Rapport de stage BTS SIO 1ère année SLAM

DU 19 MAI 2025 AU 29 JUIN 2025

Benjamin DASSONVILLE

Sommaire :

Remerciements :

Avant de commencer ce rapport de stage, je tiens à remercier **Lyes**, mon tuteur de stage, pour son accueil chaleureux, sa disponibilité et ses conseils précieux tout au long de cette expérience professionnelle. Son accompagnement m'a permis de progresser, de gagner en autonomie et de mieux comprendre les exigences du monde du travail. Merci pour sa bienveillance et pour m'avoir intégré pleinement dans l'équipe.

Je souhaite également remercier **Omar Blanco** pour son aide et son soutien dans l'obtention de ce stage. Sans lui, cette opportunité n'aurait peut-être pas été possible.

Un grand merci également à toute l'équipe avec laquelle j'ai travaillé, composée de **Ali**, **Alex**, **Julien**, et **Oda**, pour leur bonne humeur. Leur accueil et leur collaboration ont grandement contribué à rendre ce stage enrichissant et agréable.

Enfin, je remercie également toutes les personnes avec qui j'ai eu l'occasion d'échanger ou de collaborer au cours de mon stage. Chacun de ces échanges m'a permis d'enrichir mes connaissances, de mieux comprendre l'environnement professionnel, et d'avancer un peu plus chaque jour.

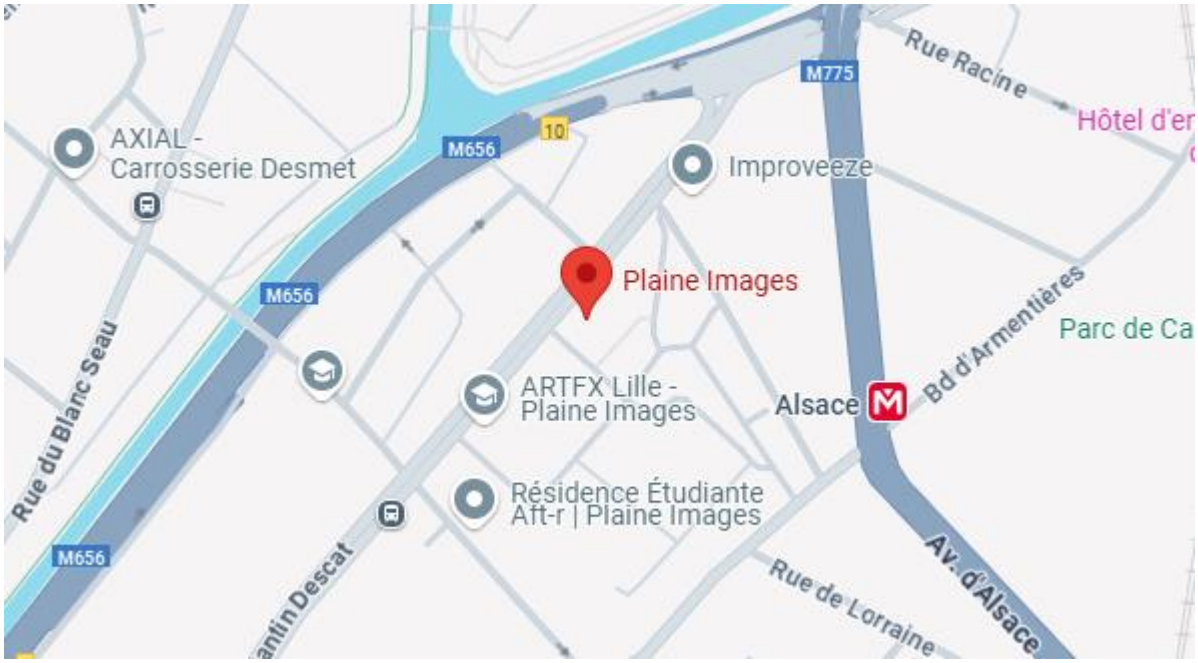
Introduction :

Dans le cadre du **B**revet de **T**echnicien **S**upérieur en **S**ervice **I**nformatique aux **O**rganisations option **S**olutions **L**ogicielles et **A**pplications **M**étiers, j'ai effectué un stage de 6 semaines au sein d'une équipe d'ingénieur de l'université de Lille a la plaine image.

Le thème principal de ce stage était, le développement web et gestion de bases de données dans un contexte thérapeutique.

Organisme d’accueil :

Localisation géographique :



99 A Bd Constantin Descat, 59200 Tourcoing

Entreprise :

Depuis sa création à Roubaix, La Plaine Images, implantée dans l’Imaginarium, s’est imposée comme un écosystème dédié aux industries créatives et numériques. Ce lieu unique rassemble entreprises, startups, étudiants, chercheurs et artistes autour de projets innovants, culturels et technologiques.

Cette dynamique est rendue possible grâce à l'engagement constant des équipes, des résidents, des partenaires et des intervenants extérieurs, qui font vivre quotidiennement cet espace d'innovation. Ensemble, ils participent au développement de projets en jeu vidéo, animation, design, audiovisuel et bien plus encore.

La Plaine Images est un véritable lieu d'échanges, de collaborations et de découvertes, au service de celles et ceux qui souhaitent apprendre, créer, entreprendre ou simplement explorer de nouveaux horizons numériques et créatifs.

L'écosystème Plaine Images bénéficie de nombreux soutiens.
Il s'appuie sur des partenariats solides avec des acteurs publics et privés, tels que la Région Hauts-de-France, la MEL (Métropole Européenne de Lille), ainsi que plusieurs incubateurs, écoles, laboratoires de recherche et entreprises du secteur.

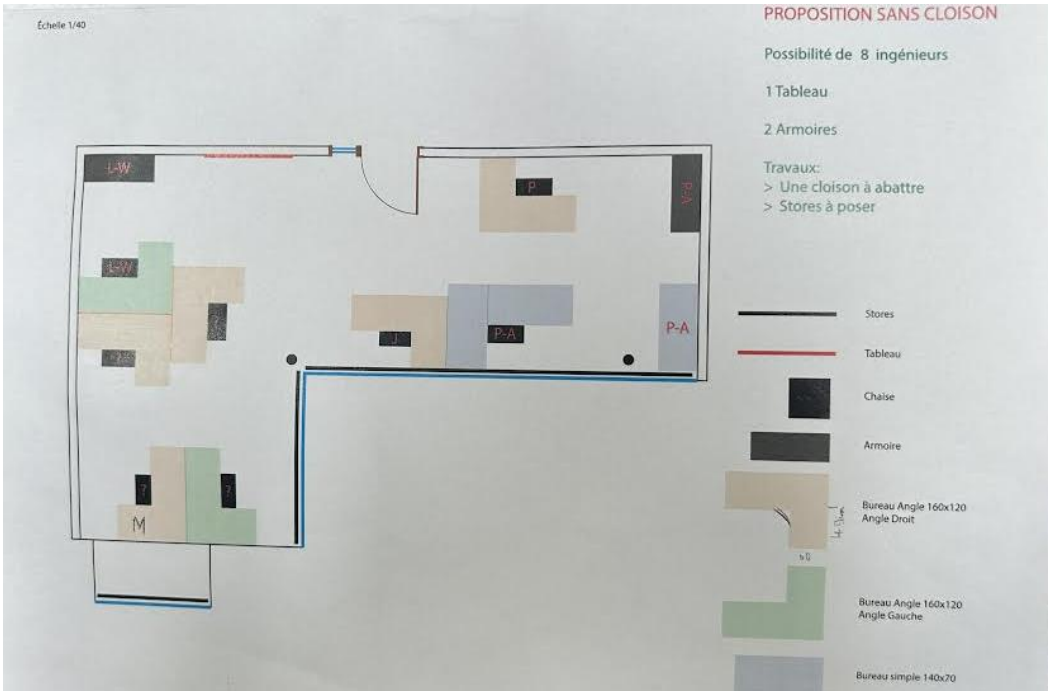
Ces collaborations permettent à La Plaine Images de soutenir l'innovation, de favoriser l'entrepreneuriat et de créer des passerelles entre le monde académique, culturel et industriel, au cœur du territoire roubaisien.

Equipe :

Durant mon stage à La Plaine Images, j'ai travaillé exclusivement avec Lyes, mon tuteur. Cette collaboration rapprochée m'a permis de bénéficier d'un encadrement personnalisé, d'apprendre directement de son expertise et de mener à bien les missions qui m'ont été confiées.

Bien que j'aie pu échanger avec d'autres membres et équipes sur place, c'est avec Lyes que j'ai construit l'essentiel de mon expérience professionnelle durant ce stage.

BUREAUX DES INGENIEURS :



NOTRE EQUIPE :

Lyes - ingénieurs

Julien - ingénieurs

Alex - ingénieurs

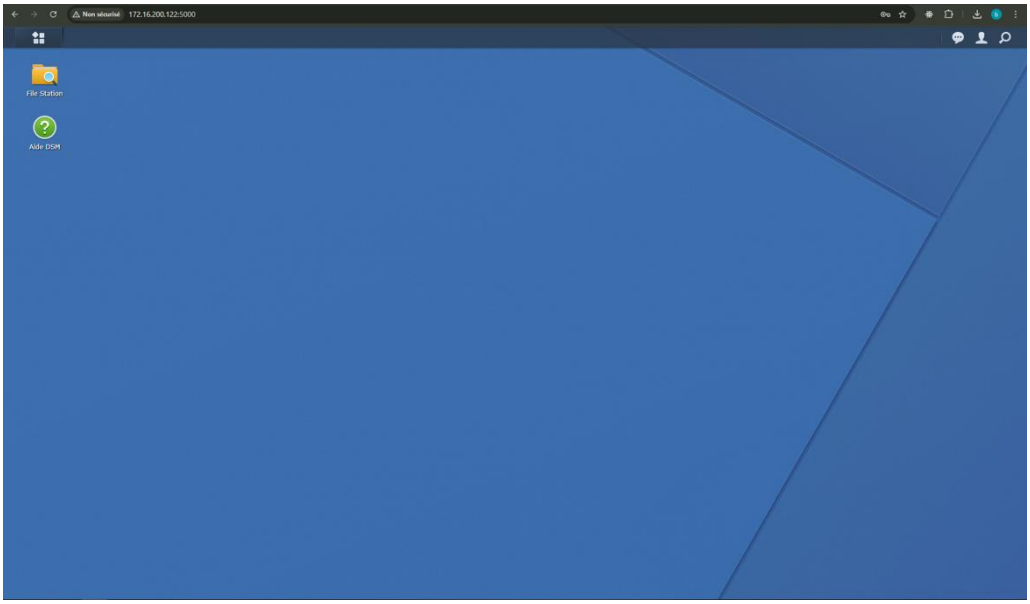
Oda - ingénieurs

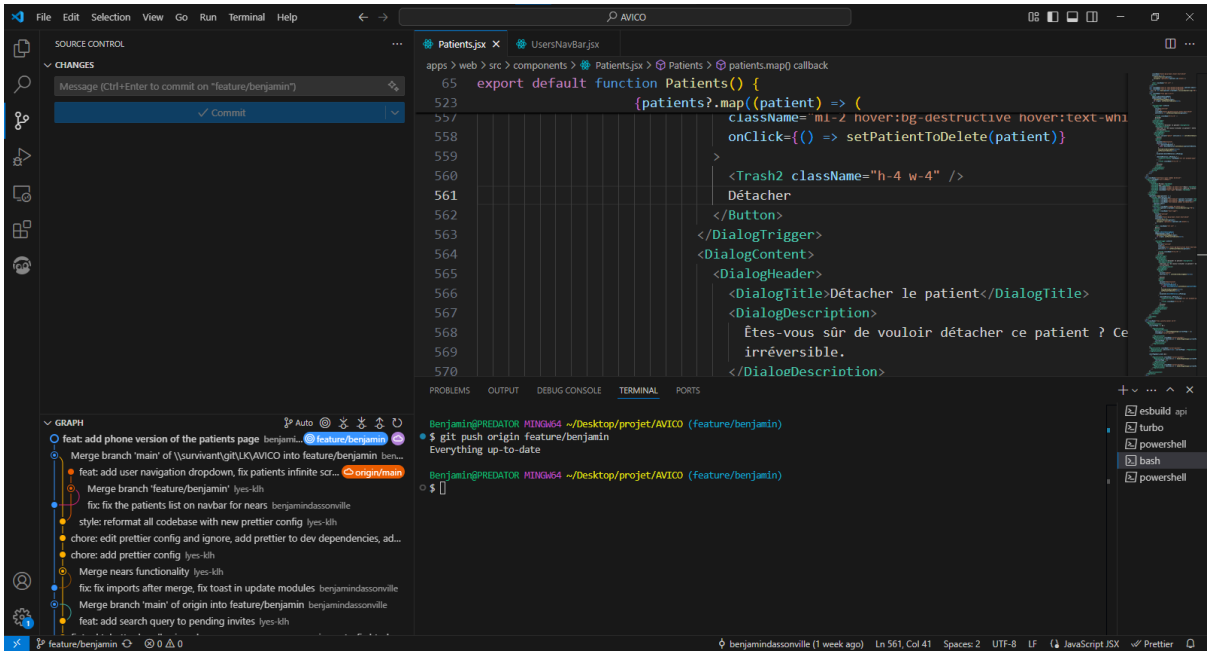
Ali - ingénieurs

Contexte Technologique :

Dans le cadre de la découverte du service, j'ai pu découvrir les différentes technologies utilisées par le service ou j'ai effectué mon stage, les voici :

- GitHub : pour les repos
- VS Code : pour la gestion du code
- NAS : pour stocker les fichiers





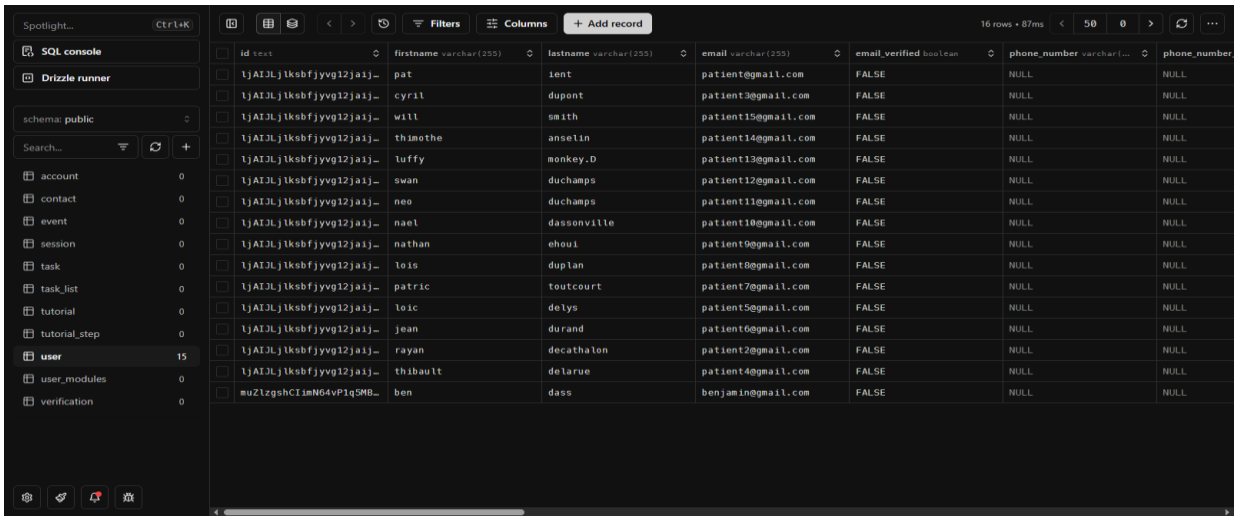
Mission :

Mise en place :

La première étape a été l'installation des logiciels nécessaires, suivie d'une présentation détaillée du projet et de son fonctionnement. Ce projet est structuré en trois parties distinctes : le backend, l'API, et le front-end.

Il s'agit d'un site web permettant la gestion des utilisateurs par d'autres utilisateurs. Ma première tâche consiste à ajouter une fonctionnalité de pagination sur la liste des utilisateurs. Pour cela, j'utilise ReactJS avec TypeScript, un langage que je découvre et apprends pendant ce stage. L'implémentation de la pagination s'appuie sur la bibliothèque shadcn/ui.

Le backend repose sur une base de données PostgreSQL hébergée dans un conteneur Docker, tournant sous WSL2, une couche qui simule Linux sur Windows. Pour gérer les données, nous utilisons Drizzle Studio comme système de gestion de base de données relationnelle.



Pagination

Pour commencer à implémenter la pagination, j'ai d'abord installé les dépendances nécessaires directement dans le dossier du front-end en lançant la commande suivante :

npx shadcn@latest add pagination

Pour pouvoir utiliser la pagination, j'ai commencé par importer le composant dans la page concernée avec la ligne suivante (exemple) :

```
import {
  getPatients,
  getPatientsNear,
  addPatient,
  addPatientNear,
  deletePatient,
  deletePatientNear,
  getPendingInvites,
  getPendingInvitesNear,
  cancelInvite,
  cancelInviteNear,
} from "@/services/patientsService";
```

Cet import permet d'intégrer facilement le composant Pagination dans le JSX de la page, et de l'utiliser pour gérer la navigation entre les différentes pages de la liste des utilisateurs.

Puis j'ai utilisé des hooks `useSearchParams` et `useState` pour synchroniser la page courante avec l'URL et gérer dynamiquement le changement de page.

```
const [searchParams, setSearchParams] = useSearchParams();
const [currentPage, setCurrentPage] = useState(() => {
  return parseInt(searchParams.get("page") || "1");
});
const [pageSize, setPageSize] = useState(() => {
  return parseInt(searchParams.get("size") || "5");
});
```

Cela permet de garder la pagination synchronisée avec l'URL (ex: `/patients?page=2&size=10`) pour une meilleure UX et la possibilité de partager un lien direct vers une page donnée.

Cette fonction permet de changer la page actuelle tout en mettant à jour l'URL :

```
const handlePageChange = (newPage) => {
  setCurrentPage(newPage);
  setSearchParams({
    page: newPage.toString(),
    size: pageSize.toString(),
  });
};
```

Voici comment les boutons de pagination sont affichés dans le JSX :

```tsx
<Pagination>
  <PaginationContent>
    {currentPage > 1 && (
      <PaginationItem>
        <PaginationPrevious onClick={() => handlePageChange(currentPage - 1)} />
      </PaginationItem>
    )}
    <PaginationItem>
      <PaginationLink isActive={true}>{currentPage}</PaginationLink>
    </PaginationItem>
    {nextPageQuery.data && (
      <PaginationItem>
        <PaginationNext onClick={() => handlePageChange(currentPage + 1)} />
      </PaginationItem>
    )}
  </PaginationContent>
</Pagination>
```

Le composant affiche :

- Le bouton « Précédent » si `currentPage > 1`
- Le numéro de la page actuelle en surbrillance
- Le bouton « Suivant » si une page suivante existe (`nextPageQuery.data`)

Pour éviter que l'utilisateur accède à une page vide, une requête vérifie s'il existe une page suivante :

```tsx
const nextPageQuery = useQuery({
  queryKey: ["check-next-page", currentPage + 1, pageSize],
  queryFn: async () => {
    const next = await getPatients({
      searchQuery,
      pageParam: currentPage + 1,
      limit: pageSize,
      therapistId: currentUserQuery.data?.id,
    });
    return next.length > 0;
  },
  enabled: !!currentUserQuery.data?.id,
});
```

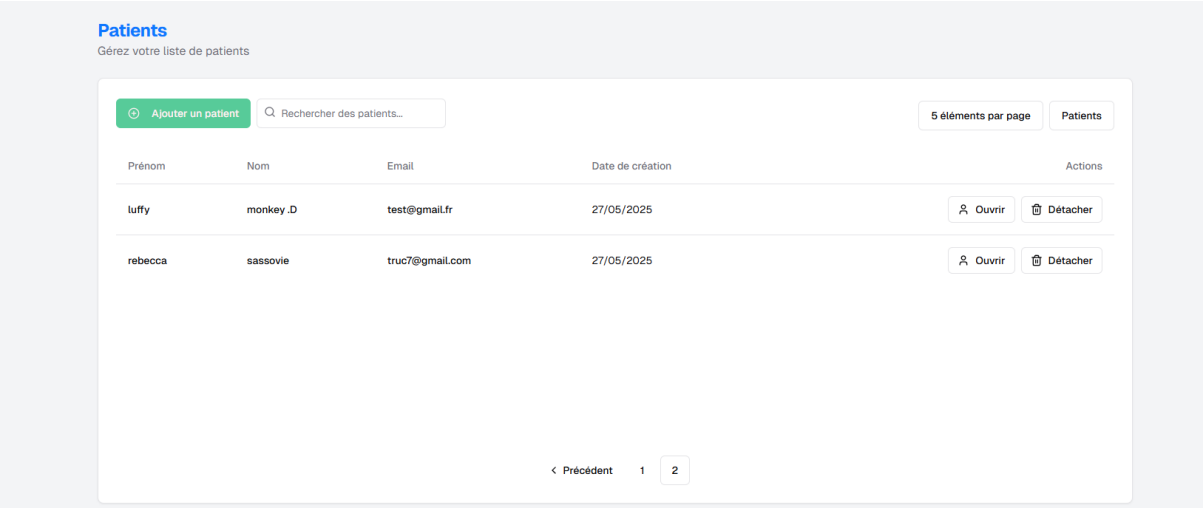
Les données sont récupérées dynamiquement en fonction de la page et de la taille demandée :

```tsx
const { data: patients } = useQuery({
  queryKey: ["patients", currentPage, pageSize],
  queryFn: async () => {
    return await getPatients({
      searchQuery,
      pageParam: currentPage,
      limit: pageSize,
      therapistId: currentUserQuery.data?.id,
    });
  },
  enabled: !!currentUserQuery.data?.id,
});
```

L’utilisateur peut sélectionner combien d’éléments afficher par page grâce à un `DropdownMenuRadioGroup` :

```
<DropdownMenuRadioGroup
  value={pageSize.toString()}
  onValueChange={(value) => {
    const newSize = Number(value);
    setPageSize(newSize);
    setCurrentPage(1);
    setSearchParams({
      page: "1",
      size: newSize.toString(),
    });
  }}
>
  {/* Options ici */}
</DropdownMenuRadioGroup>;
```

Cela améliore la personnalisation de la pagination par l’utilisateur.



Cette pagination est **entièrement pilotée par l’état React combiné à l’URL**, ce qui permet une navigation fluide, un partage facile de liens directs, et une interface optimisée pour les listes volumineuses. Elle est utilisée dans plusieurs vues du projet (patients, invitations...) avec un comportement homogène et performant.

Recherche

Pour permettre à l'utilisateur de retrouver facilement un patient ou une invitation, j'ai mis en place une **recherche dynamique filtrée côté serveur**, intégrée au système de pagination. Voici les différentes étapes avec le code correspondant.

Création d'un état local pour la recherche

```
const [searchQuery, setSearchQuery] = useState("");
```

Cette ligne permet de **stocker dynamiquement la valeur** du champ de recherche dans l'état local React.

Champ de recherche (input JSX)

```
const [searchQuery, setSearchQuery] = useState("");
<Input
  placeholder={
    activeView === "patients"
      ? "Rechercher des patients..."
      : "Rechercher des invitations..."
  }
  className="pl-8 w-full"
  value={searchQuery}
  onChange={(e) => setSearchQuery(e.target.value)}
  onKeyDown={(e) => {
    if (e.key === "Enter") {
      if (activeView === "patients") refetchPatients();
      else if (activeView === "invitations") refetchInvites();
    }
    setCurrentPage(1);
    setSearchParams({ page: "1" });
  }}
/>
```

Ce champ met à jour `searchQuery` à chaque frappe, et relance la recherche sur appui de la touche **Entrée**. Il remet aussi la pagination à la **page 1**.

Utilisation du paramètre `searchQuery` dans la requête patients

```
const { data: patients, refetch: refetchPatients } = useQuery({
  queryKey: ["patients-page", currentPage, pageSize],
  queryFn: async () => {
    if (currentUserQuery.data?.role === "therapist") {
      return await getPatients({
        searchQuery: searchQuery,
        pageParam: currentPage,
        limit: pageSize,
        therapistId: currentUserQuery.data?.id,
      });
    } else if (currentUserQuery.data?.role === "near") {
      return await getPatientsNear({
        searchQuery: searchQuery,
        pageParam: currentPage,
        limit: pageSize,
        nearId: currentUserQuery.data?.id,
      });
    }
  },
});
```

En fonction du rôle, la requête appelle `getPatients` ou `getPatientsNear`, en passant **le paramètre `searchQuery`** pour filtrer côté serveur.

Recherche également sur les invitations

```
const pendingInvitesQuery = useQuery({
  queryKey: ["pending-invites", currentPage, pageSize],
  queryFn: () => {
    if (currentUserQuery.data?.role === "therapist") {
      return getPendingInvites({
        therapistId: currentUserQuery.data?.id,
        page: currentPage,
        limit: pageSize,
        searchQuery,
      });
    } else if (currentUserQuery.data?.role === "near") {
      return getPendingInvitesNear({
        nearId: currentUserQuery.data?.id,
        page: currentPage,
        limit: pageSize,
        searchQuery, // ← Ce champ est bien utilisé ici aussi !
      });
    }
  },
  enabled: activeView === "invitations" && !!currentUserQuery.data?.id,
});
```

Quand tu tapes dans le champ de recherche et que tu appuies sur Entrée, la requête est relancée avec la valeur de searchQuery, ce qui filtre les résultats affichés dans la vue "invitations".
La pagination (page, limit) et la recherche (searchQuery) fonctionnent ensemble.

Requête côté service (API)

`getPatients` :

```
export const getPatients = async ({ searchQuery, pageParam, therapistId, limit = 5 }) => {
  const response = await fetch(
    `${API_URL}therapists/${therapistId}/patients?search=${searchQuery}&page=${pageParam}&limit=${limit}`,
    // ...
  );
  return await response.json();
};
```

`getPatientsNear` :

```
export const getPatientsNear = async ({ searchQuery, pageParam, nearId, limit }) => {
  const response = await fetch(
    `${API_URL}nears/${nearId}/patients?search=${searchQuery}&page=${pageParam}&limit=${limit}`,
    // ...
  );
  return await response.json();
};
```

`getPendingInvites` :

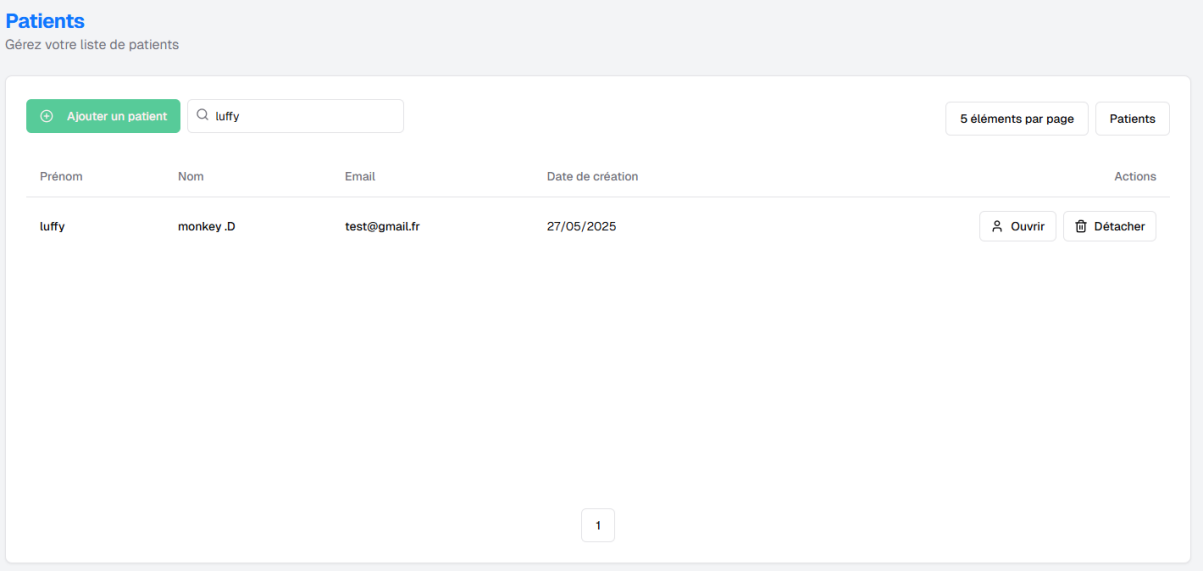
```
export const getPendingInvites = async ({ therapistId, page, limit, searchQuery }) => {
  const response = await fetch(
    `${API_URL}therapists/${therapistId}/pending-invites?search=${searchQuery}&page=${page}&limit=${limit}`
    // ...
  );
  return await response.json();
};
```

`getPendingInvitesNear` :

```
export const getPendingInvitesNear = async ({ nearId, page, limit, searchQuery }) => {
  const response = await fetch(
    `${API_URL}nears/${nearId}/pending-invites?search=${searchQuery}&page=${page}&limit=${limit}`
    // ...
  );
  return await response.json();
};
```

Chaque service utilise **le paramètre `searchQuery` dans l'URL** pour filtrer côté API. C'est une recherche **full back-end**, donc scalable.

Lorsqu'un utilisateur saisit du texte dans le champ de recherche, la valeur est stockée en temps réel dans l'état `searchQuery`. En appuyant sur la touche **Entrée**, une nouvelle requête est automatiquement déclenchée, tenant compte de ce filtre, et la pagination est réinitialisée à la première page. Le paramètre `searchQuery` est alors transmis à l'API, permettant un **filtrage directement côté serveur**, garantissant des résultats pertinents et optimisés selon la recherche effectuée.



Chargement de la page suivante

Créer une requête avec React Query

On utilise `useQuery` pour lancer une requête qui va vérifier si la page suivante (page courante + 1) contient encore des patients.
Pourquoi ? Pour savoir si on doit afficher le bouton “page suivante” dans la pagination.

```
const nextPageQuery = useQuery({ ... });
```

Définir une clé unique pour la requête

On passe une clé `queryKey` qui dépend de :

- La page suivante (donc `currentPage + 1`)
- La taille de la page (`pageSize`)
- Le rôle de l’utilisateur (`therapist` ou `near`)

Cette clé permet à React Query de mettre en cache et différencier les requêtes.

```
queryKey: ["check-next-page", currentPage + 1, pageSize, currentUserQuery.data?.role],
```

Écrire la fonction qui récupère les données de la page suivante

Dans `queryFn`, on appelle la bonne API selon le rôle :

- Si l’utilisateur est un **thérapeute**, on appelle `getPatients`
- Sinon, s’il est un **near**, on appelle `getPatientsNear`

On demande la page `currentPage + 1` avec les mêmes critères de recherche (`searchQuery`) et la même limite (`pageSize`).

```
queryFn: async () => {
  if (currentUserQuery.data?.role === "therapist") {
    const nextPageInfo = await getPatients({ ... });
    return nextPageInfo.length > 0;
  } else if (currentUserQuery.data?.role === "near") {
    const nextPageInfo = await getPatientsNear({ ... });
    return nextPageInfo.length > 0;
  }
  return false;
}
```

On retourne `true` si on a au moins un patient sur la page suivante, `false` sinon.

Activer la requête uniquement si l'utilisateur est identifié

On utilise `enabled` pour ne pas lancer la requête si on n'a pas encore récupéré l'ID du user (pas de données à checker sinon).

```
enabled: !!currentUserQuery.data?.id,
```

Utiliser ce booléen dans l'interface

Dans le composant, on vérifie `nextPageQuery.data` (vrai ou faux) pour savoir s'il faut afficher ou non le bouton "page suivante" et les liens correspondants.

Pour gérer la pagination et savoir s'il faut afficher un bouton vers la page suivante, on utilise une requête React Query qui demande la page *courante* + *1* via l'API adaptée au rôle de l'utilisateur (thérapeute ou "near"). Cette requête vérifie si la page suivante contient au moins un patient, ce qui permet de retourner un booléen vrai ou faux. La requête ne s'exécute que si l'utilisateur est bien identifié. En fonction de ce résultat, l'interface affiche ou non le contrôle permettant de passer à la page suivante, évitant ainsi de proposer une navigation vers une page vide.

Changement de page et la mise à jour des paramètres d'URL

Définir la fonction handlePageChange :

Elle met à jour la page courante et modifie les paramètres d'URL.

```
const handlePageChange = (newPage) => {
  setCurrentPage(newPage);  // Met à jour la page dans le state React
  setSearchParams({         // Met à jour les query params dans l'URL
    page: newPage.toString(),
    size: pageSize.toString(),
  });
};
```

Appeler cette fonction au clic sur les boutons de pagination :

Par exemple, pour passer à la page précédente ou suivante :

```
<PaginationPrevious onClick={() => handlePageChange(currentPage - 1)} />
<PaginationNext onClick={() => handlePageChange(currentPage + 1)} />
```

Utiliser React Query pour récupérer les données à chaque changement de page :

Le Hook `useQuery` est configuré avec `currentPage` et `pageSize` dans la clé de requête, ce qui force le rafraîchissement quand ils changent.

```
const { data: patients } = useQuery({
  queryKey: ["patients-page", currentPage, pageSize],
  queryFn: () =>
    getPatients({
      pageParam: currentPage,
      limit: pageSize,
      // autres paramètres
    }),
  enabled: !!currentUserQuery.data?.id,
});
```

La fonction `handlePageChange` met à jour la page active à la fois dans le state React et dans l'URL via les query params. Ce changement déclenche automatiquement une nouvelle requête des données grâce à React Query qui utilise `currentPage` dans sa clé. En plus, en mettant à jour l'URL, on permet une navigation plus fluide, avec un historique navigateur cohérent et un rechargement possible à la même page. En clair, tu contrôles ta pagination proprement, côté UI et côté données, en synchronisant tout ça à la perfection.

```
localhost:5173/patients?page=1&size=5
```

Sélecteur du nombre d’éléments par page

Fonction pour changer la taille de la page (nombre d’éléments affichés par page) :

Quand l’utilisateur sélectionne un nouveau nombre d’éléments par page, il faut :

- Mettre à jour le state `pageSize`,
- Remettre la pagination à la page 1,
- Modifier les paramètres d’URL pour que tout soit cohérent.

```
const handlePageSizeChange = (newSize) => {
  setPageSize(newSize); // Met à jour le nombre d'éléments par page
  setCurrentPage(1); // Remet à la page 1 car la pagination repart à zéro
  setSearchParams({
    // Met à jour les query params dans l'URL
    page: "1",
    size: newSize.toString(),
  });
};
```

Appel dans le DropdownMenu pour changer la taille de la page :

Ici, on récupère la valeur du dropdown et on lance `handlePageSizeChange`.

```
<DropdownMenuRadioGroup
  value={pageSize.toString()}
  onValueChange={(value) => {
    const newSize = Number(value);
    handlePageSizeChange(newSize);
  }}
>
  <DropdownMenuRadioItem value="5">5 éléments</DropdownMenuRadioItem>
  <DropdownMenuRadioItem value="10">10 éléments</DropdownMenuRadioItem>
  <DropdownMenuRadioItem value="20">20 éléments</DropdownMenuRadioItem>
  {/* etc... */}
</DropdownMenuRadioGroup>
```

React Query se charge de recharger les données avec la nouvelle taille automatiquement :

Puisque `pageSize` est dans la clé de la requête, ça déclenche une nouvelle requête avec la nouvelle limite.

Quand tu changes le nombre d’éléments par page via le dropdown, la fonction dédiée met à jour le `pageSize`, remet la page active à 1 (sinon tu pourrais rester coincé sur une page vide ou inexistante), et met à jour l’URL pour que ce soit bien transparent. React Query, lui, détecte ce changement dans sa clé de cache, ce qui force le rechargement des données en fonction de ce nouveau `pageSize`. Résultat : un contrôle fluide et propre de la pagination, que ce soit pour naviguer dans les pages ou ajuster la quantité d’informations affichées à la fois. Simple et efficace, comme on aime.

Ajout des utilisateurs “Nears”

Récupérer les patients d’un near

Fonction : getPatientsNear

```
export const getPatientsNear = async ({ searchQuery, pageParam, nearId, limit }) => {
  const response = await fetch(
    `${API_URL}nears/${nearId}/patients?search=${searchQuery}&page=${pageParam}&limit=${limit}`,
    {
      method: "GET",
      credentials: "include",
      headers: { "Content-Type": "application/json" },
    }
  );
  if (!response.ok) {
    const error = await response.json();
    throw error;
  }
  return await response.json();
};
```

Cette requête permet de récupérer la liste paginée des patients associés à un "near", avec possibilité de recherche (searchQuery). Utilisée dans le composant pour afficher la liste des patients côté proche.

Ajouter un patient à un near

Mutation : addPatientNear

```
export const addPatientNear = async ({ patient, nearId }) => {
  const response = await fetch(`${API_URL}nears/${nearId}/attach-patient`, {
    method: "POST",
    credentials: "include",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ patientEmail: patient.email }),
  });
  if (!response.ok) {
    const error = await response.json();
    throw error;
  }
  return await response.json();
};
```

Cette mutation permet à un proche d’inviter un patient via son email.

Elle est utilisée lors de la soumission du formulaire "Ajouter un patient" si l’utilisateur connecté est un "near".

Détacher (supprimer) un patient d’un near

Mutation : deletePatientNear

```
export const deletePatientNear = async ({ patientId, nearId }) => {
  const response = await fetch(`${API_URL}nears/${nearId}/detach-patient/${patientId}`, {
    method: "POST",
    credentials: "include",
    headers: { "Content-Type": "application/json" },
  });
  if (!response.ok) {
    const error = await response.json();
    throw error;
  }
  return await response.json();
};
```

Cette mutation permet à un proche de détacher un patient de sa liste.

Elle est appelée lors de la confirmation de suppression d'un patient dans l'interface.

Récupérer les invitations en attente d'un near

Fonction : getPendingInvitesNear

```
export const getPendingInvitesNear = async ({ nearId, page, limit, searchQuery }) => {
  const response = await fetch(
    `${API_URL}nears/${nearId}/pending-invites?search=${searchQuery}&page=${page}&limit=${limit}`,
    {
      method: "GET",
      credentials: "include",
      headers: { "Content-Type": "application/json" },
    }
  );
  if (!response.ok) {
    const error = await response.json();
    throw error;
  }
  return await response.json();
};
```

Cette requête permet de récupérer la liste paginée des invitations envoyées par un "near" et qui n'ont pas encore été acceptées. Utilisée pour afficher la vue "Invitations en cours" côté proche.

Annuler une invitation envoyée par un near

Mutation : cancelInviteNear

```
export const cancelInviteNear = async ({ inviteId, nearId }) => {
  const response = await fetch(`${API_URL}nears/${nearId}/invites/${inviteId}/cancel`, {
    method: "POST",
    credentials: "include",
    headers: { "Content-Type": "application/json" },
  });
  if (!response.ok) {
    const error = await response.json();
    throw error;
  }
  return await response.json();
};
```

Cette mutation permet à un proche d'annuler une invitation envoyée à un patient.

Elle est utilisée lors de la confirmation d'annulation dans la liste des invitations.

Utilisation dans le composant Patients.jsx

Dans le composant, chaque action (ajout, suppression, récupération, annulation) vérifie le rôle de l'utilisateur et utilise la bonne fonction :

```tsx
// Ajout d'un patient
const { mutateAsync } = useMutation({
  mutationFn: (data) => {
    if (currentUserQuery.data?.role === "therapist") {
      return addPatient({ patient: data, therapistId: currentUserQuery.data?.id });
    } else if (currentUserQuery.data?.role === "near") {
      return addPatientNear({ patient: data, nearId: currentUserQuery.data?.id });
    }
  },
  // ...
});

// Suppression d'un patient
const deleteMutation = useMutation({
  mutationFn: (patientId) => {
    if (currentUserQuery.data?.role === "therapist") {
      return deletePatient({ patientId, therapistId: currentUserQuery.data?.id });
    } else if (currentUserQuery.data?.role === "near") {
      return deletePatientNear({ patientId, nearId: currentUserQuery.data?.id });
    }
  },
  // ...
});

// Récupération des patients
const { data: patients } = useQuery({
  queryKey: ["patients-page", currentPage, pageSize],
  queryFn: async () => {
    if (currentUserQuery.data?.role === "therapist") {
      return await getPatients({ ... });
    } else if (currentUserQuery.data?.role === "near") {
      return await getPatientsNear({ ... });
    }
  },
  // ...
});

// Récupération des invitations en attente
const pendingInvitesQuery = useQuery({
  queryKey: ["pending-invites", currentPage, pageSize],
  queryFn: () => {
    if (currentUserQuery.data?.role === "therapist") {
      return getPendingInvites({ ... });
    } else if (currentUserQuery.data?.role === "near") {
      return getPendingInvitesNear({ ... });
    }
  },
  // ...
});

// Annulation d'une invitation
const cancelInviteMutation = useMutation({
  mutationFn: (inviteId) => {
    if (currentUserQuery.data?.role === "therapist") {
      return cancelInvite({ inviteId, therapistId: currentUserQuery.data?.id });
    } else if (currentUserQuery.data?.role === "near") {
      return cancelInviteNear({ inviteId, nearId: currentUserQuery.data?.id });
    }
  },
  // ...
});
```

Chaque action (ajout, suppression, récupération, annulation) a maintenant une version dédiée pour les "nears".

Le composant détecte le rôle de l'utilisateur et utilise la bonne requête/mutation.

Cela permet aux proches d'avoir exactement les mêmes fonctionnalités que les thérapeutes pour la gestion des patients et des invitations, tout en gardant le code centralisé et maintenable.

Fonctionnalités restantes

Voici un résumé des dernières fonctionnalités mises en place ou en cours de développement pendant mon stage :

- **Annulation des invitations** : un bouton d'action permet d'annuler une invitation envoyée à un "near", avec un dialogue de confirmation et une requête de suppression côté API.
- **Affichage du statut des invitations** : les invitations sont marquées comme "En attente" ou "Expirée" en fonction de leur date d'expiration, avec un code couleur distinctif (jaune ou rouge).
- **Responsiveness de l'interface** : toutes les vues ont été optimisées pour un affichage fluide sur desktop et mobile, avec des composants adaptatifs comme les tableaux ou les cards.
- **Sécurité et permissions** : certaines actions ne sont possibles que si l'utilisateur est un thérapeute ; le rôle est vérifié à plusieurs endroits pour sécuriser l'accès aux fonctionnalités sensibles.

Celle-ci seront disponible dans les Annexes.

Mon stage m'a permis de participer activement au développement d'une plateforme de gestion de patients, en intégrant notamment des fonctionnalités avancées comme la pagination dynamique, la recherche filtrée côté serveur, l'adaptation à différents rôles utilisateurs ("therapist" et "near"), et la gestion des invitations. L'ensemble de ces développements a été réalisé avec des technologies modernes telles que React, TypeScript, PostgreSQL, et Drizzle ORM, dans un environnement structuré autour de composants réutilisables (shadcn/ui) et d'un backend conteneurisé. Cette expérience m'a permis de monter en compétence techniquement, mais aussi de mieux comprendre les enjeux de structuration, de lisibilité du code, et de collaboration dans un cadre professionnel.

Conclusion

Ce stage de six semaines au sein de l'équipe d'ingénieurs de l'Université de Lille à la Plaine Images a été une véritable immersion dans le monde du développement web professionnel. J'ai pu découvrir et manipuler des technologies modernes comme React, TypeScript, Drizzle ORM et PostgreSQL, tout en développant des fonctionnalités concrètes telles que la pagination, la recherche filtrée ou encore la gestion des rôles utilisateurs.

Ce projet m'a permis de renforcer mes compétences techniques mais aussi de mieux appréhender l'organisation d'un projet en équipe, l'importance d'un code structuré, et les bonnes pratiques liées au travail collaboratif. J'ai appris à être plus autonome, à poser les bonnes questions, et à trouver des solutions aux problèmes rencontrés.

Je repars de cette expérience avec une vision plus claire de mon avenir professionnel et une motivation encore plus forte pour continuer dans la voie du développement web. Ce stage a non seulement consolidé mes acquis de première année de BTS SIO SLAM, mais m'a également donné un avant-goût concret du métier vers lequel je me dirige.

Annexe

Annulation des invitations :

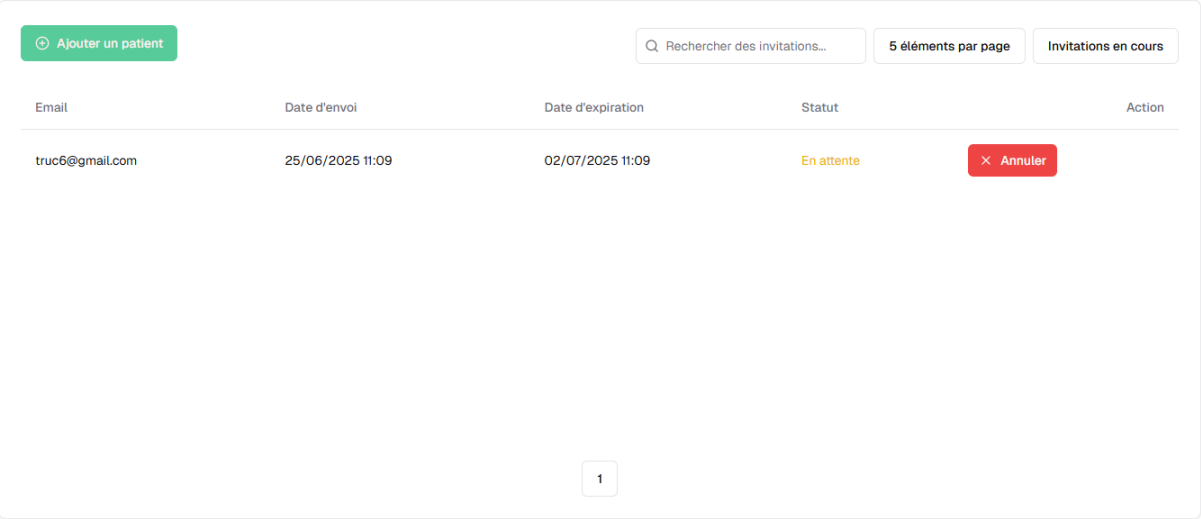
Principe

L’annulation d’une invitation permet à un thérapeute ou un proche (near) de retirer une invitation envoyée à un patient avant qu’elle ne soit acceptée. Cette action est irréversible et retire l’accès à l’inscription pour le patient invité.

Fonctionnement côté interface (front-end)

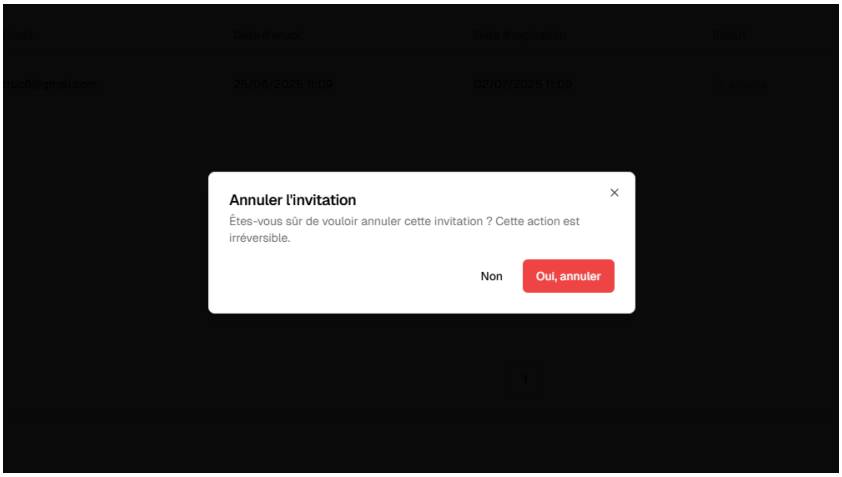
Affichage :

Dans la vue "Invitations en cours", chaque invitation listée possède un bouton "Annuler".



Ouverture d’un dialogue de confirmation :

Lorsqu’on clique sur "Annuler", une boîte de dialogue s’ouvre pour demander la confirmation de l’action.



Confirmation :

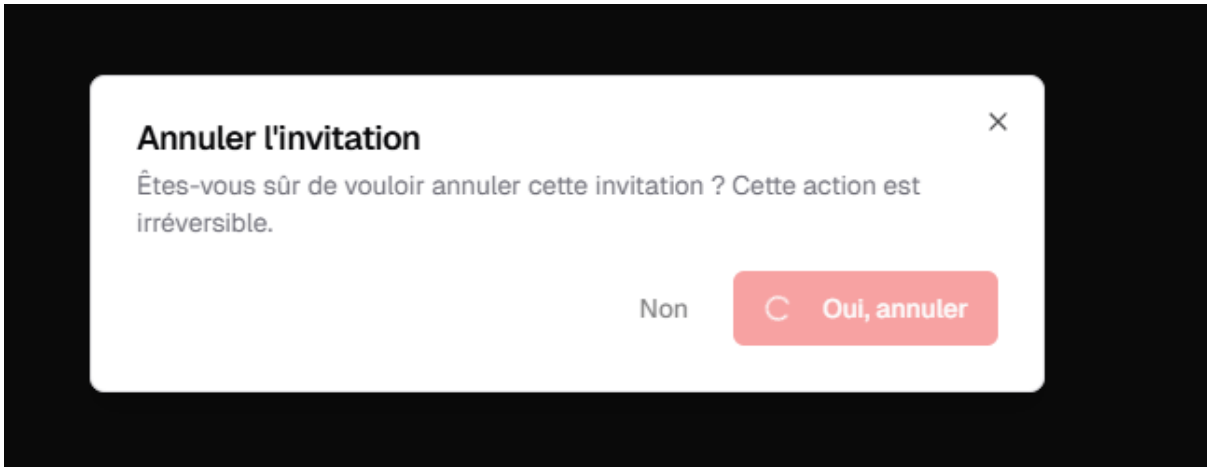
Si l'utilisateur confirme, la requête d'annulation est envoyée à l'API.

Retour visuel :

Un indicateur de chargement s'affiche pendant l'opération.

Si l'annulation réussit, l'invitation disparaît de la liste.

En cas d'erreur, un message d'erreur s'affiche.



Fonctionnement côté code

Bouton d'annulation et dialogue

```
<Button
  variant="destructive"
  size="sm"
  onClick={() => setSelectedInvite(invite)}
  className="flex items-center gap-2"
>
  <X className="h-4 w-4" />
  Annuler
</Button>
```

Ouvre le dialogue de confirmation et sélectionne l'invitation à annuler.

Dialogue de confirmation

Affiche la confirmation et gère l'état de chargement/erreur.

Mutation d'annulation

```
const cancelInviteMutation = useMutation({
  mutationFn: (inviteId) => {
    if (currentUserQuery.data?.role === "therapist") {
      return cancelInvite({
        inviteId,
        therapistId: currentUserQuery.data?.id,
      });
    } else if (currentUserQuery.data?.role === "near") {
      return cancelInviteNear({
        inviteId,
        nearId: currentUserQuery.data?.id,
      });
    }
  },
  onSuccess: () => {
    queryClient.invalidateQueries({ queryKey: ["pending-invites"] });
    setCancelError(null);
    setIsCancelDialogOpen(false);
    setSelectedInvite(null);
  },
  onError: (error) => {
    setCancelError(error.message);
  },
});
```

Envoie la requête d'annulation à l'API.
Rafraîchit la liste des invitations après succès.

Résumé

L'utilisateur peut annuler une invitation non encore acceptée.
Une confirmation est demandée pour éviter les erreurs.

L’annulation retire l’invitation de la liste et empêche le patient de s’inscrire.

Affichage du statut des invitations

Principe

Chaque invitation envoyée à un patient affiche un statut visuel indiquant si elle est encore valide ("En attente") ou si elle a expiré ("Expirée"). Ce statut est déterminé automatiquement en fonction de la date d’expiration de l’invitation.

Fonctionnement côté interface (front-end)

Affichage du statut :

Dans la liste des invitations, une étiquette de couleur apparaît à côté de chaque invitation :

Jaune ("En attente") : l’invitation est encore valide, le patient peut l’utiliser.
Rouge ("Expirée") : la date d’expiration est dépassée, l’invitation n’est plus utilisable.
Code couleur :

Jaune (text-yellow-500) pour "En attente"
Rouge (text-red-500) pour "Expirée"

Fonctionnement côté code

```
{new Date(invite.inviteTokenExpiresAt) < new Date() ? (
  <span className="text-red-500 text-xs font-medium">Expirée</span>
) : (
  <span className="text-yellow-500 text-xs font-medium">En attente</span>
)}
```

Explication :
Si la date d’expiration (invite.inviteTokenExpiresAt) est passée, le statut "Expirée" s’affiche en rouge.
Sinon, le statut "En attente" s’affiche en jaune.
Résumé
Le statut de chaque invitation est affiché automatiquement selon la date d’expiration.
Un code couleur permet de repérer rapidement les invitations valides ou expirées.
Cela améliore la lisibilité et l’expérience utilisateur pour la gestion des invitations.

Responsiveness de l’interface

Principe

L’interface de l’application a été conçue pour offrir une expérience utilisateur optimale aussi bien sur ordinateur (desktop) que sur mobile. Les composants s’adaptent dynamiquement à la taille de l’écran pour garantir la lisibilité et la facilité d’utilisation.

Fonctionnement côté interface (front-end)

Affichage adaptatif :

Sur desktop, les données sont présentées dans des tableaux larges et détaillés, permettant une vue d’ensemble complète.

Sur mobile, l’affichage bascule automatiquement vers des cartes (cards) empilées, plus lisibles et adaptées à la navigation tactile.

Composants réactifs :

Utilisation de classes utilitaires (hidden md:block, block md:hidden, etc.) pour afficher ou masquer certains éléments selon la taille de l’écran.

Les boutons, champs de recherche et menus sont également adaptés pour rester accessibles et ergonomiques sur tous les supports.

Exemple de code :

```jsx
function PatientsList({ patients }) {
  return (
    <>
      {/* Affichage mobile : cards */}
      <div className="block md:hidden">
        {patients.map((patient) => (
          <div
            key={patient.id}
            className="border rounded-lg p-4 mb-3 flex flex-col gap-2 bg-muted"
          >
            <span className="font-semibold">
              {patient.firstname} {patient.lastname}
            </span>
            <span className="text-xs text-muted-foreground">{patient.email}</span>
            <span className="text-xs text-muted-foreground">
              Créé le {new Date(patient.createdAt).toLocaleDateString("fr-FR")}
            </span>
            {/* Actions ou autres infos ici */}
          </div>
        ))}
      </div>

      {/* Affichage desktop : tableau */}
      <div className="overflow-x-auto hidden md:block">
        <table className="min-w-[600px] w-full border">
          <thead>
            <tr>
              <th className="px-4 py-2">Prénom</th>
              <th className="px-4 py-2">Nom</th>
              <th className="px-4 py-2">Email</th>
              <th className="px-4 py-2">Date de création</th>
            </tr>
          </thead>
          <tbody>
            {patients.map((patient) => (
              <tr key={patient.id} className="border-t">
                <td className="px-4 py-2">{patient.firstname}</td>
                <td className="px-4 py-2">{patient.lastname}</td>
                <td className="px-4 py-2">{patient.email}</td>
                <td className="px-4 py-2">
                  {new Date(patient.createdAt).toLocaleDateString("fr-FR")}
                </td>
              </tr>
            ))}
          </tbody>
        </table>
      </div>
    </>
  );
}
```

Résumé

L’interface s’adapte automatiquement à la taille de l’écran :

Tableaux sur desktop
Cards sur mobile

Les composants restent lisibles, accessibles et agréables à utiliser sur tous les appareils.

Cette approche améliore l’expérience utilisateur et rend l’application utilisable partout.