ANTOINE DINH

RAPPORT DE PROJET

Février 2025

AP3 GSB

Bts SIO OPtion SLAM

SOMMAIRE

01

CONTEXTE GSB

Contexte GSB

Galaxy Swiss Bourdin (GSB) a émergé d'une fusion stratégique entre deux grands noms de l'industrie pharmaceutique : le géant américain Galaxy, spécialisé dans le traitement des maladies virales comme le SIDA et les hépatites, et le conglomérat européen Swiss Bourdin, connu pour ses médicaments traditionnels. Cette collaboration en 2009 a donné naissance à un leader incontesté du secteur pharmaceutique.

Siège Social et Administration

GSB a établi son siège administratif à Paris pour son entité européenne, tandis que le siège social de la multinationale se situe à Bagneux, dans les Hauts-de-Seine. Ces choix stratégiques reflètent l'engagement de GSB envers l'excellence et l'innovation au cœur de l'Europe.

Force de Vente et Présence Médicale

Avec une équipe de vente composée de 480 visiteurs médicaux en France métropolitaine et 60 dans les départements et territoires d'outre-mer, GSB maintient une présence significative et influente dans le domaine médical, assurant une couverture complète et efficace sur tout le territoire national.

Expertise Clinique et Commerciale

GSB est reconnu comme un expert mondial en études cliniques, contribuant de manière significative à la progression de la recherche médicale. Au niveau national, GSB excelle dans la vente de médicaments en B2B avec les professionnels de santé et en C2B avec les clients particuliers, consolidant ainsi sa position d'expert en commercialisation de solutions thérapeutiques.

02

BESOINS ET OBJECTIFS

Expression des besoins et objectifs

Consultation des Stocks : Tous les utilisateurs doivent pouvoir consulter la liste des médicaments et du matériel disponible. Cette consultation inclura des informations telles que le nom du produit, sa description, et la quantité actuellement en stock. La distinction entre médicaments et matériel doit être clairement établie pour faciliter la gestion et la recherche.

Passation de Commandes : Les utilisateurs standards doivent avoir la possibilité de passer des commandes pour des médicaments ou du matériel. Ces commandes seront initialement marquées comme étant en attente, indiquant qu'elles n'ont pas encore été approuvées ou traitées par un administrateur. Cela permet de centraliser les demandes et d'assurer un suivi rigoureux des besoins.

Gestion des Commandes : Les utilisateurs administrateurs doivent pouvoir consulter la liste complète des commandes, y compris celles en attente. Ils peuvent alors approuver ou rejeter ces commandes. L'approbation d'une commande entraîne une mise à jour de la quantité de stock concernée, reflétant la sortie (pour les commandes approuvées) ou l'annulation (pour les commandes rejetées). Ce processus garantit une gestion efficace et transparente des stocks.

Ajout de Stock : Les administrateurs doivent également pouvoir saisir des commandes auprès des fournisseurs pour ajouter du stock. Chaque entrée de stock est considérée comme un mouvement et doit être enregistrée de manière à refléter une augmentation de la quantité de stock disponible. Cela permet de maintenir un inventaire précis et à jour.

Ces objectifs sont essentiels pour développer une application robuste et sécurisée qui répond aux exigences des professionnels de santé. Ils s'inscrivent également dans les meilleures pratiques en matière de gestion des stocks et de sécurité des patients.

03

ANALYSE FONCTIONNELLE ET CHOIX TECHNIQUES

Analyse fonctionnelle et choix techniques

Listage des fonctionnalités

L'application créée pour GSB présente diverses fonctionnalités clés, divisées pour objectif finale la création de l'ordonnance

Authentification et gestion des utilisateurs :

- Écran de connexion pour les utilisateurs: Permet aux utilisateurs de se connecter de manière sécurisée.
- Écran de gestion des rôles : Offre la possibilité de visualiser et de monter le niveau d'autorisation des utilisateurs "classiques".

Gestion des stocks :

- 1.Consultation du stocks : Permet à tous les utilisateurs de vérifier les stocks avec les informations comme le nom, la description, la quantité et le type
- 2.Ajout d'un nouveau stock : Offre la possibilité d'intégrer un nouveau type de stock qui aura par défaut une valeur de 1.

Gestion des commandes:

- Liste des commandes: Affiche toutes les commandes sans distinction avec l'utilisateur liée, le statut, la date de la commande la quantité er le stock concerné.
- Création d'une nouvelle commande: Permet de générer une nouvelle commande.
- Suppression d'une commande existante : Permet la suppression d'une commande qui a le statut : "en attente".

Gestion des commandes en attente (admin):

- Liste des commandes qui ont le statut en attente : Affiche uniquement les commandes avec le statut en attente (attention besoin temps d'attente entre la fonctionnalité "commandes" et "commande en attente"
- Changement de statut de la commande : Possibilité d'inclure ou de refuser une commande et si accepté influence le stock et crée une entrée dans la liste des mouvements avec comme type "sortie".

Gestion de commandes (admin) :

- Même liste de commandes
- Création d'une commande admin : cette commande ne passe par l'étape en attente, elle est directement accepté ce qui augmente la quantité de stock et crée un mouvement comme type "entrée"

Gestion des mouvements

- Affichage des mouvements : affiche tous les mouvements avec leur type, leur quantité, le stock concerné ainsi que la date du mouvement.

Langage et technologies utilisées

React :

- **Réactivité et Performance : React est une bibliothèque JavaScript populaire pour la création d'interfaces utilisateur réactives et performantes. Elle permet de construire des composants réutilisables, ce qui facilite le développement et la maintenance de l'application.**
- **Écosystème Riche : Avec un vaste écosystème de bibliothèques et d'outils, React offre de nombreuses solutions pour les besoins courants du développement web, comme la gestion de l'état et le routage.**

Next.js :

- **Rendu Côté Serveur (SSR) et Génération de Sites Statique (SSG) : Next.js permet de combiner le meilleur des deux mondes avec le rendu côté serveur pour des performances optimales et la génération de sites statiques pour une meilleure SEO et une livraison rapide des pages.**
- **Facilité de Déploiement : Next.js simplifie le déploiement des applications grâce à son intégration avec des plateformes comme Vercel, permettant une mise en production rapide et efficace.**
- **Fonctionnalités Avancées : Next.js offre des fonctionnalités avancées comme le pré-rendu, le chargement dynamique et les API routes, ce qui en fait un choix puissant pour les applications modernes**

Prisma :

- **ORM Moderne : Prisma est un ORM (Object-Relational Mapping) moderne qui simplifie l'interaction avec la base de données. Il offre une syntaxe intuitive et des migrations automatiques, ce qui accélère le développement et réduit les erreurs.**
- **Type Safety : Prisma génère des types TypeScript pour les modèles de données, assurant une sécurité de type et réduisant les bugs liés aux types**

Supabase :

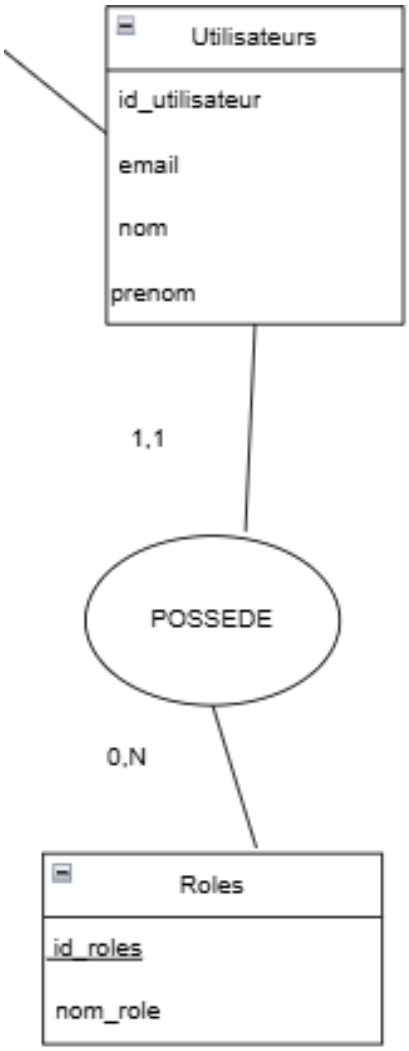
- **Backend as a Service : Supabase fournit une solution complète de backend, incluant une base de données PostgreSQL, l'authentification, et le stockage de fichiers. Cela permet de se concentrer sur le développement de l'application sans se soucier de la gestion de l'infrastructure.**
- **Intégration Facile : Supabase s'intègre facilement avec Prisma et Next.js, offrant une solution cohérente et efficace pour le développement full-stack**

Ces technologies sont bien adaptées pour le développement de notre nouvelle application, offrant à la fois flexibilité, performance et facilité de maintenance. Elles permettent de construire une application moderne et scalable, répondant aux besoins des utilisateurs tout en assurant une gestion efficace des données.

04

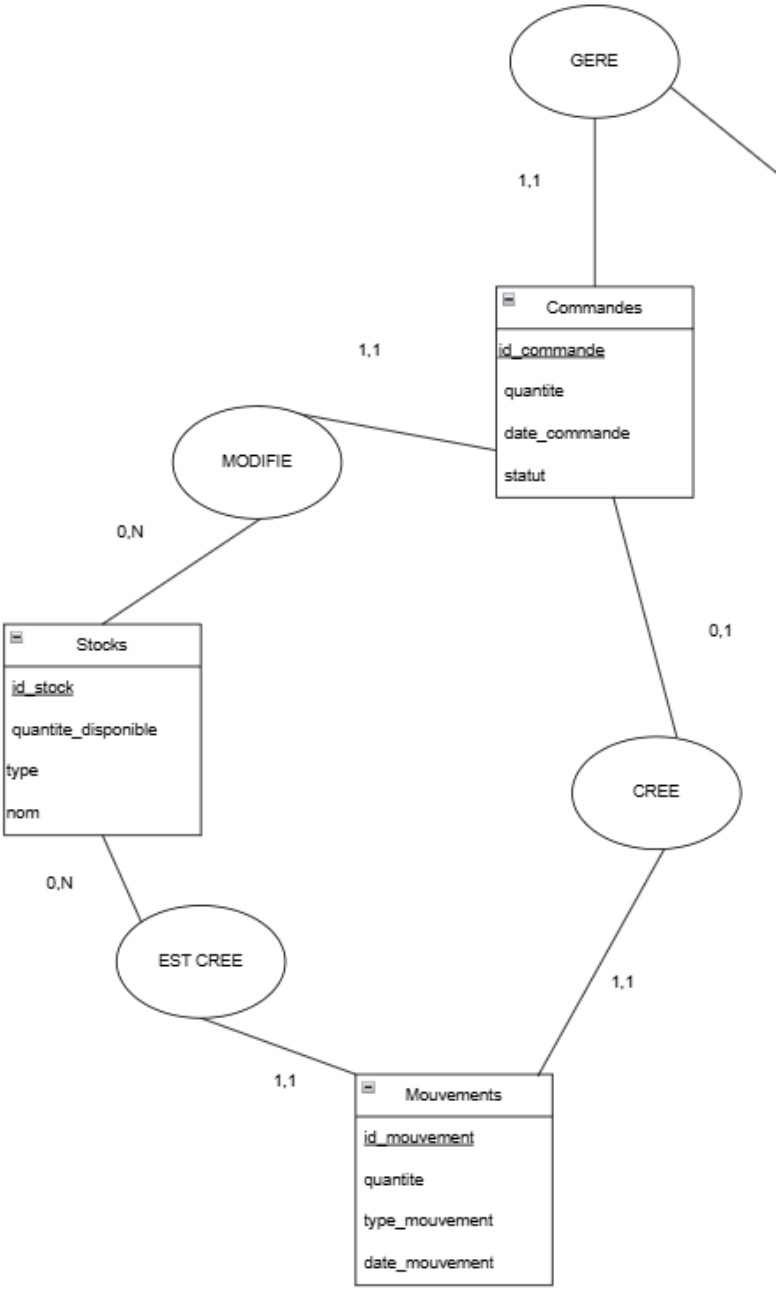
MA BDD

MCD



1. Utilsateurs: Cette entité représente les utilisateurs du système. Chaque utilisateur a un identifiant unique (id_utilisateur), ainsi que son email, nom, date de naissance et son prénom.
2. Roles: Cette entité contient les informations relatives aux rôles. Chaque rôle a un identifiant unique (id_role) ainsi que son nom de rôles.

3. Commandes: Cette entité contient les informations des commmandes. Chaque patient a un identifiant unique (id_commande), ainsi que sa quantité, sa date de commande et son statut.



4.Mouvements: Cette entité donne tous les mouvements contenant tous un identifiant(id_mouvement), une quantité, le type et la doute de mouvement.

5.Stocks: Cette entité donne tous les stocks contenant tous un identifiant(id_stock), une quantité disponible, le type et le nom.

Les relations entre ces entités sont également dépeintes, montrant comment elles interagissent au sein du système pour la gestion des fiches de frais :

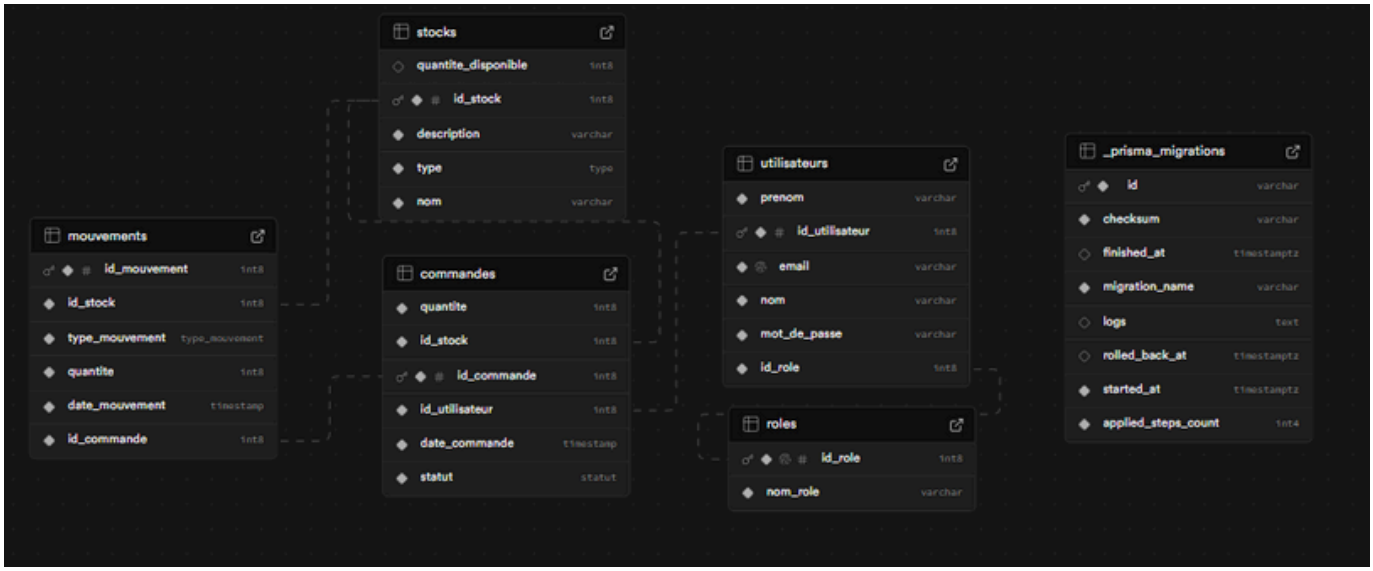
- Roles a une relation 0-à-plusieurs avec Utilisateur.
- Mouvements une relation un-à-un avec Stocks ainsi qu'avec Commandes
- Stocks a des relation 0-à-plusieurs avec Mouvements et Commandes

MLD

Roles (id_role, nom_role)
Utilisateurs(id_utilisateur, email, nom, prenom, #id_role)
Commandes (id_commande, quantite, date_commande, statut, #id_stock, #id_utilisateur)
Stocks(id_stock, quantite_disponible, type, nom)
Mouvements(id_mouvement, quantite, type_mouvement, date_mouvement, #id_commande, #id_stock)

Une transformation du MCD en MLD permet de mieux structurer les entités pour la création dans la base de donnée, les identifiants sont mis en évidence en étant soulignés et les # permettent d'identifier les clés étrangères, on peut donc remarquer les dépendances entre les entités comme par exemple avec l'id_stock qui a été mis en clé étrangère dans commandes.

Schéma conceptuel Supabase



Après les 2 dernières étapes, une création de la base de donnée est possible, toutes les informations renseignés avec le MLD sont mises avec les bons types .
Les relations sont bien mis évidence pour faciliter la compréhension et la maintenance pour le futur. des symboles de clé sont mis en évidence pour les clés.

05

ORGANISTAION DU CODE

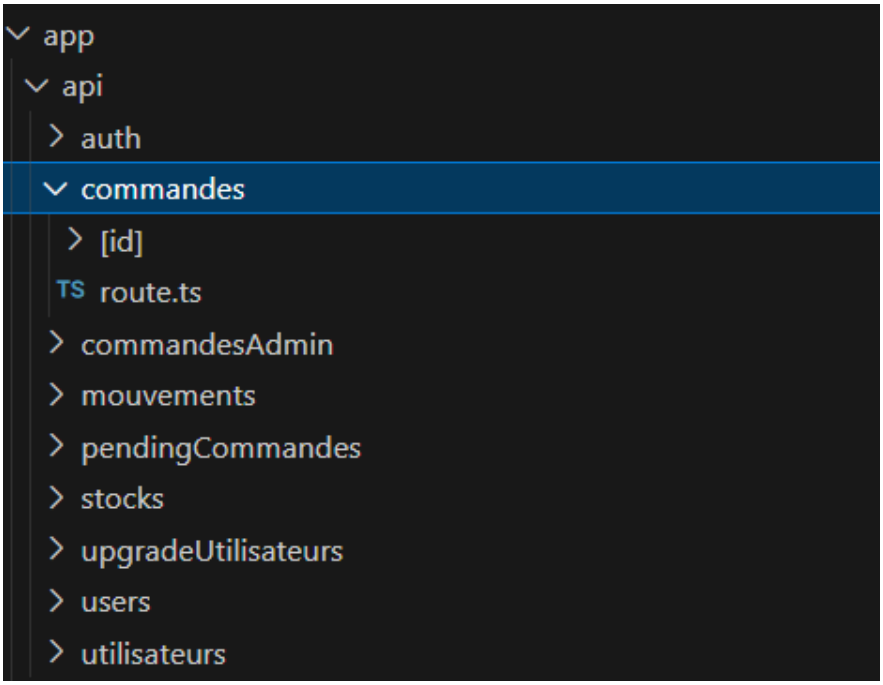
Organisation du code

React

L'utilisation de React permet une organsation du code avec division par module qui chacun a une utilité

Api

Les routes api dans les sous dossiers du dossiers api dans app, permet de faire un lien entre le front end et le backend (les services)



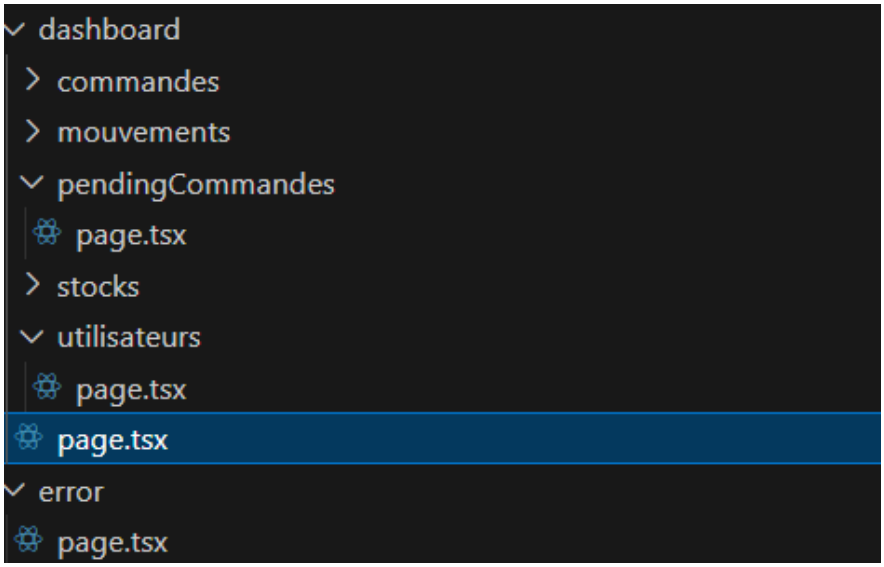
L'app

Les autres dossiers et fichiers qui ne sont pas dans le dossier api mais dans le dossier app permet le routage de l'application, il y a aussi la page d'erreur affichée éventuellement.

ARCHITECTURE DU PROJET

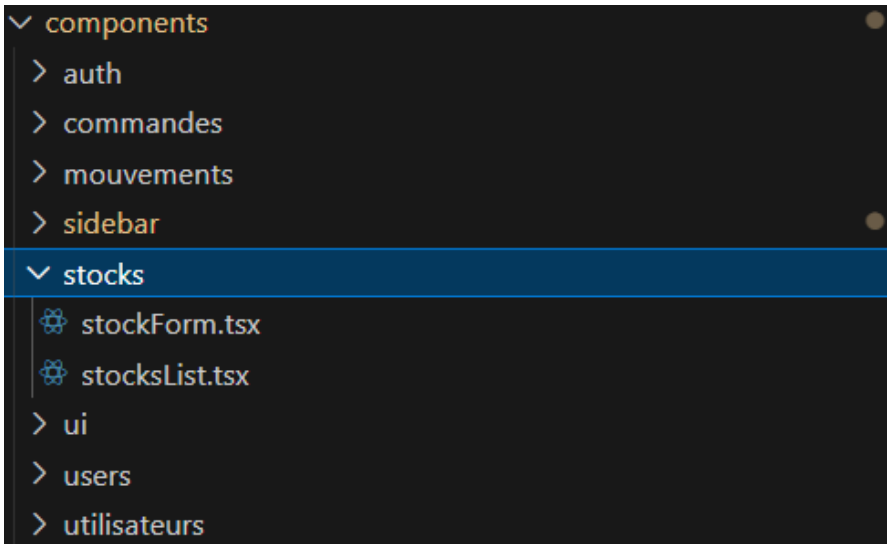
Voici l'organisation simplifiée de mon code pour l'application

```
AP3_NextJS/
│
├── app/
│   ├── api/
│   │   ├── commandes/
│   │   ├── mouvements/
│   │   ├── pendingCommandes/
│   │   ├── stocks/
│   │   ├── utilisateurs/
│   │   └── error/
│   │
│   ├── dashboard/
│   │   ├── commandes/
│   │   ├── mouvements/
│   │   ├── pendingCommandes/
│   │   ├── stocks/
│   │   ├── utilisateurs/
│   │   └── page.tsx
│   │
│   ├── components/
│   │   ├── auth/
│   │   ├── commandes/
│   │   ├── mouvements/
│   │   ├── sidebar/
│   │   ├── stocks/
│   │   ├── ui/
│   │   └── utilisateurs/
│   │
│   ├── context/
│   ├── hooks/
│   ├── lib/
│   │
│   ├── services/
│   │   ├── client.ts
│   │   ├── middleware.ts
│   │   ├── server.ts
│   │   └── utils/supabase/
│   │
│   ├── layout.tsx
│   ├── page.tsx
│   ├── actions.ts
│   ├── favicon.ico
│   ├── globals.css
│
├── prisma/
│   ├── migrations/
│   └── schema.prisma
│
├── public/
│   ├── images/
│
└── node_modules/
```



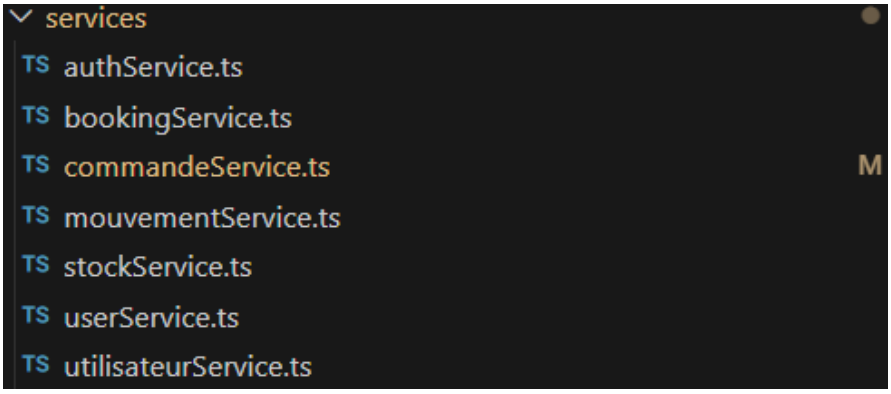
Components

Les composants seront ce qui sera le plus souvent les éléments affichés sur le frontend, il sera appelé sur les page.tsx vu précédemment.



Services

Les services servent à récupérer les données en backend de la base de données pour les envoyer ensuite à l'api qui renvoie ensuite au frontend.



06

PRÉSENTATION DU CODE

Présentation du code

LOGIN / INSCRIPTION

La gestion de l'authentification utilise un formulaire puis supabase qui redirige ensuite sur le Dashboard. Il y a aussi la possibilité de s'inscrire qui par la même occasion crée un utilisateur dans la table utilisateurs.

```tsx
export default function Login() {
  const { toast } = useToast();
  const router = useRouter();
  const [loading, setLoading] = useState(false);

  const { register, handleSubmit, watch } = useForm();

  const email = watch('email');
  const password = watch('password');

  const onSubmit = async (data: any) => {
    setLoading(true);

    const result = await authUserByEmailAndPassword(
      data.email,
      data.password
    );

    setLoading(false);

    if (result) {
      await handleRevalidate();
      location.reload();
      router.push("/");
    } else {
      toast({
        title: 'Erreur',
        description: 'Erreur lors de la connexion',
        variant: 'destructive',
      });
    }
  };
```

```ts
export const authUserByEmailAndPassword = async (
  email: string,
  password: string
): Promise<boolean> => {
  if (!email || !password) return false;

  const { error } = await supabase.auth.signInWithPassword({
    email: email,
    password: password,
  });

  if (error) return false;

  return true;
};
```

```ts
try {
  const { data, error } = await supabase.auth.signUp({
    email: email,
    password: password,
  });

  if (error) {
    console.error("Erreur lors de la création dans Supabase:", error);
    return false;
  }

  if (data.user) {
    try {
      const response = await fetch("/api/utilisateurs", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          email,
          password,
          nom,
          prenom,
        }),
      });
```

Logique de l'api

Pour réaliser une action, dans les pages ou les composant il y a un fetch de l'api avec selon les requêtes un body ou non puis la route api appelle le service associé.

```
const handleFormSubmit = async (data: z.infer<typeof CommandeFormSchema>) =>
  try {
    const updateData = {
      ...data,
      id_utilisateur : utilisateur?.id_utilisateur
    }
    await fetch("/api/commandes", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(updateData),
    });

    setIsDialogOpen(false);
    toast({
      title: "Success",
      description: "Commande créée",
      variant: "default",
    });
    CommandeListRef.current?.refresh();
  } catch (error) {
    console.error("Erreur lors de la création de la commande :", error);
  }
};
```

```
export async function POST(req: NextRequest) {
  try {
    const body = await req.json();
    const newCommande = await CreateCommande({
      id_utilisateur: body.id_utilisateur,
      quantite: body.quantite,
      id_stock: body.id_stock,
    });
    return NextResponse.json(newCommande, { status: 201 });
  } catch (error) {
    console.error("Error creating commande:", error);
    return NextResponse.json(
      { error: "Failed to create commande" },
      { status: 500 }
    );
  }
}
```

L'affichage de liste

Dans la majorité des fonctionnalités il y la possibilité de voir la liste d'une table associé, prenons par exemple "Stocks", dans ces situations j'utilise une interface personnalisé car ma table est en int8 ce qui n'est pas bien pris avec les fonctions et plus difficile à utiliser

```
enum type {
  medicament,
  materiel,
}

export interface SerializedStocks {
  id_stock: number;
  nom: string;
  description: string;
  quantite_disponible: number;
  type: type;
}
```

```
export async function GetAllStocks(): Promise<SerializedStocks[]> {
  try {
    const stocks = await prisma.stocks.findMany();
    const serializedStocks: SerializedStocks[] = JSON.parse(
      JSONbig.stringify(stocks)
    );
    return serializedStocks;
  } catch (error) {
    console.error(error);
    throw new Error("Failed to fetch stocks");
  }
}
```

Insertion d'élément

Pour la majorité des tables, il y a une fonctionnalité pour ajouter un objet à la table, il y a un exemple dans le service Stocks

```
export async function CreateStocks(data: {
  nom: string;
  description: string;
  type: $Enums.type;
}): Promise<SerializedStocks | null> {
  try {
    const stock = await prisma.stocks.create({
      data: {
        nom: data.nom,
        description: data.description,
        quantite_disponible: parseInt("0", 10),
        type: data.type,
      },
    });
    const serializedStocks: SerializedStocks = JSON.parse(
      JSONbig.stringify(stock)
    );
    return serializedStocks;
  } catch (error) {
    console.error("Error creating commande:", error);
    throw new Error("Failed to create commande");
  }
}
```

Mise à jour du rôle d'utilisateur

La fonction prend en paramètre l'id d'un utilisateur pour lui changer son rôle

```
export async function UpdateRoleUtilisateur(
  id_utilisateur: number
): Promise<SerializedUtilisateurs | null> {
  try {
    if (!id_utilisateur) {
      throw new Error("Utilisateur non trouvée.");
    }

    const updatedUtilisateur = await prisma.utilisateurs.update({
      where: { id_utilisateur: BigInt(id_utilisateur) },
      data: { id_role: 1 },
    });
    const serializedUtilisateurs: SerializedUtilisateurs = JSON.parse(
      JSONbig.stringify(updatedUtilisateur)
    );
    return serializedUtilisateurs;
  } catch (error) {
    console.error("Error updating utilisateurs:", error);
    throw error;
  }
}
```

Mise à jour d'une commande

Pour mettre à jour le statut d'une commande qu'on il y a une validation ou un refus, la fonction utilise le type Partial qui prend tous les attributs d'un type mais en les mettant en "nullable". Si il s'agit d'une validation il crée un mouvements et modifie le stock sinon il change juste le statut de la commande.

```
try {
    const updateData: Partial<commandes> = {};
    if (data.statut !== undefined) {
        updateData.statut = data.statut;
    }
    if (data.id_stock !== undefined) {
        updateData.id_stock = BigInt(data.id_stock);
    }
    if (data.quantite !== undefined) {
        updateData.quantite = BigInt(data.quantite);
    }
    if (data.statut === "validee" && data.id_stock && data.quantite) {
        const stock = await prisma.stocks.findUnique({
            where: { id_stock: BigInt(data.id_stock) },
        });

        if (!stock) {
            throw new Error("Stock non trouvée.");
        }

        if (stock.quantite_disponible === null || stock.quantite_disponible < BigInt(data.quantite)) {
            throw new Error("Quantité non valide.");
        }
```

```
        await prisma.stocks.update({
            where: { id_stock: BigInt(data.id_stock) },
            data: {
                quantite_disponible:
                stock.quantite_disponible - BigInt(data.quantite),
            },
        });

        await prisma.mouvements.create({
            data: {
                id_stock: BigInt(data.id_stock),
                type_mouvement: "sortie",
                quantite: BigInt(data.quantite),
                id_commande: BigInt(data.id_commande),
            },
        });
    }

const updatedCommande = await prisma.commandes.update({
    where: { id_commande: BigInt(data.id_commande) },
    data: updateData,
});
const serializedCommandes: SerializedCommandes = JSON.parse(
    JSONbig.stringify(updatedCommande)
);
return serializedCommandes;
catch (error) {
console.error("Error updating commande:", error);
throw error;
```

Suppression

Cette fonction supprime une commande

```
export async function DeleteCommande(id: number): Promise<boolean> {
  try {
    const commande = await prisma.commandes.findUnique({
      where: { id_commande: BigInt(id) },
    });

    if (!commande) return false;

    await prisma.commandes.delete({
      where: { id_commande: BigInt(id) },
    });

    return true;
  } catch (error) {
    return false;
  }
}
```

Récupération des informations de l'utilisateur

Une fonctionnalité utilisé pour récupérer les informations de l'utilisateur et ainsi la possibilité de donné accessible certaines fonctionnalités admin. Pour ce faire il y a besoin des données mis dans le contexte

```
interface AuthContextValue {
  user: User | null;
  loading: boolean;
  utilisateur: utilisateurs | null;
}

const AuthContext = createContext<AuthContextValue | undefined>(undefined);

export const AuthProvider: React.FC<{ children: React.ReactNode }> = ({
  children,
}) => {
  const [user, setUser] = useState<User | null>(null);
  const [loading, setLoading] = useState(true);
  const [utilisateur, setUtilisateur] = useState<utilisateurs | null>(null);

  useEffect(() => {
    const fetchUser = async () => {
      setLoading(true);
      const currentUser = await getUser();
      setUser(currentUser);
      setLoading(false);
    };
    fetchUser();
  }, []);
```

La fonction getUser permet de récupérer l'utilisateur authentifié

```ts
export const getUser = async (): Promise<User | null> => {
  const {
    data: { user },
  } = await supabase.auth.getUser();

  if (!user) return null;

  return user;
};
```

```ts
  useEffect(() => {
    const fetchUserRole = async () => {
      if (user?.email) {
        try {
          const response = await fetch(`/api/users?email=${user.email}`);
          const data = await response.json();
          setUtilisateur(data);
        } catch (error) {
        }
      }
    };
    if (user) {
      fetchUserRole();
    }
  }, [user]);
  return (
    <AuthContext.Provider value={{ user, loading , utilisateur}}>
      {children}
    </AuthContext.Provider>
  );
};

export const useAuth = (): AuthContextValue => {
  const context = useContext(AuthContext);
  if (context === undefined) {
    throw new Error("useAuth must be used within an AuthProvider");
  }
  return context;
};
```

L'api fait une comparaison du user authentifié avec le premier utilisateur avec lequel l'email est le même. Il y a l'intérieur de l'api une transformation des types des id utilisateur et id rôle.

```ts
const { searchParams } = new URL(request.url);
const email = searchParams.get('email');
if (!email) {
  return NextResponse.json(
    { error: 'Email requis' },
    { status: 400 }
  );
}
const utilisateur = await prisma.utilisateurs.findFirst({
  where: {
    email: email
  }
});
if (!utilisateur) {
  return NextResponse.json(
    { error: 'Utilisateur non trouvé' },
    { status: 404 }
  );
}
const serializedUser = {
  ...utilisateur,
  id_utilisateur: Number(utilisateur.id_utilisateur),
  id_role: Number(utilisateur.id_role)
};
return NextResponse.json(serializedUser);
```

Pour l'utilisation, il y a un exemple dans commandeAdmin

```
const { user, loading, utilisateur } = useAuth();
```

```
const userType = String(utilisateur?.id_role)
```

En sachant que les admin sont dans la base de données un id de valeur 1.

```
{userType !== '1' && (
  <div> Seul les administrateurs peuvent effectuer des actions ici </div>
)}
```

Middleware

Le middleware est appelé à chaque changement d'url, il permet de vérifier si l'utilisateur est bien connecté, si c'est le cas il redirige sur une url qui commence par /dashboard sinon il renvoie sur la page de login

```
const {
  data: { user },
} = await supabase.auth.getUser();

//Si l'utilisateur est connecté et pas sur le dashboard, on le redirige vers /dashboard
if (user && !request.nextUrl.pathname.startsWith("/dashboard")) {
  const url = request.nextUrl.clone();
  url.pathname = "/dashboard";
  return NextResponse.redirect(url);
}

// Si l'utilisateur n'est pas connecté et essaie d'accéder à une route '/dashboard'
if (!user && request.nextUrl.pathname.startsWith("/dashboard")) {
  const url = request.nextUrl.clone();
  url.pathname = "/";
  return NextResponse.redirect(url);
}

return supabaseResponse;
```

07

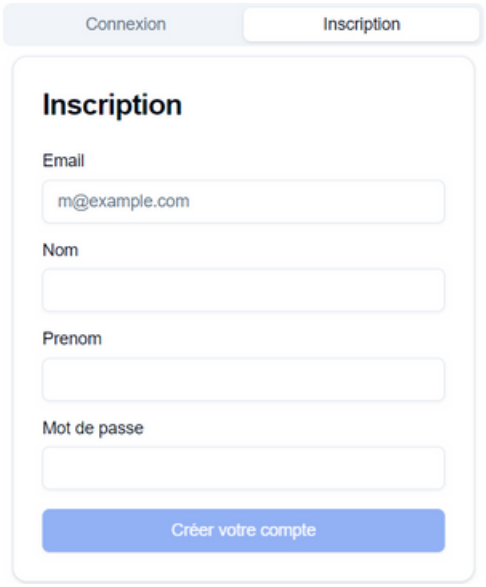
PRÉSENTATION APPLICATION

Présentation de l'application

LOGIN

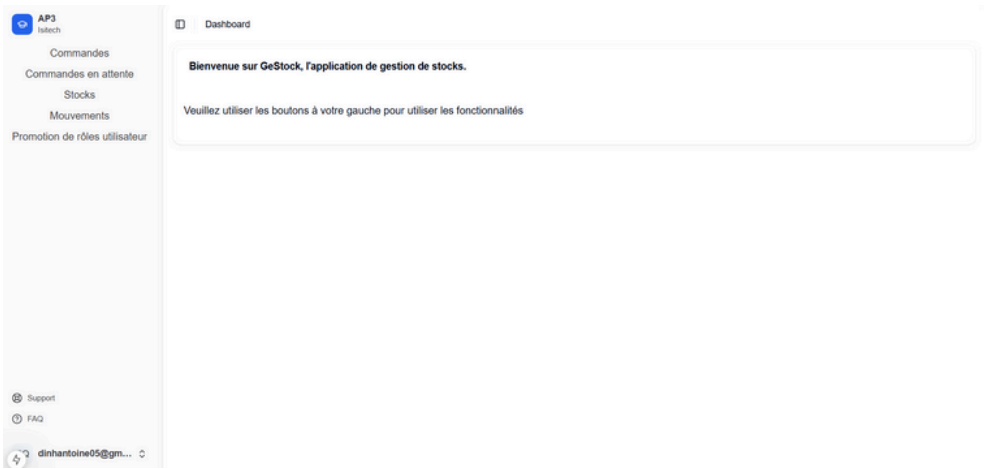
Au lancement de l'application, la page de Login s'affiche avec 2 zones de textes dont 1 en texte cachée et 2 boutons.

Un click sur le bouton Connexion redirigera vers le bonne page si les données sont bonnes, il y de plus une option pour créer un compte avec Inscription.



Dashboard

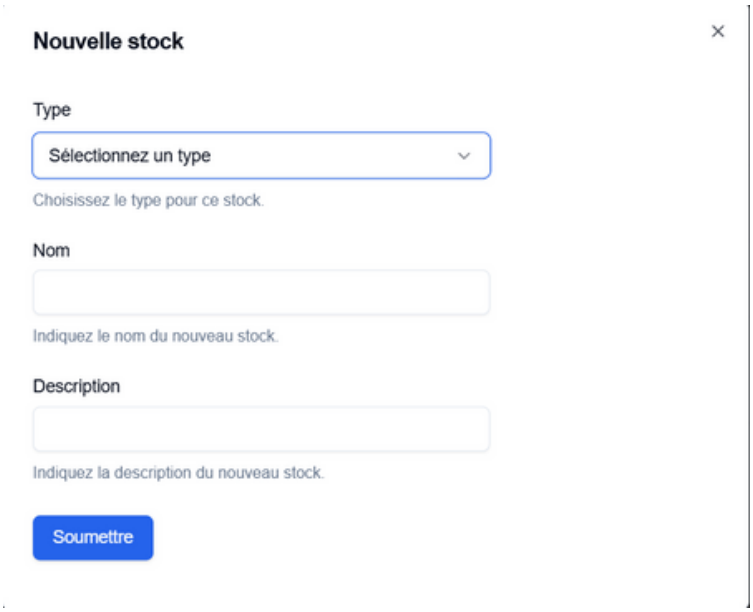
Après être connecté, une page du tableau de bord est affiché avec toutes les fonctionnalités possibles.



Gestion des stocks

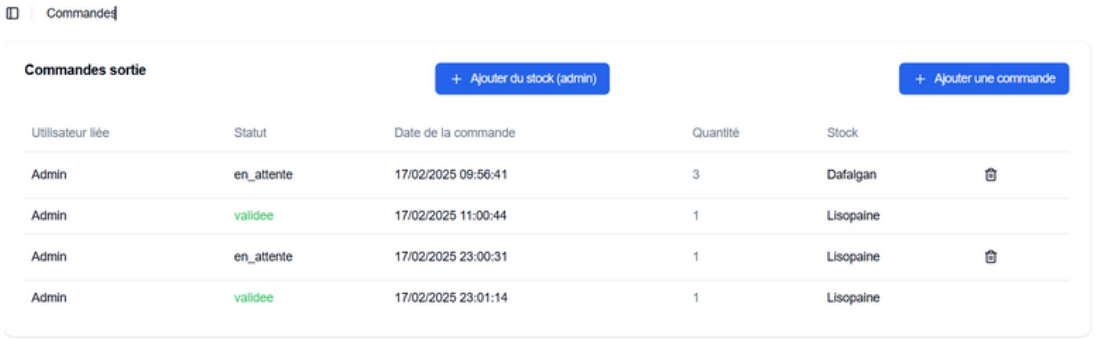
Dans cette fonctionnalité, la liste des stocks avec leur nom, leur description, leur quantité et leur type sont affichés.

Il y a de même si l'utilisateur est un administrateur, la possibilité de rajouter un nouveau stock.

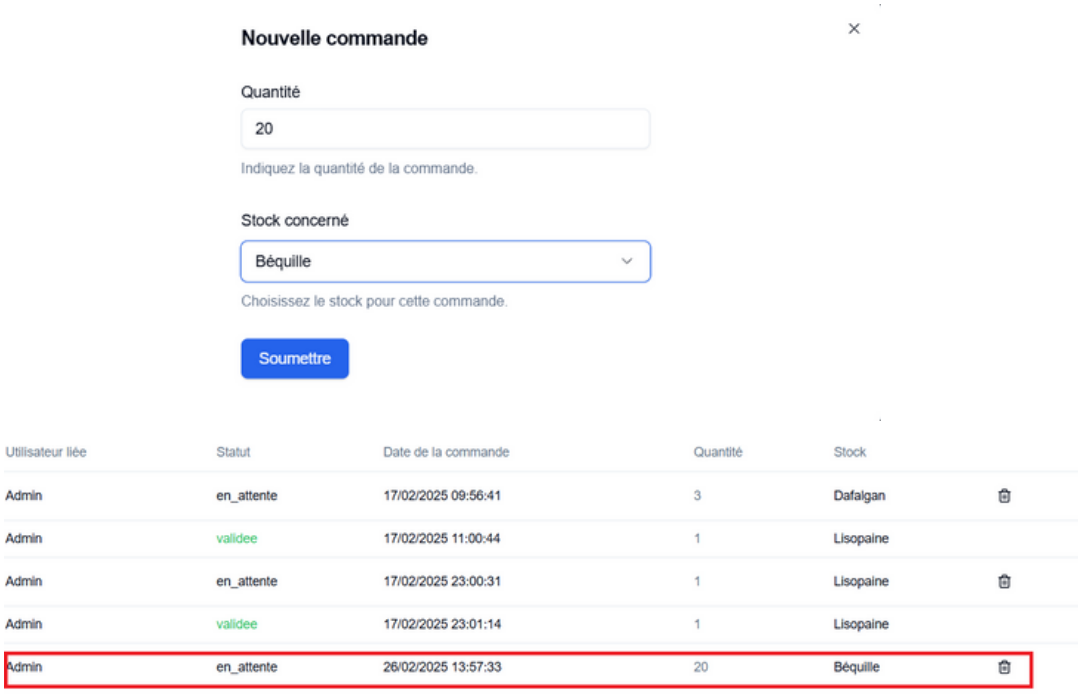


Gestion des commandes

Cette fonctionnalité permet d'afficher toutes les commandes liés à l'utilisateur si des commandes ont le statut en attente il est possible de les supprimer.

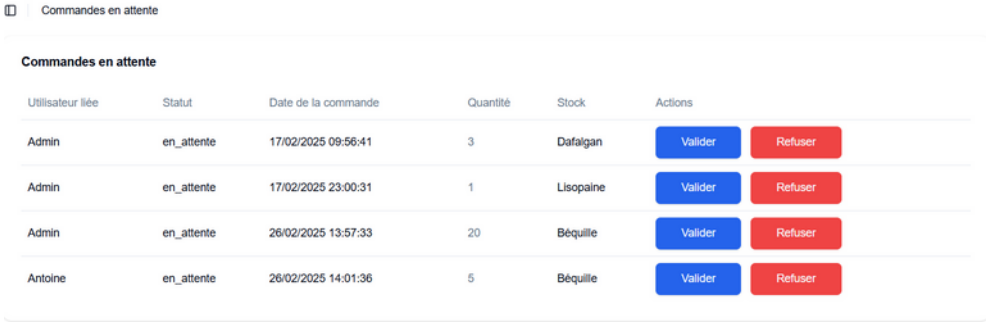


Il y a au dessus de la liste 2 boutons, qui permettent soit avec le bouton de droite de créer une commande "classique", de sortie de stock ou bien une commande admin pour ajouter du stock.

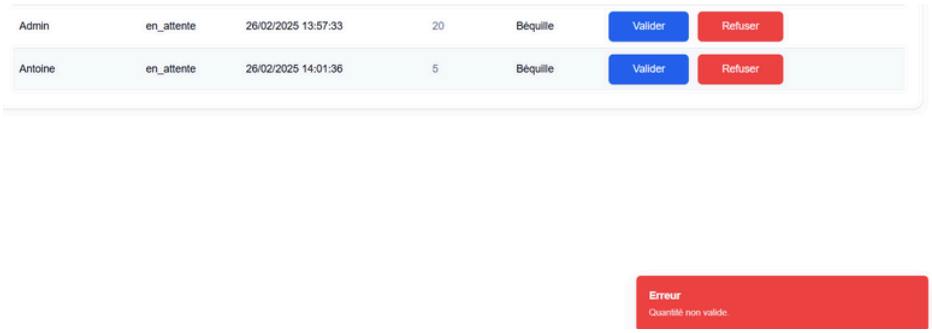


Gestion des commandes en attente

Les administrateurs peuvent accepter ou refuser toutes les commandes

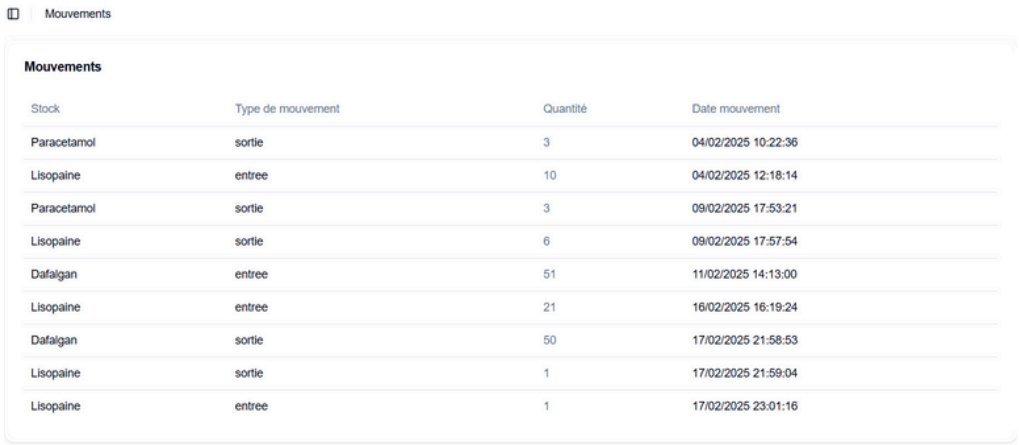


En cas de tentative d'acceptation de commande alors que la quantité disponible dans les stocks n'est pas suffisante, un message d'erreur apparaît.



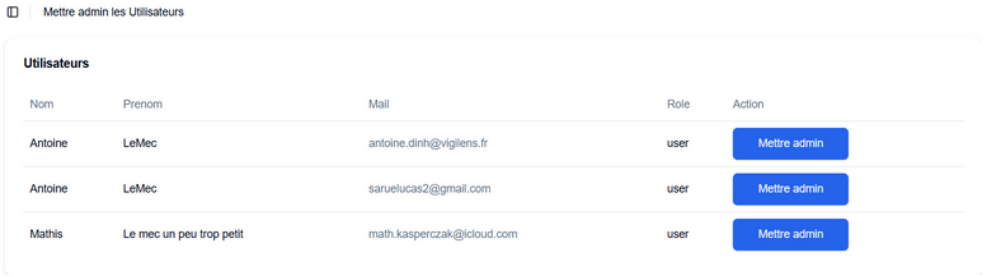
Affichage des mouvements

Quand l'utilisateur clique sur la fonctionnalité mouvement, l'historique de tous les mouvements s'affiche pour chaque mouvement, le stock concerné, le type de mouvement, la quantité et la date du mouvement



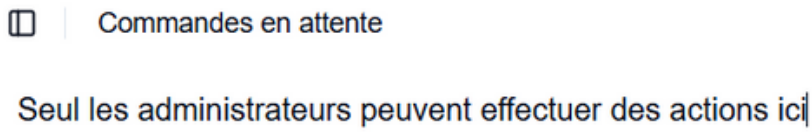
Gestion des droits

Les utilisateurs avec le droit admin peuvent mettre en administrateur les utilisateurs normaux



Blocage selon le rôle

Une vérification du rôle de l'utilisateur est faite lors ce qu'il essaye d'accéder à des fonctionnalités admin. Si il ne l'est pas, un message comme ci-dessous est mis à la place de la fonctionnalité



Lancement de l'application

Aller sur ce lien github :

https://github.com/Goyef/StockManager

Télécharger le zip et l'extraire

Dans l'invite de commande de l'application
faire npm i --force puis npx prisma generate

Le lancement se fait avec npm run dev

Aller sur le localhost:3000

Les identifiants sont
pour l'administrateur identifiant :
dinhantoine05@gmail.com
mot de passe : ouinon

pour l'utilisateur : saruelucas2@gmail.com et
en mot de passe : testTest

08

ANNEXES

GITHUB

Lien github dans l'éventualité où il y a un problème avec le code envoyé :

https://github.com/Goyef/StockManager

Lien vercel dans l'éventualité où il y a un problème avec le code envoyé :

https://stock-manager-gilt.vercel.app

ANTOINE DINH

Février 2025

Isitech

BTS SIO SLAM