states qui vérifie la où se situe l'utilisateur dans la complétion du formulaire.

```
const [choiceDoctors, setChoiceDoctors] = useState(false);
const [appointment, setAppointment] = useState(false);
```



Une fois la spécialité et le médecin sélectionné, un calendrier généré à l'aide de big-calendar s'affiche et est peuplé par tous les créneaux de 30 minutes possible pour les 6 prochains mois en retirant les rendez-vous déjà pris avec ce médecin de la liste, créant ainsi des trous dans le planning des disponibilités.



En cliquant sur un créneau disponible, l'utilisateur pourra confirmer son choix à l'aide d'une modale récapitulative.

Remarque : Le calendrier est affiché en anglais et la date dans la modale également. Effectuer des changements pour les passer en français sont des améliorations qui sont dans ma liste des tâches à effectuées, tout comme le fait de permettre une auto-complétions du formulaire lorsque l'on arrive sur cette page à l'aide du bouton « prendre un rendez-vous » de la page DoctorPage (Qui a pour but d'afficher les informations d'un médecin, cette page est accessible via la recherche de médecin)

Tests

Documentation

Déploiement

Mon application est aujourd'hui en ligne. Comme ma volonté initiale était d'avoir un frontend séparé de mon backend, j'ai donc mon frontend hébergé sur netlify (*https://www.netlify.com/*) et mon backend hébergé sur render (*https://render.com/*). J'ai également configuré un DNS (Domain Name System) via cloudflare pour l'associé à mon domaine existant et je lui ai dédié un sous-domaine.

Mon projet Fic'tif est donc accessible à l'adresse : https://fictif.freyza.net/

La trêve de novembre

Problèmes rencontrés

MCD

Conception du backend

Login/Logout

Conclusion

Technologies balayées par le projet

- Frontend
- Backend

Axes d'amélioration

Et si c'était à refaire ?!

Bilan

- Remerciements
- Le mot de la fin