



Contexte GSB

Application de gestion des visites et des médecins

Partie 2 - Page d'accueil et de gestion des médecins

Mise en place de la page d'accueil

Maintenant que votre application permet à un visiteur (commercial) de se connecter, nous allons nous occuper de son interface de gestion.

Ce dernier doit pouvoir effectuer les opérations suivantes :

- Onglet rapports :
 - Consulter / modifier des rapports qu'il à saisi,
 - Ajouter de nouveaux rapports,
- Onglet medecins :
 - Consulter / modifier les coordonnées d'un médecin,
 - Consulter les rapports de chaque médecin, quelque soit le visiteur l'ayant saisi

Travail à faire

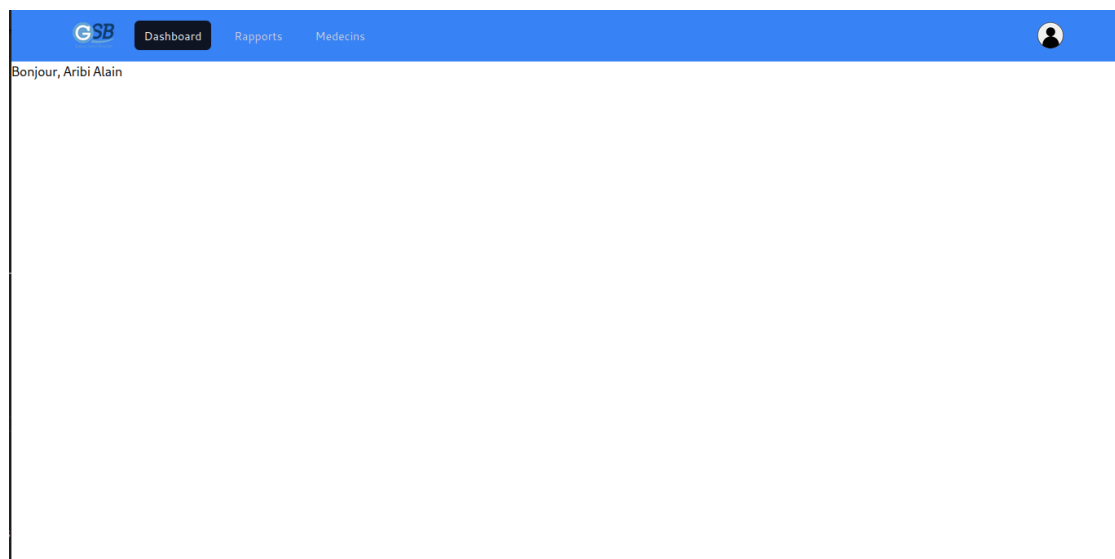
Dans votre répertoire pages/, créez un sous-répertoire accueil/.

Déplacez- y le fichier accueil.jsx (N'oubliez pas d'update les imports dans main.jsx), et ajoutez deux nouveaux composants :

- Medecins.jsx,
- Rapports.jsx

Vous allez d'abord commencer par mettre en place la page d'accueil, puis s'attaquer à la page des médecins.

En fin de première partie, vous aviez mis en place une redirection vers la page d'accueil du visiteur, en affichant ses données bruts renvoyées par l'API. Maintenant nous allons mettre en place une barre de navigation, qui va lui permettre de parcourir les différents pages de notre application :



Exemple de barre de navigation

Page d'accueil et barre de navigation

(Si cela est fait, vous pouvez sauter cette étape !!!)

Dans React, toute page et tout fichier est un composant. Cependant nous pouvons différencier deux éléments :

- le dossier **components/** : il va contenir des composants, c'est à dire des éléments que l'on pourra intégrer dans n'importe quel page,
- le dossier **pages/** : Il va contenir les vues, c'est-à-dire les pages de notre application.

Donc en suivant ce principe, la barre de navigation sera donc un composant que vous pourrez directement intégrer dans n'importe quelle page.

Pour celà, vous allez créer un nouveau composant Navbar :

```

1  import { Disclosure, DisclosureButton, DisclosurePanel, Menu, MenuButton, MenuItem, MenuItems } from
2  import { Bars3Icon, BellIcon, XMarkIcon } from '@heroicons/react/24/outline'
3  import logo from "../assets/logo-gsb.png"
4  import { Link, useNavigate } from 'react-router-dom'
5
6  const navigation = [
7    { name: 'Dashboard', href: '', current: true },
8    { name: 'Rapports', href: 'rapports', current: false },
9    { name: 'Medecins', href: 'medecins', current: false },
10 ]
11
12 function classNames(...classes) {
13   return classes.filter(Boolean).join(' ')
14 }
15
16 export default function Navbar({state}) {
17
18
19   return (
20     <Disclosure as="nav" className="bg-blue-500">
21       <div className="mx-auto max-w-7xl px-2 sm:px-6 lg:px-8">
22         <div className="relative flex h-16 items-center justify-between">
23           <div className="absolute inset-y-0 left-0 flex items-center sm:hidden">
24             {/* Mobile menu button*/}
25             <DisclosureButton className="group relative inline-flex items-center justify-center rounded
26               <span className="absolute -inset-0.5" />
27               <span className="sr-only">Open main menu</span>
28               <Bars3Icon aria-hidden="true" className="block h-6 w-6 group-data-[open]:hidden" />
29               <XMarkIcon aria-hidden="true" className="hidden h-6 w-6 group-data-[open]:block" />
30             </DisclosureButton>
31           </div>
32           <div className="flex flex-1 items-center justify-center sm:items-stretch sm:justify-start">

```

Je ne mets pas volontairement tout le code. En effet, n'ayant pas de charte graphique imposée, libre à vous de coder votre barre de navigation. Vous pouvez vous inspirer de divers exemples sur internet, notamment avec TailwindCSS.

Puis intégrez directement le composant dans la vue Accueil (n'oubliez pas les imports) :

```

return (
  <>
    <Navbar />
    <h1>Bonjour, {nom} {prenom}</h1>
  </>
);

```

Travail à faire

Créez la barre de navigation, et tentez de l'intégrer à la page Accueil.

Routages des composants enfants : La balise Outlet

Sous React, on insiste beaucoup sur le principe de composants parents/enfants. Dans notre application, on sait que pour pouvoir accéder/modifier aux médecins et aux rapports, l'utilisateur doit être connecté, et donc doit forcément passer par la fameuse page d'Accueil. De ce fait, les composants pages Rapports et Médecins deviennent des enfants du composant page Accueil, et ne seront accessibles que lorsque l'utilisateur connecté accèdera à cette page.

Sous React Router, ce principe a un nom : les ***Nested Routes*** (ou routes imbriquées). Cela permet de créer des routes accessibles que via un composant parent, et lui permettre d'afficher dynamiquement son contenu grâce à la balise ***Outlet*** .

Tout d'abord, nous allons retourner dans notre routeur, et créer les nouvelles routes "enfants" de notre composant Accueil :

```
{
  path: 'accueil',
  element: <Accueil />,
  /**
   * Ici, c'est le tableau d'objets children qui
   * va contenir les routes des enfants spécifiques
   * au parent Accueil
   */
  children: [
    {
      path: 'medecins',
      element: <Medecins />
    },
    {
      path: 'rapports',
      element: <Rapports />
    },
  ],
}
```

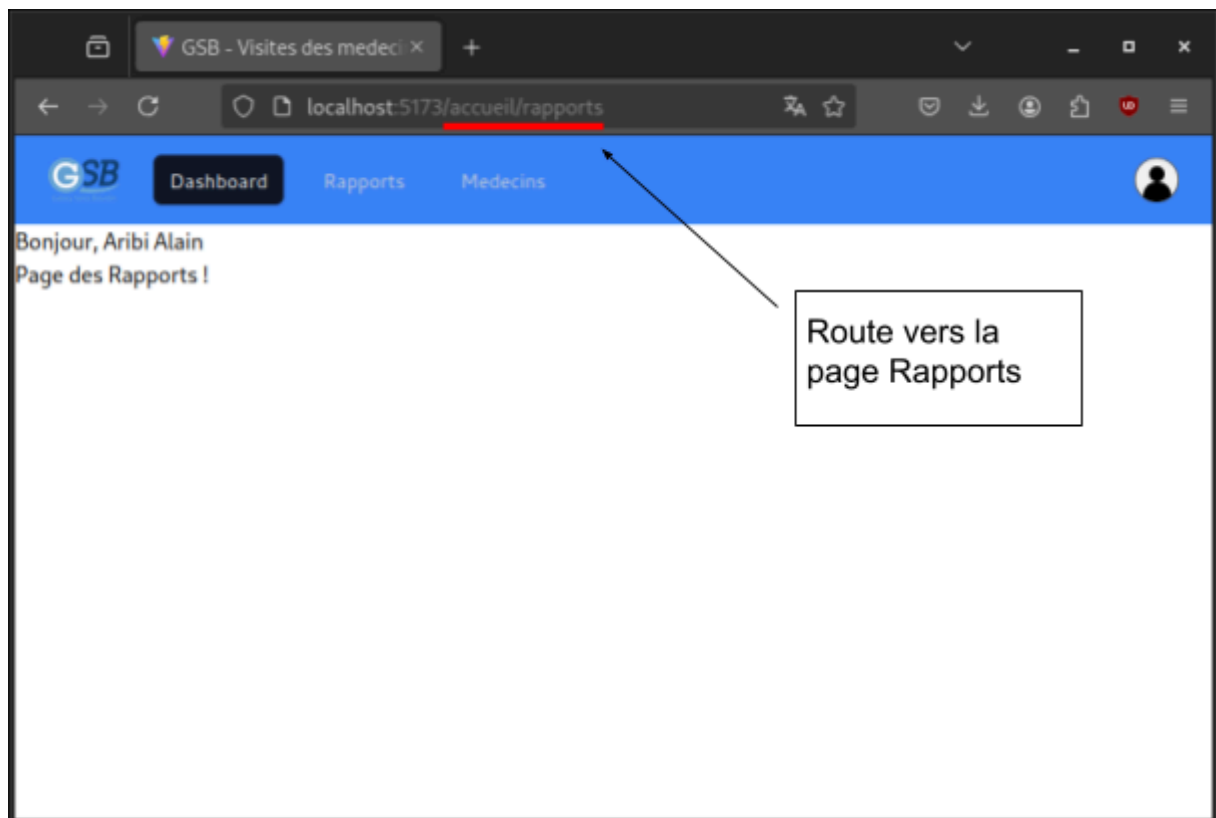
Nous venons donc de créer les routes suivantes :

- /accueil/rapports : Pour charger la page des Rapports,
- /accueil/medecins : Pour charger la page des Medecins

Puis enfin, on mets à jour l'affichage de notre composant Accueil, en y ajoutant la balise Outlet :

```
return (  
  <>  
    <Navbar />  
    <h1>Bonjour, {nom} {prenom}</h1>  
    <Outlet />  
  </>  
);
```

Maintenant, il ne reste plus qu'à appeler les routes correspondantes, React se chargera d'afficher le composant concerné dans la sortie de la balise Outlet :

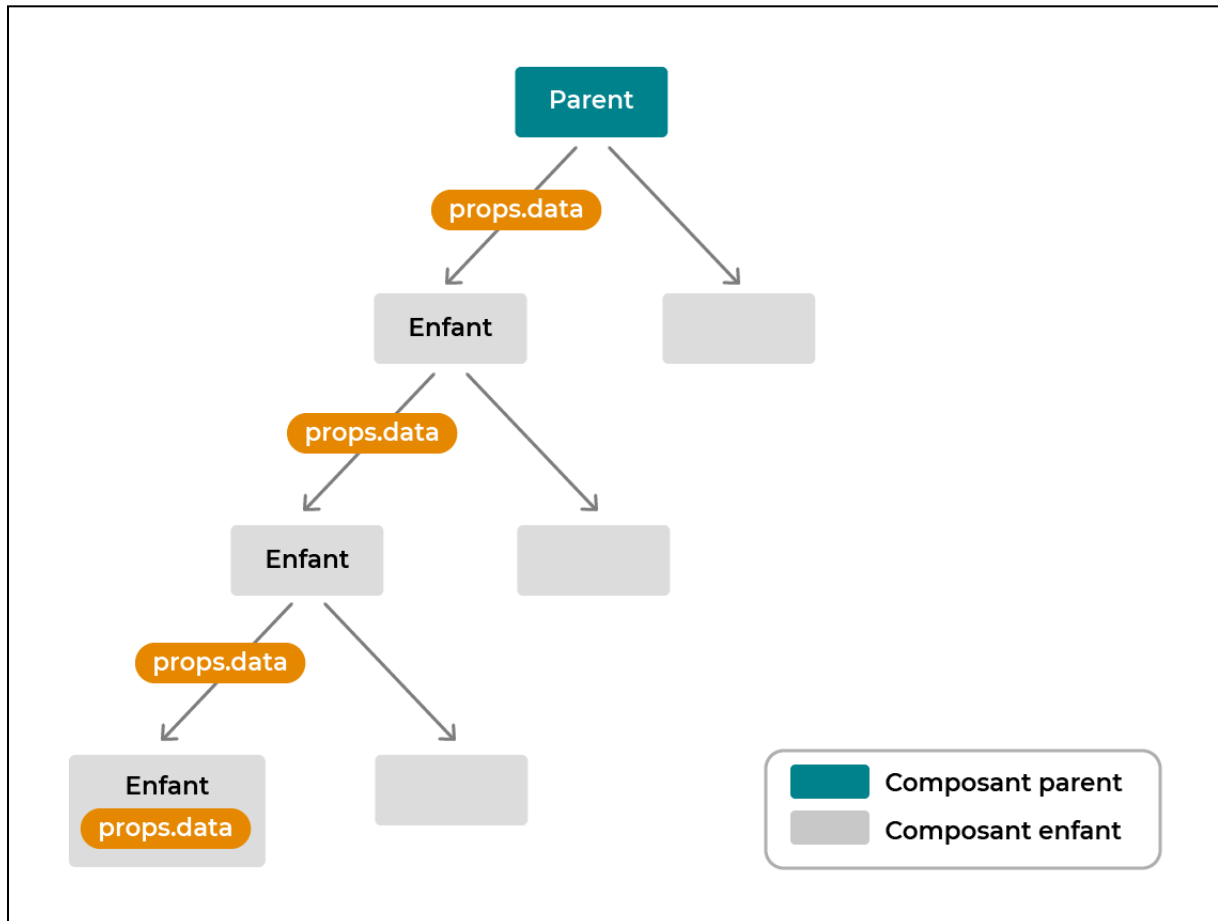


Travail à faire

Créez les routes enfants Medecins et Rapports, et tentez d' accéder à vos pages, d'abord via URL (saisie manuelle), et via votre barre de navigation.

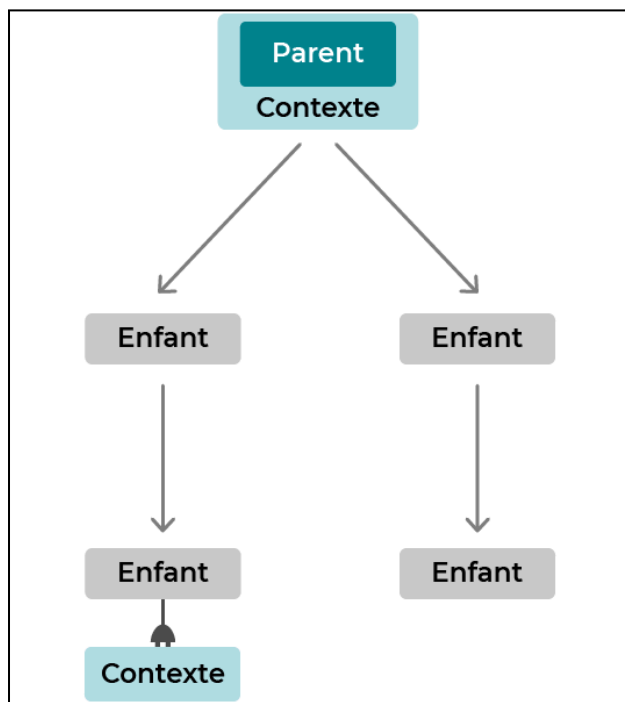
Partage d'états entre les composants : Contexte API

On a déjà vu comment passer de simples props entre les composants parents et enfants, et comment utiliser les props pour faire remonter le state (état). Mais dans une application complexe avec plusieurs composants enfants, vous devez **le faire passer par des dizaines de composants** parents qui n'ont eux-mêmes pas besoin de cette prop :



Cette approche, connue sous le nom de **prop drilling**, peut rendre le code difficile à maintenir. Chaque modification d'un composant peut avoir des conséquences sur d'autres composants plus profonds dans la hiérarchie, augmentant la complexité.

Contexte (Context API) de React est une solution à ce problème. Il permet à un composant d'accéder directement à des données, sans avoir à les transmettre via les props à chaque niveau intermédiaire.



Comment cela fonctionne ? On englobe le composant parent le plus haut dans l'arborescence de composants avec ce qu'on appelle un **Provider**. Tous les composants enfants pourront alors se connecter au **Provider** (littéralement en anglais, le "fournisseur") et ainsi accéder aux props, sans avoir à passer par tous les composants intermédiaires. On dit que les composants enfants sont les **Consumers** (consommateurs).

Cet API est propre à React, il n'y a pas besoin d'imports supplémentaires. De plus, un hook nommé `useContext` a été créé pour simplifier sa mise en place.

Voici quelques sites pour approfondir vos connaissances :

- [WebInsightHaven - useContext](#)
- [OpenClassRooms - Contexte et useContext](#)

Cependant dans notre cas, l'opération est encore plus simplifiée : en effet, React Router propose sa propre solution de contexte déjà en place via :

- La props `context`, qui va servir de provider auquel nous pourrions passer nos états,
- Le hook `useOutletContext`, qui va permettre au composant enfant de consommer les états du provider,

Pour approfondir, vous pouvez vous aider de l'API de React Router : [Outlet Props | React Router](#)

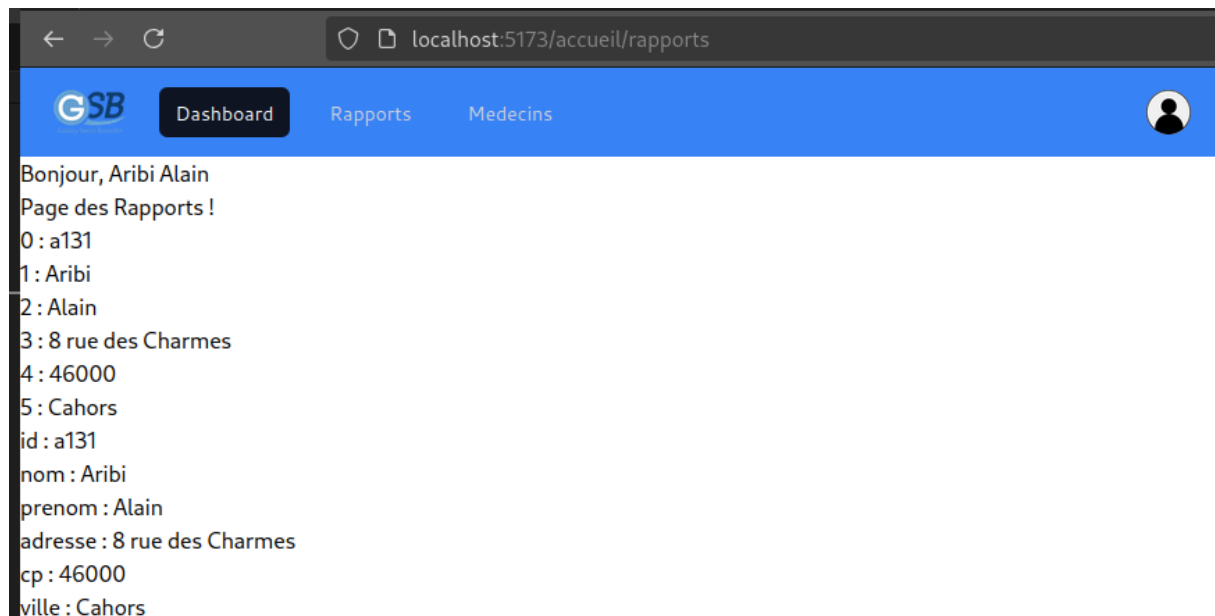
Afin de créer votre provider qui va être relié à notre composant Accueil, on va ajouter un état nommée visiteur, qui va récupérer toutes les informations du visiteur :

```
/**
 * Ici, ma variable d'état visiteur va servir de contexte : elle va permettre
 * de récupérer le visiteur connecté, afin de pouvoir la transmettre sur les
 * enfants
 *
 * La "question" state ? : représente l'opérateur dit "ternaire", qui équivaut
 * à faire un :
 *
 * if(state){
 *   state.user
 * }else{
 *   null
 * }
 */
const [visiteur, setVisiteur] = useState(state ? state.user : null);
```

Travail à faire

En vous aidant de l'API de React Router :

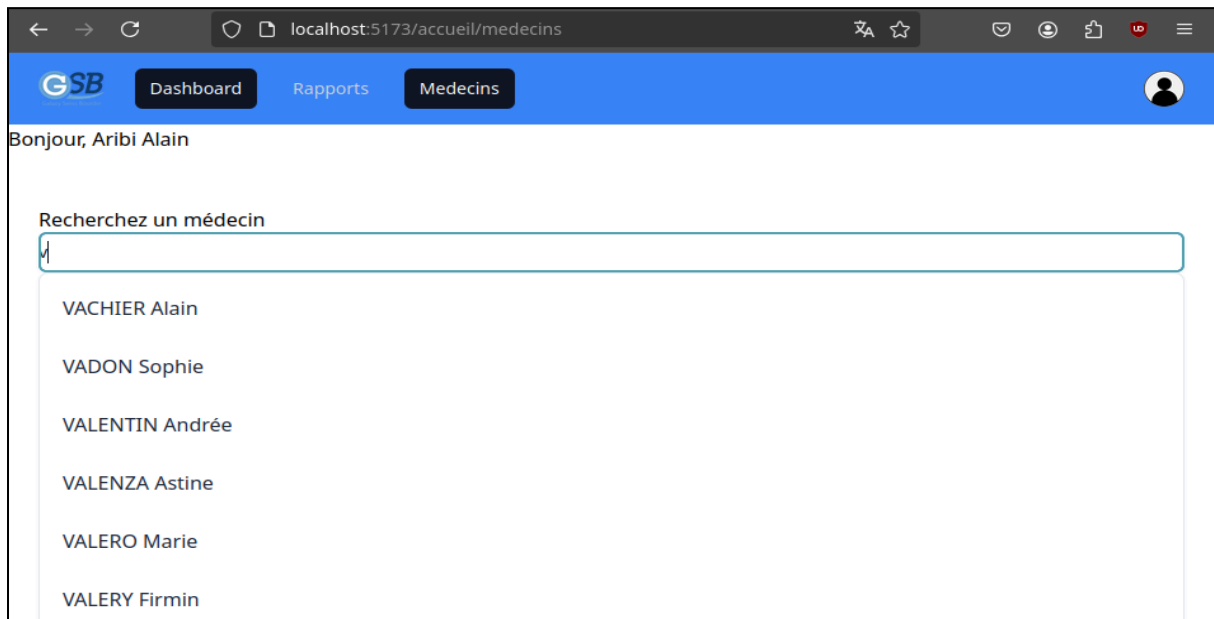
- Mettez en place le provider à transmettre via la balise Outlet,
- Essayez d'afficher dans le composant enfant Rapports, les informations du visiteur, récupérées grâce au provider :



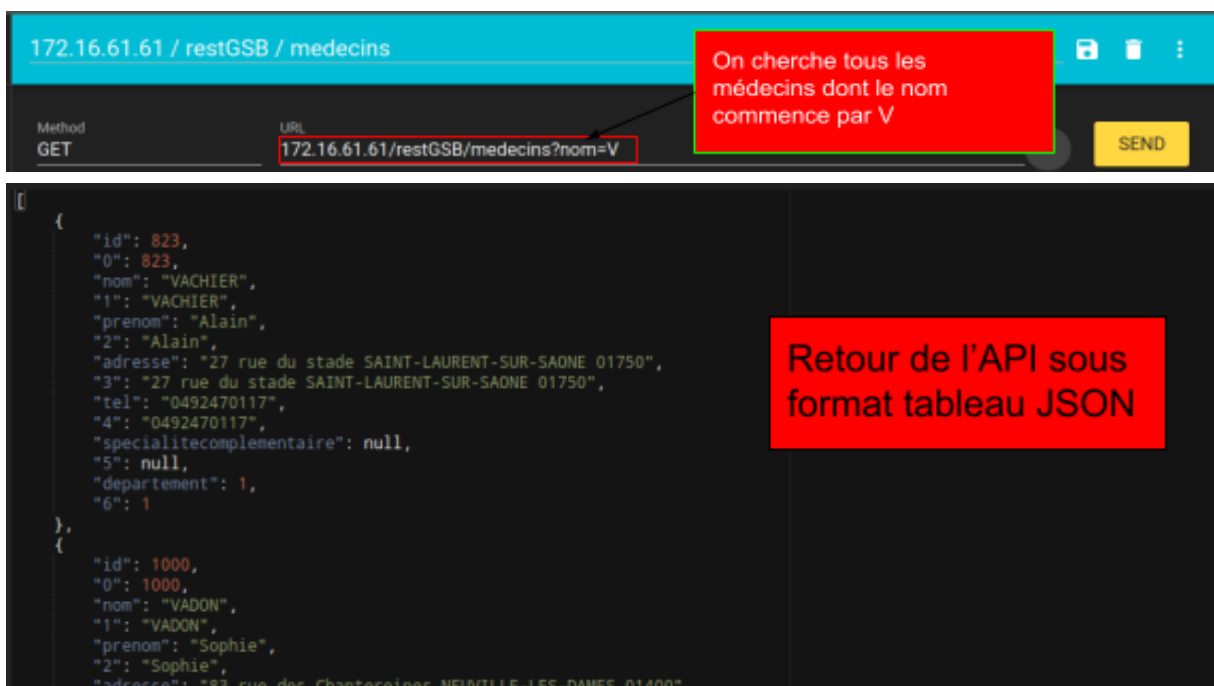
Mise en place de la page des médecins

Barre de recherche de médecin

Afin de permettre au visiteur de choisir le médecin sur lequel il souhaite consulter/modifier les informations, vous allez mettre en place une barre de recherche via le nom de famille du médecin. En effet, à chaque caractère saisi, l'application affichera une liste de médecins dont les noms se rapprochent de la saisie de l'utilisateur :



Pour récupérer ces informations, nous allons à nouveau faire appel à l'API, via la méthode "médecins", avec comme paramètre "nom" :



Voici une ébauche de ce à quoi doit ressembler votre composant Medecins (vous n'êtes pas obligé de le suivre à la lettre, mais vous en inspirer) :

```
export default function Medecins() {
  //const [visiteur, setVisiteur] = useOutletContext();

  const navigate = useNavigate(); // Pour utiliser la navigation du routeur

  const [listeVisible, setListeVisible] = useState(false); // État visibilité de la liste
  const [nomMedecin, setNomMedecin] = useState(''); // État champ de saisie
  const [listeMedecins, setListeMedecins] = useState([]); // Liste qui va contenir les médecins trouvés
  const [medecin, setMedecin] = useState({}); // État qui contient les données du medecin sélectionné
  const [version, setVersion] = useState(0); // État qui permet de forcer le rafraîchissement du contenu

  /**
   * Se déclanche à chaque touche du clavier pressée :
   * Lorsque l'utilisateur saisit un nom :
   * - Si le champ est saisi, on recherche dans l'API et on affecte le retour
   *   dans listeMedecins, puis on affiche la liste à l'écran,
   * - Sinon si le champ est vide, on reset la liste et on enlève l'affichage
   */
  function charger() {
    /** A compléter */
  }

  /**
   * Fonction qui va se déclencher lorsqu'un medecin est selectionné
   * dans la liste.
   * @param {JSON} leMedecin
   */
  function selectMedecin(leMedecin) {
    /** A compléter */
  }

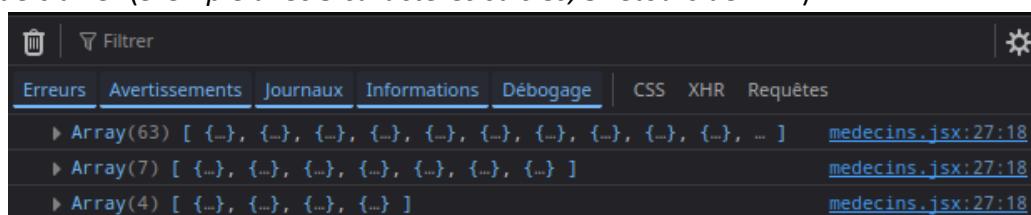
  /**
   * Appel à l'API /medecins, avec comme paramètre 'nom'
   * retour en fonction de la saisie de l'utilisateur
   * @returns response - Variable au format JSON
   */
  async function rechercherRapports() {
    /** A compléter */
  }

  return (
    <>
    {/** A compléter */}
    </>
  )
}
```

Travail à faire

Complétez et testez le composant Medecins.

D'abord, testez avec un log de console si vous avez un retour de l'API à chaque nouvel appui de clavier (exemple avec 3 caractères saisis, 3 retours de l'API) :



Ensuite, vous pouvez vous occuper de l'affichage dans la liste.

Récupération des informations : fonction `selectMedecin`

Une fois la liste affichée, nous allons nous occuper des événements à réaliser lorsqu'un médecin est sélectionné (via la fonction `selectMedecin`). En clair, pour le médecin sélectionné, l'application doit :

- Valoriser la variable **`médecin`** avec les données du medecin sélectionné,
- Cacher la liste des médecins trouvées, et n'afficher dans la barre de recherche juste le nom du médecin sélectionné :



Travail à faire

Complétez la fonction `selectMedecin()` afin de mettre en place les éléments souhaités.

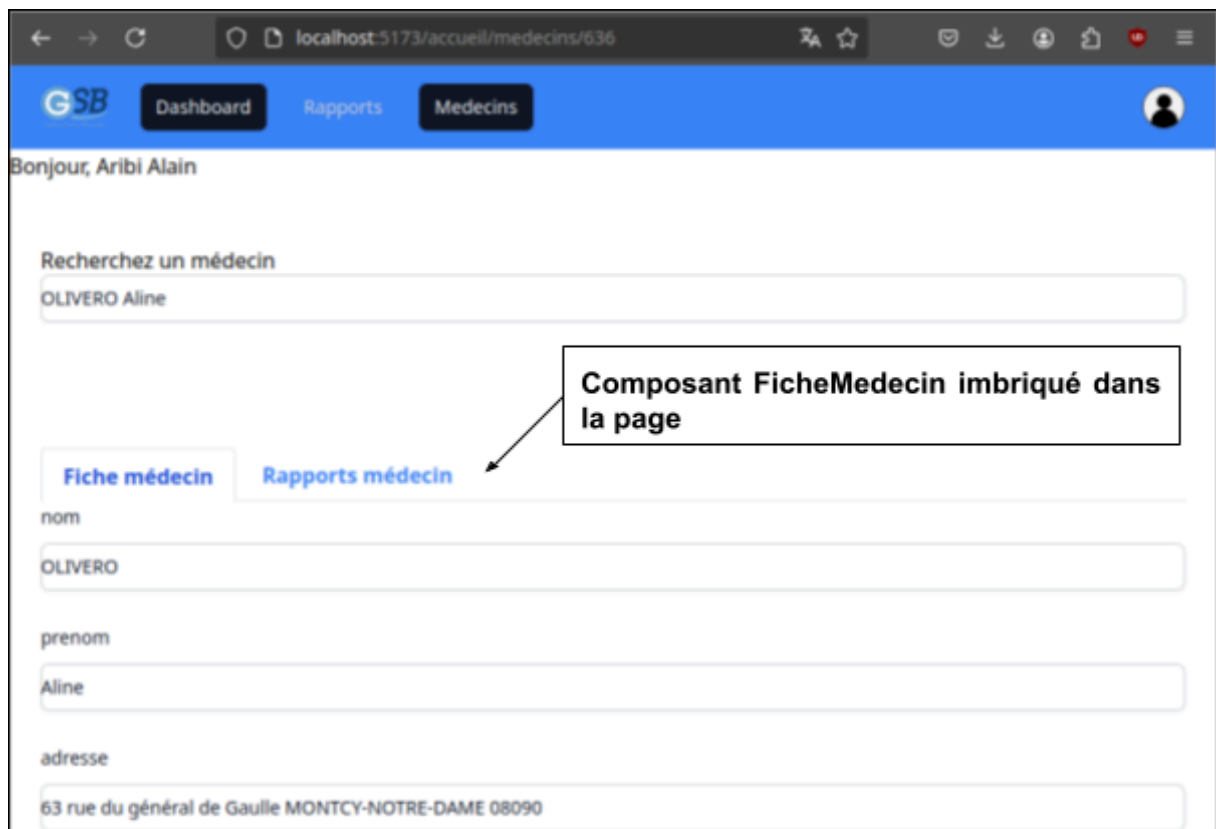
Affichage des informations : Composant `FicheMedecin`

Maintenant que le médecin est sélectionné, il faut mettre en place l'affichage et les fonctionnalités. Pour rappel, le visiteur doit pouvoir :

- Éditer la fiche de renseignements du médecin,
- Pouvoir consulter les rapports préalablement saisis, même s'ils ne proviennent pas de lui.

Nous allons donc créer un composant enfant nommé `ficheMedecin`, qui vient s'imbriquer directement dans notre page `Medecin`, de la même manière que ce qui a été effectué précédemment concernant l'Accueil, avec la fameuse balise **`Outlet`**.

Ce composant sera affiché par la fonction *selectMedecin()*, donc que lorsqu'un médecin sera sélectionné :



localhost:5173/accueil/medecins/636

GSB Dashboard Rapports Medecins

Bonjour, Aribi Alain

Recherchez un médecin

OLIVERO Aline

Fiche médecin Rapports médecin

nom

OLIVERO

prenom

Aline

adresse

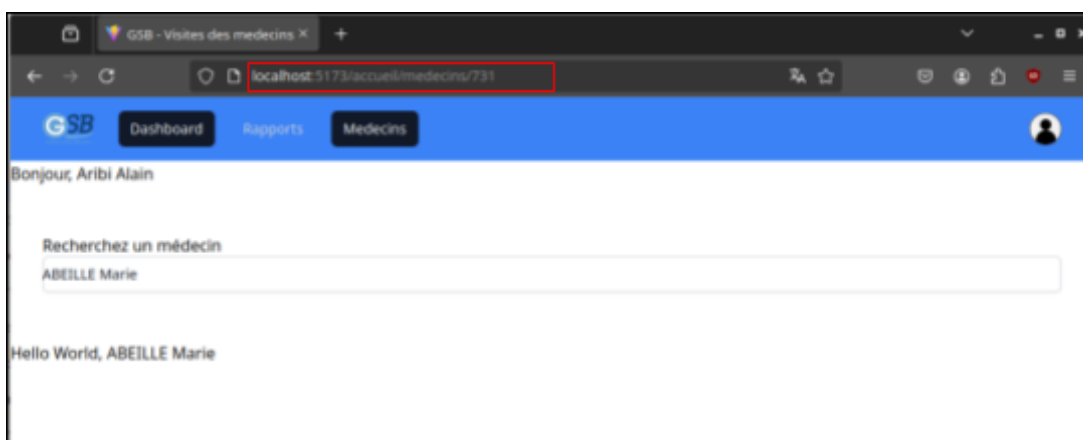
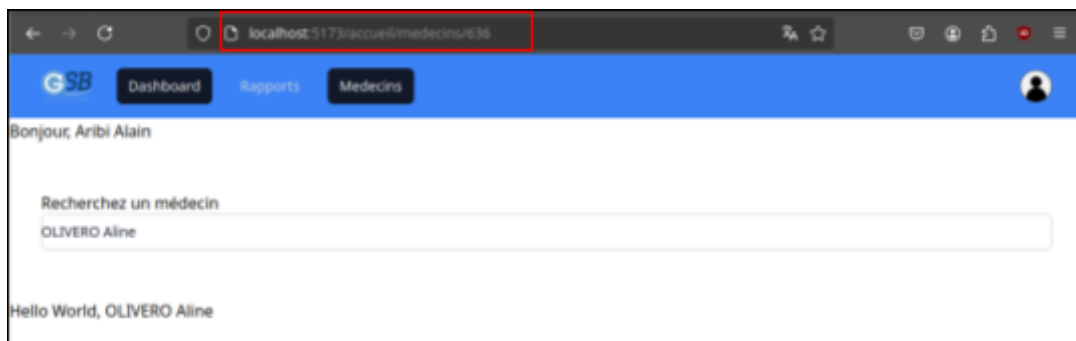
63 rue du général de Gaulle MONTCY-NOTRE-DAME 08090

Composant FicheMedecin imbriqué dans la page

Travail à faire

Créez le fichier `fichemedecin.jsx`, avec le composant `FicheMedecin`. Vous pouvez pour le moment juste lui faire retourner un affichage "Hello World"

Afin de pouvoir mettre en place cette intégration, il faut ajouter une route spécifique dans notre routeur. Observez les deux captures suivantes :



Vous remarquez que l’affichage est le même, mais que la route est différente pour chacun des visiteurs !

Rappelez vous que nous avons utilisé le principe des routes imbriquées notamment pour la partie Accueil, ici nous allons utiliser une variante : les **routes imbriqués dynamiques** (ou **Dynamic Nested Routes**).

Le but est de créer une route de manière dynamique en fonction d’une donnée (sans en connaître sa valeur, et donc directement la route), et dont le contenu sera spécifique à celle-ci. Dans notre cas, nous allons choisir l’ID du médecin sélectionné, qui comportera les données qui seront donc en lien avec ce dernier.

Pour le mettre en place, tout d’abord nous allons créer cette route dans le routeur :

```
{
  path: 'medecins',
  element: <Medecins />,
  /**
   * Création de la route, en fonction de l'ID du médecin sélectionné
   * Attention à bien mettre le ':' dans le path, c'est lui qui indique que la route
   * est dynamique
   */
  children: [
    {
      path: ':id',
      element: <FicheMedecin />
    },
  ],
},
```

Une fois la route créée, nous allons intégrer la balise Outlet pour gérer l'intégration dans notre composant Medecins :

```
{/**
 * Outlet qui permet de charger le composant ficheMedecin,
 * en lui partageant l'état medecin
 */}
<Outlet context={[medecin, setMedecin]} key={version} />
```

Un élément doit attirer votre attention : c'est la props **key**.

Nous n'avons pas directement besoin de cette valeur, mais elle va nous être utile pour contrer le problème des re-renders de React : en effet, il se peut que votre fenêtre ne se rafraîchisse pas en temps voulu, donc pour forcer ce rafraîchissement, on va mettre en place des versions de la page.

Rappelez vous que je vous ai demandé de créer une variable d'état **version**, qui était initialement à 0 par défaut. A chaque fois que nous allons sélectionner un médecin, nous allons incrémenter la variable version, donc en quelque sorte avoir une "nouvelle version" de la page, ce qui va forcer React à re-render votre composant.

Donc pour charger notre composant, nous allons ajouter les lignes suivantes à la fonction **selectMedecin()**:

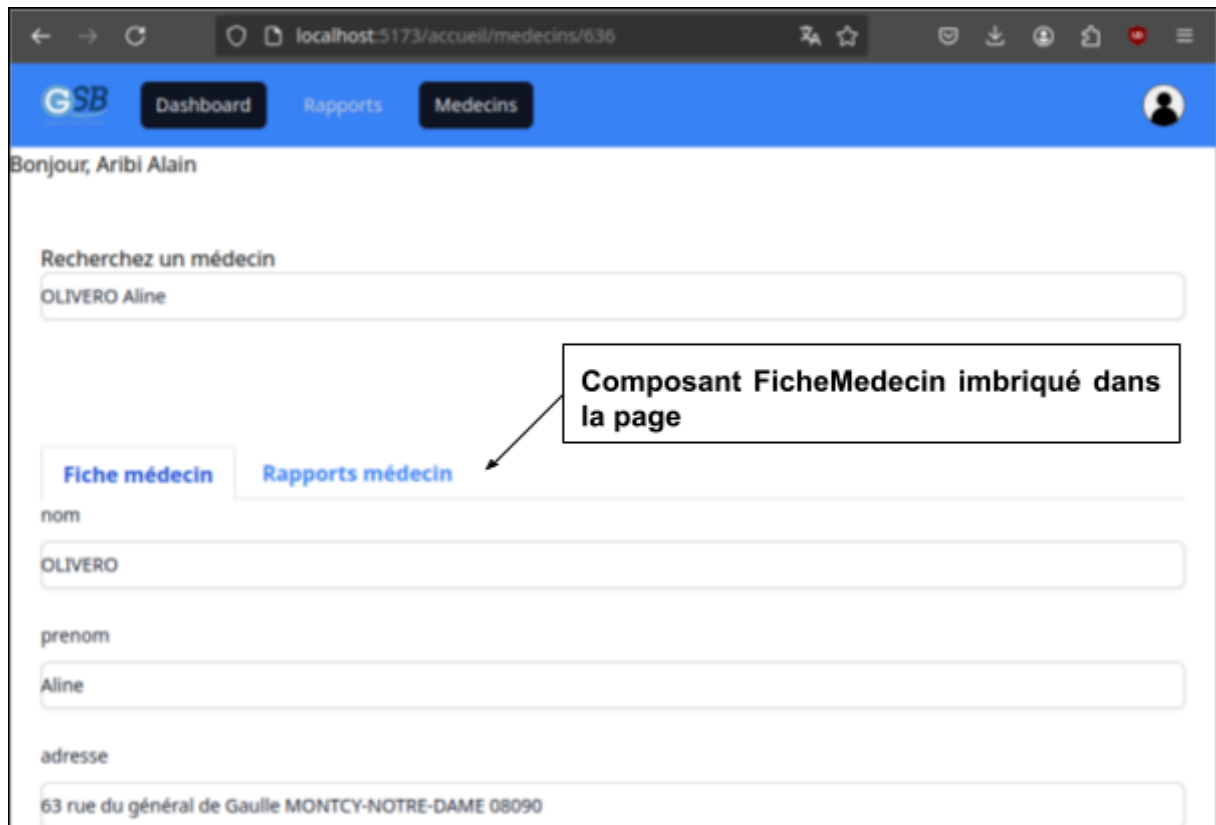
```
// On incrémente la version, et on fait appel au composant
// ficheMedecin de manière dynamique (avec ID Médecin)
setVersion(version+1)
navigate(' '+leMedecin.id)
```

Travail à faire

- Créez la route, et complétez la fonction *selectMedecin()*.
- Tester le bon fonctionnement, avec l'essai suivant :
 - Sélectionner deux médecins différents, vérifiez que l'URL change, et que l'affichage reste le même. Vérifiez aussi que le composant récupère bien les informations du médecin via le contexte (cf capture ci-dessus)

FicheMedecin : Organisation du composant

Maintenant que nous avons réussi à afficher la fiche du médecin, avec le passage des informations le concernant dans le contexte, nous allons organiser notre composant afin qu'il soit divisé en deux parties, comme vu dans cette capture :



localhost:5173/accueil/medecins/636

GSB Dashboard Rapports Medecins

Bonjour, Aribi Alain

Recherchez un médecin

OLIVERO Aline

Fiche médecin Rapports médecin

nom

OLIVERO

prenom

Aline

adresse

63 rue du général de Gaulle MONTCY-NOTRE-DAME 08090

Composant FicheMedecin imbriqué dans la page

Les deux parties seront indépendantes, et seront des composants imbriqués à FicheMedecin, et selon le choix de l'utilisateur, on affichera :

- Soit la fiche du médecin (Composant **<Fiche />**) : il s'agira d'un formulaire pré-rempli avec les informations du médecin, que le visiteur pourra éditer, et valider les modifications afin qu'ils soient enregistrés dans la base de données
- Soit les rapports du médecin (Composant **<Rapports />**) : il s'agira d'un tableau qui permettra au visiteur de consulter les rapports saisis concernant ce médecin, quel que soit celui qu'il a saisi.

Dans un premier temps, nous allons mettre à jour le fichier FicheMedecin, afin d'ajouter les composants imbriqués.

Au début de notre fichier, nous allons créer les deux composants imbriqués, Fiche et Rapports :

```
/**
 * Composant Fiche Medecin, qui va prendre comme paramètre le hook Medecin
 * et gérer l'affichage et les actions du formulaire (mise à jour des coordonnées)
 * @param {JSON} leMedecin
 * @returns Affichage du formulaire pré-rempli
 */
function Fiche({ leMedecin }) {

  return (
    <h1>Fiche formulaire </h1>
  )
}

/**
 * Composant Rapports Medecin, qui prend en props idMedecin,
 * et va récupérer les rapports concernant le médecin depuis l'API,
 * et les afficher sous forme de tableau
 * @param {string} idMedecin
 * @returns Tableau avec la liste des rapports du medecin en cours
 */
function Rapports({idMedecin}) {

  return (
    <h1>Fiche Rapports</h1>
  )
}
```

Puis nous allons mettre à jour la fonction FicheMedecin, afin d'afficher le sous-menu, puis gérer l'affichage du composant sélectionné par le visiteur.

Pour garder en mémoire le choix de l'utilisateur, nous allons aussi créer un état nommé **affichage**, qui aura comme valeur "fiche" par défaut.

```
export default function FicheMedecin() {
  const [medecin, setMedecin] = useOutletContext() // Récupérer le state Medecin
  const [affichage, setAffichage] = useState('fiche') // Choix de l'affichage, fiche par défaut

  return (
    <>
      {/** Affichage de la sous-barre de navigation */}
      <div className="flex min-h-full flex-col justify-center px-6 py-12 lg:px-8">
        <ul className="flex border-b">
          <li className="mb-px mr-1">
            <a className="bg-white inline-block border-l border-t border-r rounded-t
            py-2 px-4 text-blue-700 font-semibold"
            href="#" onClick={() => setAffichage('fiche')}>
              Fiche médecin
            </a>
          </li>
          <li className="mr-1">
            <a className="bg-white inline-block py-2 px-4 text-blue-500 hover:text-blue-800
            font-semibold"
            href="#" onClick={() => setAffichage('rapports')}>
              Rapports médecin
            </a>
          </li>
        </ul>
      <br />
    </>
  )
}
```



```

{
  /**
   * Test ternaire (équivalent du if) : on se demande si la valeur de affichage est
   * égale à fiche : Si oui, on affiche le composant fiche, sinon on affiche le
   * composant Rapports
   */
  affichage == 'fiche' ?
    <Fiche leMedecin={medecin, setMedecin]} />
    :
    <Rapports idMedecin={medecin.id} />
}
</div>
</>
);
}

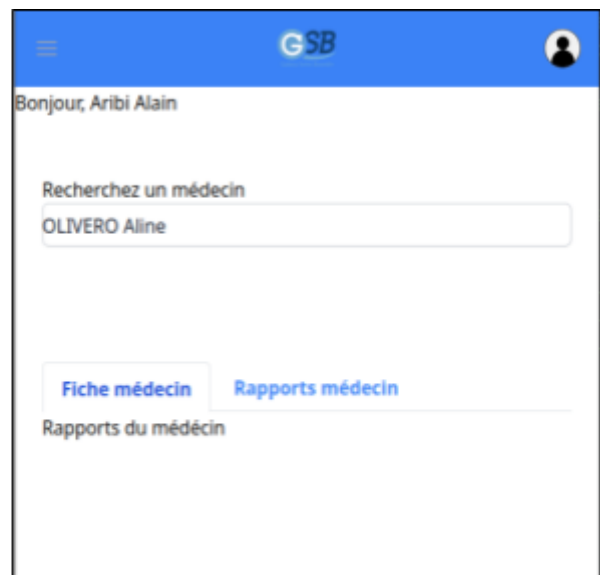
```

Travail à faire

- Ajoutez les composants, et mettez à jour FicheMedecin.
- Tester le bon fonctionnement de la navigation :



Clic sur "Fiche Médecin"



Clic sur "Rapports Médecin"

FicheMedecin : Édition de la fiche du médecin

Comme indiqué précédemment, ce composant devra permettre au visiteur de modifier les informations du médecin sélectionné. Pour cela, ses informations de base (récupérées grâce au contexte) permettront de pré-remplir le formulaire.

Lors que le visiteur le validera :

- Déclenchement d'un appel à l'API via l'URL **/majMedecin** et la méthode PUT, avec comme paramètre les données du formulaire (consultez l'API de Axios),
- Si succès de la requête : afficher une bannière de succès,
- Si échec : afficher une bannière d'erreur

Voici une ébauche possible pour ce composant Fiche :

```
function Fiche({ leMedecin }) {  
  
  const [medecin, setMedecin] = leMedecin;  
  const [updateMedecinSuccess, setUpdateMedecinSuccess] = useState() // Check si la MAJ à reussi ou non  
  
  /**  
   * Fonction qui se déclanche lors de la soumission du formulaire,  
   * fait appel à l'API avec les données saisies pour MAJ le médecin  
   * dans la base de données  
   * @param {*} e  
   */  
  function updatemedecin(e) {  
    /**  
     * A compléter  
     */  
  }  
  
  /**  
   * Appel à l'API pour mettre à jour le medecin dans la base  
   * de données, via la méthode PUT  
   * @param {JSON} params  
   * @returns Promesse Axios  
   */  
  async function sendUpdateMedecin(params) {  
    /**  
     * A compléter  
     */  
  }  
  
  return  
    /**  
     * A compléter  
     */  
}
```

Travail à faire

- Mettez à jour le composant Fiche, avec le formulaire et ses actions.
- Vérifiez que la mise à jour fonctionne : affichage de la bannière, et surtout au niveau de la base de données :

Fiche médecin	Rapports médecin
nom	
OLIVERO	
prenom	
Aline	
adresse	
63 rue du général de Gaulle MONTCY-NOTRE-DAME 08090	
tel	
0330296222	
specialitecomplementaire	
Je fais un test	
departement	
8	
Mettre à jour	
MAJ effectuée	

Affichage en cas de succès

✓ Affichage des lignes 0 - 0 (total de 1, traitement en 0.0005 seconde(s)).

```
SELECT * FROM 'medecin' WHERE 'nom' LIKE 'OLIVERO'
```

☐ Profilage [Éditer en ligne] [Éditer] [Expliquer SQL] [Créer le code source PHP] [Actualiser]

☐ Tout afficher | Nombre de lignes : 25 | Filtrer les lignes: Chercher dans cette table

Options supplémentaires

	id	nom	prenom	adresse	tel	specialitecomplementaire	departement
<input type="checkbox"/> Éditer Copier Supprimer	636	OLIVERO	Aline	63 rue du général de Gaulle MONTCY-NOTRE-DAME 0809...	0330296222	Je fais un test	8

⬅️ ➡️ ☐ Tout cocher Avec la sélection : Éditer Copier Supprimer Exporter

BDD : Modification de la spécialité prise en compte

FicheMedecin : Affichage de la liste des rapports

Nous arrivons à la dernière étape de la partie 2 : l’affichage des rapports du médecin. De ce fait, nous devons donc dès le départ faire appel à notre API afin de récupérer les rapports du médecin sélectionné, en fonction de son ID.

Afin de nous simplifier la tâche, et synchroniser l’appel au composant avec l’API, nous allons utiliser le hook `useEffect` : ce hook va nous permettre de faire appel à notre API, à chaque fois que id Médecin va changer.

Voici une ébauche possible pour ce composant Rapports :

```
function Rapports({idMedecin}) {

  const [rapportsMedecin, setRapportsMedecin] = useState([]); // Stockage des rapports du Medecin

  /**
   * Utilisation du hook useEffect : Appel à l'API via la méthode GET
   * dès le chargement/rafraîchissement du composant.
   * Cette synchronisation va dépendre de idMedecin
   * URL API : '/rapports/'+idMedecin
   */
  useEffect(() => {
    async function rapports() {
      /** A compléter */
    }
    rapports();
  },[idMedecin])

  return
  /**
   * A compléter
   */
}
```

Travail à faire

- Compléter l’appel à l’API dans le hook, et vérifiez dans un premier temps si cela fonctionne via un affichage console :

```
▼ Array [ {...} ] fichemedecin.jsx:99:22
  ▶ 0: Object { 0: "Demande du médecin", 1: "2016-10-02", 2: "A revoir assez rapidement", ... }
    length: 1
  ▶ <prototype>: Array []
```

- Mettre en place l’affichage sous forme d’un tableau :

Recherchez un médecin			
OLIVERO Aline			
Fiche médecin	Rapports médecin		
DATE	MOTIF	BILAN	VISITEUR
02/10/2016	Demande du médecin	A revoir assez rapidement	Desmarquest Nathalie

Attention : Pensez à convertir la date au format “JJ-MM-AAAA” lors de l’affichage.