

# RAPPORT DE PROJET

## *AP3 BTS SIO*



Projet réalisé dans le cadre de la formation  
BTS SIO SLAM

# Sommaire

<b>1</b>	<b>INTRODUCTION</b> .....	<b>3</b>
1.1	Contexte .....	3
1.1.1	Présentation du projet.....	3
1.1.2	Définitions des besoins et objectifs du projet .....	3
<b>2</b>	<b>CONCEPTION</b> .....	<b>5</b>
2.1	Analyse fonctionnelle et choix techniques.....	5
2.1.1	Fonctionnalités de l'application .....	5
2.1.2	Langages et technologies choisis .....	6
2.2	Explication de la BDD (méthode MCD/MLD).....	8
2.2.1	Explication de la méthode : MCD.....	8
2.2.2	Explication de la méthode : MLD.....	11
2.2.3	Réalisation de la BDD avec Supabase .....	12
2.3	Organisation du code .....	18
2.3.1	Conception des différentes parties du programme.....	18
2.4	Présentation du code sur certaines parties spécifiques.....	21
2.4.1	Créer un nouveau stock.....	21
2.4.2	Retirer une commande en attente.....	24
2.4.3	Mouvements.....	26
2.5	Présentation de l'application.....	28
2.5.1	Créer un nouveau compte .....	28
2.5.2	Ajouter un stock (Administrateur) .....	30
2.5.3	Formuler une commande .....	31
<b>3</b>	<b>CONCLUSION</b> .....	<b>34</b>
3.1	Apprentissages personnels .....	34
3.1.1	Difficultés rencontrées .....	34
3.1.2	Evolution personnelle .....	34
3.2	Annexes.....	36
3.2.1	Annex 1 : Lien du Projet.....	36
3.2.2	Annexe 2 : Git.....	36
3.2.3	Annexe 3 : BDD.....	36

# 1 INTRODUCTION

## 1.1 Contexte

### 1.1.1 Présentation du projet

Dans le monde du travail, il est commun pour une entreprise d'avoir à gérer du « stock », c'est-à-dire un ensemble de marchandises et/ou de matériels disponibles qui sont la propriété de l'entreprise.

Le projet d'Atelier de professionnalisation 3 a pour but de créer une application de gestion de stockage moderne, facile d'utilisation et sécurisée, qui pourrait avoir été demandée par une entreprise. Il s'agit d'une situation vraisemblable car une telle demande d'application est tout à fait plausible.

L'entreprise dite « GSB » est un laboratoire pharmaceutique, issu de la fusion entre le géant américain Galaxy (spécialisé dans le secteur des maladies virales dont le SIDA et les hépatites) et le conglomérat européen Swiss Bourdin (travaillant sur des médicaments plus conventionnels).

Durant la réalisation de ce projet, je suis parti du principe que l'entreprise GSB serait l'entreprise ayant passé commande, intéressée par une application capable de gérer son stock de matériels et de médicaments.

### 1.1.2 Définitions des besoins et objectifs du projet

L'entreprise GSB souhaite mettre en place une application web pour gérer son stockage.

Une application web est un logiciel accessible via un navigateur, qui utilise une connexion réseau pour permettre aux utilisateurs d'interagir avec des fonctionnalités ou des données hébergées sur un serveur distant.

L'objectif est donc de créer une telle application capable de répondre aux différents besoins spécifiques au sein de l'entreprise : ici, héberger le stock de l'entreprise sous la forme d'une base de données, et permettre de créer et de gérer des commandes pour interagir avec cette base.

On cherche également à différencier les utilisateurs entre eux avec un rôle de visiteur pour ceux qui ont une utilisation classique de l'application, et un rôle d'administrateur pour les utilisateurs de confiance capables d'opérer plus d'actions depuis l'interface de l'application.

Pour s'assurer de la qualité finale de l'application achevée, on définit quelques consignes en avance afin de coller aux besoins de l'entreprise :

L'application doit soutenir deux principaux types d'utilisateurs : les utilisateurs standards (par exemple, le personnel soignant ou administratif) et les administrateurs (par exemple, les responsables de pharmacie ou les gestionnaires de stock).

Consultation des stocks : Tous les utilisateurs doivent pouvoir consulter la liste des médicaments et du matériel disponible. Cette consultation inclurait 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.

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.

Gestion des commandes : Les utilisateurs administrateurs ont la capacité de 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).

Ajout de stock : Les administrateurs peuvent également 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.

Pour que l'utilisation de l'application reste claire, chaque action qu'il est possible d'effectuer est séparée de manière distincte dans un composant attribué.

## 2 CONCEPTION

### 2.1 Analyse fonctionnelle et choix techniques

Avant de créer une application, il est essentiel de savoir quelles technologies seront nécessaires ou avantageuses dans la conception du service.

#### 2.1.1 Fonctionnalités de l'application

Les fonctionnalités sont des caractéristiques ou des capacités spécifiques à l'application qu'elle peut mettre à disposition des utilisateurs pour répondre à leurs besoins ou pour résoudre des problèmes.

Je me suis d'abord consacré au développement des fonctionnalités principales.

Je me suis donc concentré dans un premier temps sur les fonctionnalités suivantes :

Page d'authentification : à partir de cette page, un utilisateur peut soit se connecter, soit créer un compte à l'aide d'une adresse e-mail. Cette adresse ne permet à l'utilisateur de se connecter par la suite que si elle a été préalablement vérifiée.

Page d'accueil : à partir de cette page, toutes les autres actions sont accessibles.

Espace du Stock : à partir de cet espace, le stock peut être consulté. Seuls les administrateurs peuvent ajouter de la quantité au stockage.

Espace des Commandes : à partir de cet espace, n'importe quel utilisateur peut consulter les commandes existantes et en créer une nouvelle.

Espace des Commandes en attente : à partir de cet espace, seuls les utilisateurs disposant de privilèges d'administrateur peuvent consulter les commandes en attente pour y effectuer des actions (Valider ou Refuser).

## 2.1.2 Langages et technologies choisis

Pour la conception de mon application, j'ai intégré plusieurs technologies et langages de programmation, afin de garantir une solution robuste et performante.

Chaque technologie a été choisie en fonction de ses capacités à répondre à certains besoins du projet et à optimiser son développement, sa maintenance et l'expérience utilisateur :

Le langage de programmation TypeScript (ou TS), est un langage typé et orienté objet qui, associé à Next.js, offre une robustesse et une sécurité accrues lors du développement. Sa forte typisation permet de détecter les erreurs dès la phase de compilation, ce qui facilite la maintenance et l'évolution de mon code. Dans cette application, TypeScript est utilisé pour l'ensemble de la logique, que ce soit pour la gestion des API côté serveur ou pour l'implémentation de la partie visible de l'application via React.

Le framework Next.js, construit sur React, est spécialement conçu pour créer des applications web performantes et optimisées. Next.js intègre également un système de routage avancé et la gestion des API, ce qui simplifie la structure globale de l'application et accélère le processus de développement.

Le langage de balisage TSX (TypeScript XML) est une extension de JSX qui intègre la puissance de TypeScript, permettant ainsi de créer des interfaces utilisateur dynamiques et réactives avec un typage strict. Il facilite l'écriture de composants en mélangeant du code TypeScript et des éléments HTML, tout en offrant une vérification rigoureuse des types à la compilation. Cette caractéristique améliore considérablement la robustesse du code en réduisant les erreurs potentielles et en rendant le développement plus fiable et maintenable.

Dans mon application, le TSX joue un rôle central dans la conception du front-end, assurant une structuration claire et efficace des composants de l'interface utilisateur. Grâce à cette technologie, il est possible de manipuler facilement l'état et les propriétés des composants tout en conservant une parfaite cohérence avec la logique métier définie dans le back-end (partie invisible à l'utilisateur, qui concentre la logique du programme). Cette synergie entre TSX, TypeScript et Next.js garantit une interaction fluide entre l'interface graphique et les fonctionnalités de l'application, contribuant ainsi à une expérience utilisateur optimisée et performante.

Enfin, j'ai structuré l'application Next.js en suivant l'architecture recommandée par ce framework, qui repose sur une séparation claire des responsabilités pour améliorer la lisibilité et la maintenabilité de l'application. Cette organisation se divise en plusieurs parties distinctes :

Les pages et composants représentent l'interface utilisateur et sont chargés d'afficher les éléments visibles de l'application. Les pages de 'Next.js' se trouvent dans le dossier 'app', et chaque fichier y correspond à une route spécifique. Les composants permettent de réutiliser des éléments de l'interface et sont généralement situés dans un dossier 'components' pour assurer une meilleure modularité.

Les routes API servent à gérer le traitement des requêtes côté serveur. Stockées dans le dossier 'api', elles permettent d'exécuter du code backend, d'interagir avec la base de données et de traiter les demandes des utilisateurs sans exposer la logique métier au client.

Le modèle de donnée est utilisé pour structurer et gérer les informations stockées dans la base de données. Avec Prisma ORM comme connexion directe à Supabase, il définit la structure des entités et facilite la communication entre le frontend (partie visible à l'utilisateur, qui concentre la totalité de l'interface graphique) et les données stockées.

Grâce à cette organisation, l'ajout de nouvelles fonctionnalités se fait de manière fluide, sans altérer le reste de l'application, ce qui garantit un développement évolutif et performant.

## 2.2 Explication de la BDD (méthode MCD/MLD)

Avant de commencer le développement de mon application, je sais qu'une Base De Données (BDD) est indispensable. Je me suis donc dans un premier temps concentré sur cette tâche.

Définir la structure de la BDD est un élément central du projet : il faut s'assurer que le modèle est impeccable, car la moindre erreur commise dans la création de la base de données pourrait avoir des répercussions bien plus tard dans le développement de l'application.

Pour assurer une bonne conception de la base de données, je crée d'abord le modèle de base en suivant la méthode MCD.

### 2.2.1 Explication de la méthode : MCD

La méthode MCD est une façon de représenter les données d'une organisation ou d'un système sans se préoccuper de leur stockage précis. Ce serait comme faire un plan simplifié des informations importantes et de la manière dont elles sont connectées.

Le principe de base est simple, il existe :

- les Entités : ce sont des choses ou des concepts dans le monde réel, comme un Client ou un Produit.
- les Relations : ce sont les liens entre ces entités ; par exemple, une entité Client possède la relation « acheter » avec l'entité Produits.
- les Attributs : ce sont les détails ou caractéristiques des entités. Par exemple, une entité Client possède comme attributs un nom et une adresse.
- et les Cardinalités : elles spécifient le nombre d'occurrences d'une entité, qui peut être associée à un nombre donné d'occurrences d'une autre entité dans une relation.

On note trois symboles différents : 0, 1 et N, chaque symbole est associé à un autre pour former une paire qui illustre la cardinalité d'une relation.

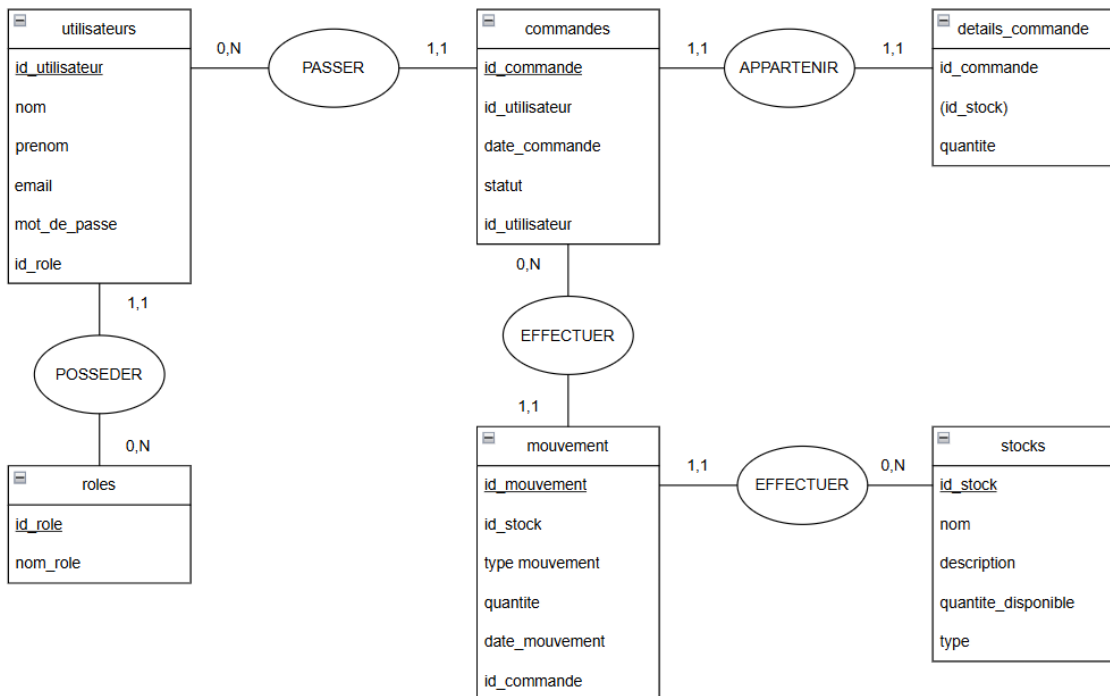
Quelques exemples :

- Un employé doit avoir au moins un contrat de travail, mais peut en avoir plusieurs, donc la cardinalité de la relation EMPLOYÉ vers CONTRAT est (1, N).
- Un contrat peut ne pas être lié à un employé, ou être associé à plusieurs, donc la cardinalité de CONTRAT vers EMPLOYÉ est (0, N).



- La cardinalité 0,1 signifie qu'une voiture peut ne pas avoir de propriétaire, mais si elle en a un, il n'y en aura qu'un seul.
- La cardinalité 1,1 signifie qu'un arbre a toujours un tronc.

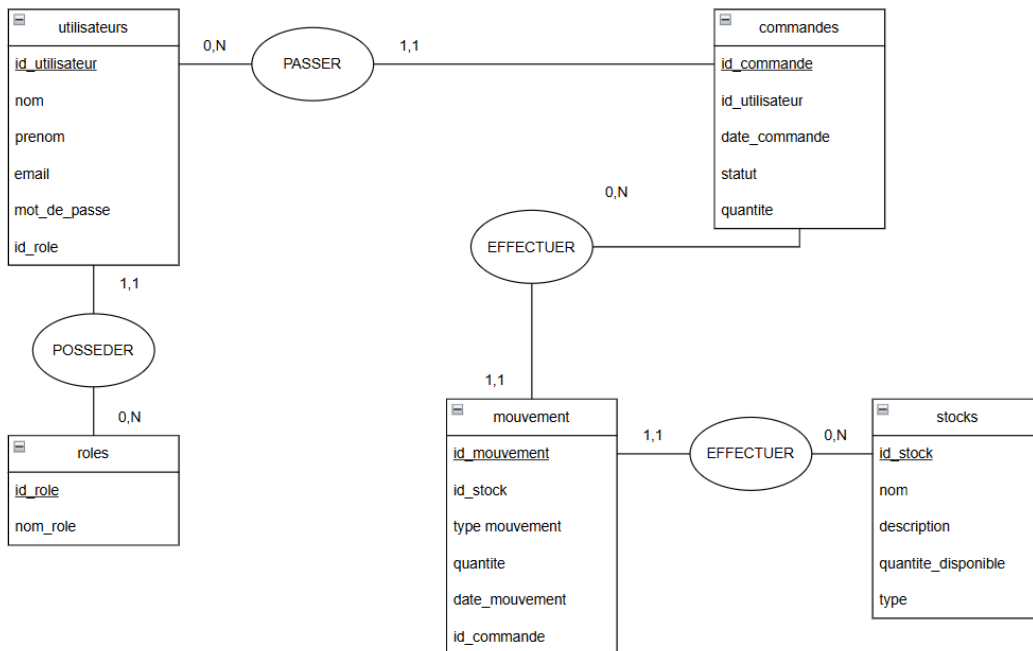
Dans notre cas, cela se traduit ainsi :



*Première version du MCD pour la BDD de l'application. Le MCD final a évolué une fois lors du développement de l'application.*

**SECONDE VERSION**

MCD\_AP3\_versionFinale.drawio  
19,1 Ko • Il y a 56 minutes



*Seconde version du MCD pour la BDD de l'application. Lors de la conception de l'application, le table « details\_commande » s'avérait ajouter de la complexité inutile et a donc fusionné avec la table « commandes ».*

Une fois chaque Entité, Attribut et Relation définis graphiquement, cela permet d'ajuster les détails techniques plus tard sans changer tout le plan initial ; en résumé, cette méthode simplifie l'organisation des informations importantes d'un système en les représentant visuellement et en aidant à préparer le terrain pour la création d'une base de données.

Simplifier permet de rendre les choses plus efficaces et plus faciles à développer et à gérer.

Avec un schéma MCD complet et correct, la dernière étape avant la création concrète de la Base De Données est de convertir le dit schéma à l'aide de la méthode MLD.

## 2.2.2 Explication de la méthode : MLD

La méthode MLD (Modèle Logique de Données) est une étape dans la conception de bases de données qui suit celle du MCD (Modèle Conceptuel de Données).

Alors que le MCD se concentre sur la représentation abstraite et conceptuelle des données et de leurs relations, le MLD vise à traduire ce modèle conceptuel en une structure plus concrète et technique, prête à être implémentée dans une base de données.

La conversion d'un schéma MCD en Modèle Logique de Données donne ceci :

UTILISATEURS: (id\_utilisateur, nom, prenom, email, mot\_de\_passe, #id\_role)

ROLES: (id\_role, nom\_role)

MOUVEMENTS: (id\_mouvement, type\_mouvement, quantite, date\_mouvement, #id\_commande, #id\_stocks)

STOCKS: (id\_stocks, nom, description, quantite\_disponible, type)

COMMANDES: (id\_commande, date\_commande, statut, quantite, #id\_utilisateur)

Pour obtenir ce résultat, il faut suivre quelques étapes simples :

- À partir du MCD existant, traduire chaque entité en une table, chacun de ses attribut en une colonne et chaque relation en une contrainte de clé étrangère.
- Définir les clés primaires pour chaque table : les clé primaires identifient de manière unique chaque table.
- Définir les clés étrangères pour établir les relations entre les tables. Une clé étrangère fait référence à la clé primaire d'une autre table ; elle établit ainsi une relation entre les deux tables, permettant de lier les données de manière cohérente.

Chaque clé est soulignée, pour différencier la clé primaire des clés étrangères. Les clés étrangères commencent toutes par un dièse (#) : ainsi dans le MLD obtenu plus haut, on remarque que la table ETAT est liée à la table FICHE\_DE\_FRAIS car cette dernière possède comme clé étrangère la clé primaire de la table ETAT « id\_etat ».

Une fois la conversion achevée et le résultat MLD obtenu, la réalisation de la Base De Données peut débuter ; pour ce faire, j'ai utilisé l'interface web de Supabase.

## 2.2.3 Réalisation de la BDD avec Supabase

Supabase est une plateforme open-source moderne qui facilite la gestion de bases de données PostgreSQL, le type de base que j'utilise dans ce projet. C'est un outil complet, apprécié pour sa simplicité d'utilisation et ses nombreuses fonctionnalités :

- Il propose une interface visuelle intuitive pour créer, modifier ou supprimer des bases de données, ainsi que gérer leurs structures, tables, colonnes, clés primaires et clés étrangères.
- Il offre une visualisation claire des tables et des données, facilitant la navigation entre les enregistrements et la compréhension des relations entre les tables.

En temps normal, il permet également d'exécuter des requêtes SQL pour manipuler les données directement, or dans ce cas précis j'utilise Prisma.

Prisma est un ORM (Object-Relational Mapping) moderne pour Node.js et TypeScript, qui facilite la gestion des bases de données dans les projets tel que Next.js. Il agit comme un pont entre l'application et la base de données en simplifiant les requêtes SQL qui pourrait être complexes en un code TypeScript plus facilement lisible et sécurisé.

Cet outil est particulièrement pratique, car il accélère le développement de mon application en permettant une gestion efficace de la base de données tout en étant dans une logique orientée objet.

Voici à quoi ressemble Prisma dans le projet : une fois intégré à mon application, l'ORM crée une table par *modèle* trouvé dans l'espace de travail de mon éditeur de code, puis, sans rentrer dans les détails, dans chaque table créée sont configurés les attributs et les relations entre ces entités.

```
model commandes {
  id_commande    BigInt          @id @default(autoincrement())
  id_utilisateur BigInt
  date_commande  DateTime         @default(now()) @db.Timestamp(6)
  statut         statut_commande @default(en_attente)
  id_stock       BigInt
  quantite       BigInt
  stocks         stocks           @relation(fields: [id_stock], references: [id_stock], onDelete: SetNull)
  utilisateurs   utilisateurs     @relation(fields: [id_utilisateur], references: [id_utilisateur], onDelete: SetNull)
  mouvements     mouvements[]
}

model mouvements {
  id_mouvement    BigInt          @id @default(autoincrement())
  id_stock        BigInt
  type_mouvement  type_mouvement_enum
  quantite        BigInt
  date_mouvement  DateTime         @default(now()) @db.Timestamp(6)
  id_commande     BigInt
  commandes       commandes       @relation(fields: [id_commande], references: [id_commande], onDelete: SetNull)
  stocks          stocks          @relation(fields: [id_stock], references: [id_stock], onDelete: SetNull)
}

model roles {
  id_role        BigInt          @id @default(autoincrement())
  nom_role       String          @db.VarChar(50)
  utilisateurs   utilisateurs[]
}

model stocks {
```

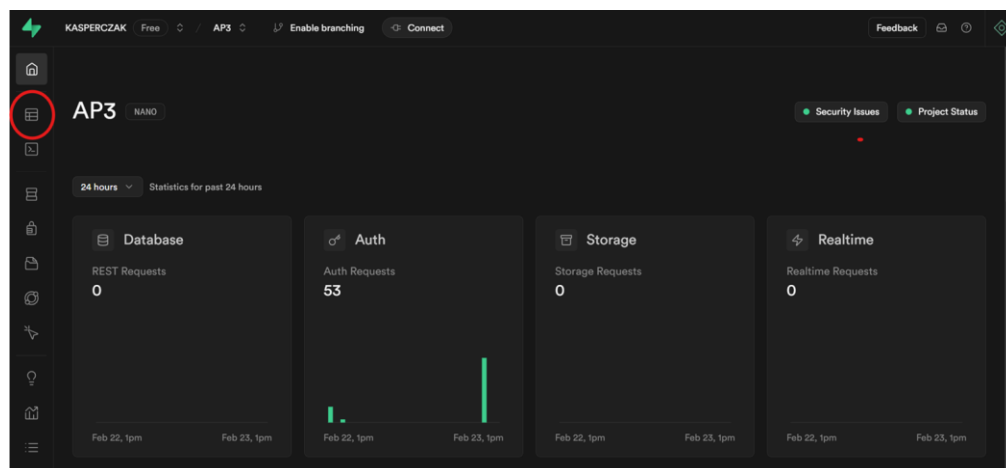
(explication : une commande, si elle est validée, provoque la création d'un mouvement, en l'occurrence un mouvement ne peut être lié qu'à une seule commande à la fois. ).

De même, la deuxième ligne permet de préciser qu'un élément de la table « mouvement » peut avoir une relation avec des éléments de la table « stocks » (le mouvement indique sur quel stock il y a eu un ajout ou un débit de quantités.)

Cette table contiendra donc des colonnes correspondant aux clés primaires des deux entités, permettant de les relier.

La relation est ensuite matérialisée sur Supabase.

Pour utiliser Supabase, je me suis créé un compte depuis son interface web, qui me permet de me connecter à mon espace de données, à savoir un serveur où est matérialisée ma base de données.



Interface Supabase

*une fois connecté  
(Capture 1)*

Une fois connecté, l'interface web de Supabase apparaît comme illustré ci-dessous. Seuls quelques éléments de l'interface sont nécessaires pour comprendre l'intégration de la base de données dans l'application. Afin de consulter les tables de la base de données, il faut cliquer sur l'icône présent dans le cercle rouge de la Capture d'écran ci-dessus (« Table Editor »).

	id_mouvement	id_stock	type_mouvement	type_mouvement_enum	quantite	date_mouvement
1	3	→	sortie		12	2025-02-13 19:16:31.541
2	4	→	sortie		5	2025-02-14 14:09:19.394
3	4	→	sortie		2	2025-02-14 14:09:55.013
4	3	→	sortie		1	2025-02-14 11:12:42.922
5	3	→	sortie		20	2025-02-14 14:42:21.524
6	3	→	sortie		1	2025-02-14 18:22:24.809
7	3	→	sortie		1	2025-02-15 10:28:10.278
8	3	→	sortie		22	2025-02-15 14:46:29.096
9	3	→	sortie		20	2025-02-15 14:19:26.659
10	4	→	sortie		1	2025-02-16 15:45:57.786
11	6	→	sortie		50	2025-02-16 15:50:59.136
12	6	→	sortie		50	2025-02-16 16:05:00.992

Capture 2

Dans cet exemple, nous examinons en détail la table « mouvement », dont le rôle principal est de stocker les informations de tous les mouvements enregistrés dans le système (un mouvement représente un ajout ou un retrait de stock).

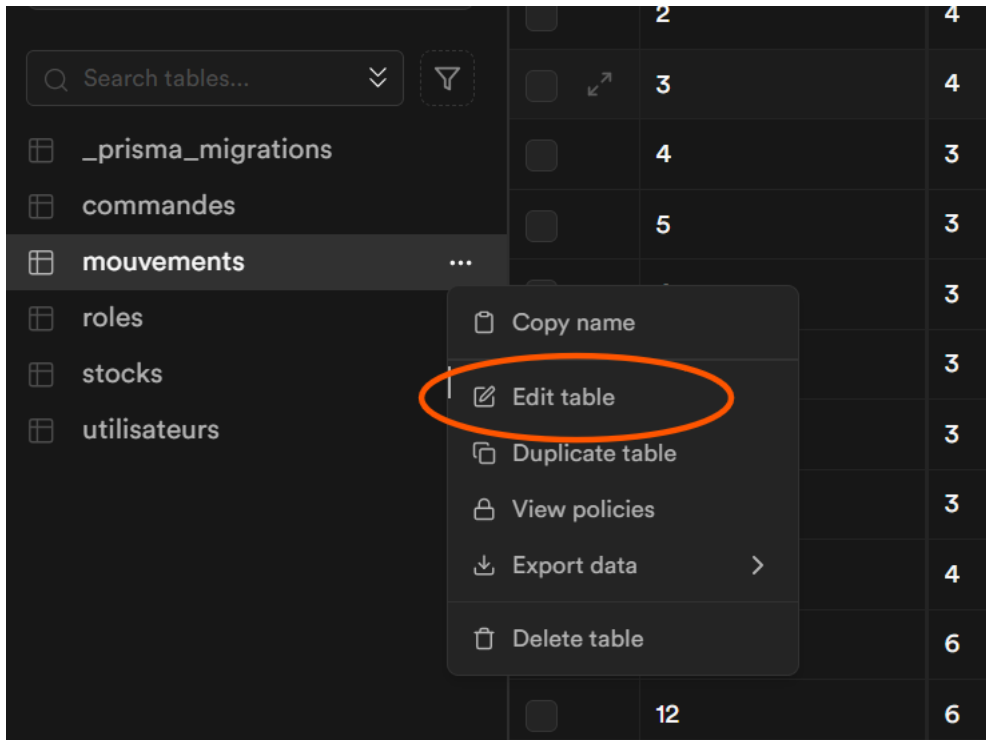
Le tableau affiche, au centre de l'écran, les différents attributs définis pour chaque « mouvements ».

- La clé primaire de la table : Dans le cercle rouge de *Capture 2*, on retrouve la colonne `id_mouvement` qui contient les identifiants uniques de chaque membre de la table.
- Une clé étrangère de la table : Dans le cercle vert de *Capture 2*, on retrouve la colonne `id_stock` qui lie chaque membres de la table avec un membres de la table « stock ».
- Les attributs de la table : Dans le cercle bleu de *Capture 2*, on retrouve tous les attributs que comporte la table, chacun possédant un type (Chaîne de caractère, nombre entier). Parmi eux, nous trouvons par exemple l'attribut « `type_mouvement` » (si le mouvement est une entrée de stock ou une sortie), « `quantite` » (la quantité de stock déplacement), et « `date_mouvement` » (quand le mouvement a été créé).

Chaque enregistrement dans la table « mouvement » comporte ainsi un identifiant unique, un type de mouvement, une quantité, une date de mouvement, ainsi que l'identifiant d'un stock.

Cette structure permet de gérer efficacement les informations des mouvements, assurant que chaque élément de la table peut être identifié et authentifié grâce à ses attributs.

Dans le cercle jaune de la *Capture 2* ci-dessus se trouve le bouton qui permet d'accéder à la structure de la table détaillée. La structure de la table détaillée est représentée dans la *Capture 3* ci-dessous.



Le

*Capture 3*

bouton « Edit table » (*Capture 3*) permet d'interagir de façon plus lisible avec les attributs de la table une fois sélectionné.

Name	Type	Default Value	Primary
id_mouvement	# int8	NULL	<input checked="" type="checkbox"/>
id_stock	int8	NULL	<input type="checkbox"/>
type_mouvement	type_mou	NULL	<input type="checkbox"/>
quantite	# int8	NULL	<input type="checkbox"/>
date_mouvement	timestamp	CURRENT_TIMESTAMP	<input type="checkbox"/>
id_commande	int8	NULL	<input type="checkbox"/>

Capture 4

Le détail de la structure de la table permet de visualiser les types de chaque attribut de la table (id\_mouvement est un int8, ce qui signifie un nombre entier qui ne peut dépasser 8 chiffres; id\_stock est un int8 également, et date\_mouvement est un timestamp, ce qui signifie une chaîne de caractères qui prend la forme d'une date (année - mois - jour) ).

Supabase met en évidence la clé primaire de la table, à l'aide d'une case « primary » à côté de l'attribut concerné. Ici la clé primaire de la table « mouvement » est bien « id\_mouvement ».

Supabase met également en évidence la (ou les) clé(s) étrangère(s) de la table, qui sont définie(s) plus bas dans cet espace. Ici l'attribut « id\_stock » et « id\_commande » sont bien des clés étrangères (Capture 4).





*Capture 4*

Si besoin, chaque élément peut être modifié facilement dans cet espace.

## 2.3 Organisation du code

### 2.3.1 Conception des différentes parties du programme

Mon application est découpée en plusieurs parties, chacune soutenant des fonctions bien spécifiques au sein de mon application, plus ou moins critiques. On peut en retirer quatre parties essentielles, chacune séparée par des dossiers :

- Le dossier Api (app/api) :

Ce dossier est responsable de la gestion des requêtes et des réponses entre le frontend et le backend. Il contient des fichiers qui définissent des routes d'API permettant d'effectuer des opérations comme la récupération, l'ajout, la modification ou la suppression de données.

L'API sert d'intermédiaire entre l'interface utilisateur et la base de données, garantissant ainsi la sécurité et l'efficacité des échanges de données. Plutôt que de permettre aux pages du front d'accéder directement à la base de données, l'API offre une couche de protection supplémentaire, où l'on peut définir des règles de validation et des contrôles d'accès. Elle permet surtout d'améliorer la scalabilité du projet : en centralisant la gestion des requêtes, on peut facilement modifier ou optimiser une fonctionnalité sans impacter directement l'affichage côté utilisateur.

- Le dossier Dashboard (app/dashboard) :

Le dossier Dashboard regroupe toutes les pages visibles par l'utilisateur. Il contient les fichiers TSX, qui structurent les vues et permettent d'afficher dynamiquement des informations récupérées via l'API, son rôle principal est de fournir les services attendus d'un frontend fluide et intuitif, en affichant des tableaux de bord interactifs et des formulaires qui permettent aux utilisateurs d'interagir avec l'application. Il gère aussi la navigation entre les différentes sections, en intégrant souvent des composants réutilisables.

Grâce à ce dossier, je peux organiser le contenu de manière logique et optimiser la gestion de l'interface, en s'assurant que chaque page affiche uniquement les données nécessaires.

- Le dossier Services (services) :

Ce dossier regroupe l'ensemble des fonctions et méthodes métiers qui sont utilisées dans plusieurs parties de mon application.

L'objectif du dossier *Services* est d'éviter la redondance du code en centralisant certaines fonctionnalités dans des fichiers accessibles depuis plusieurs endroits. Par exemple, si plusieurs composants doivent récupérer les informations d'un utilisateur, au lieu de dupliquer la logique à plusieurs endroits, on crée une seule fonction dans *Services* et on l'appelle où nécessaire.

Cela facilite également la maintenabilité de l'application : en cas de modification d'une fonctionnalité, il suffit d'éditer le fichier correspondant dans *Services*, sans impacter directement le reste du projet. En somme, ce dossier permet de rendre l'architecture du projet plus lisible et potentiellement évolutive.

- Le dossier Prisma (prisma) :

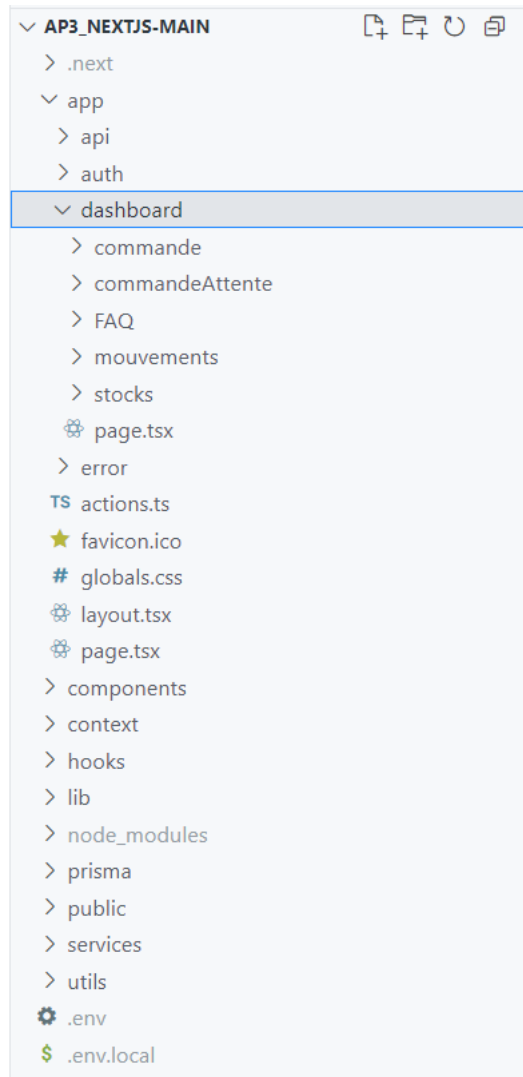
Comme vu précédemment, le dossier *prisma* contient la relation entre le code et la base de données, ce dossier est en charge de la gestion et de la connexion avec la base de données. Il contient le fichier de configuration de Prisma qui permet d'interagir avec la base de données à l'aide d'un langage plus intuitif que le SQL brut.

Pour résumer, on peut séparer la composition de l'application en deux éléments :

Le FRONTEND : Programmé en TSX géré par des composants d'affichages graphiques.

Le BACKEND : Les méthodes programmées en TS, qui effectuent des opérations sur les données au sein de l'application.

Toute mon application est divisée par dossiers et sous-dossiers. Ainsi mon dossier *Dashboard* contient toutes les pages de mon application, de même que mon dossier *Api* et mon dossier *Services* contiennent respectivement tous mes modèles d'api et tout mes services. A préciser que mon dossier *Views* est lui même composé d'autant de sous dossiers qu'il y a de contrôleurs. Chaque sous-dossier possédant les pages Vue spécifiques de chaque contrôleurs.



*Arborescence organisée par dossiers et sous dossiers*

## 2.4 Présentation du code sur certaines parties spécifiques

La plupart des fonctionnalités de mon application suivent une logique claire basée sur une structure « affichage > API > services ». D'abord, l'interface utilisateur affiche les données dynamiquement en récupérant les informations nécessaires via l'API. Cette API sert d'intermédiaire et gère les échanges entre le front et les services métier, qui contiennent la logique applicative et interagissent avec la base de données.

Lorsqu'un utilisateur effectue une action (comme soumettre un formulaire ou modifier un élément), l'API traite la requête et appelle les services adéquats pour valider, enregistrer ou mettre à jour les informations. Une fois l'opération effectuée, la réponse est renvoyée au front pour actualiser l'affichage en conséquence.

Cette structure assure une séparation claire des responsabilités, garantissant une meilleure maintenabilité, une scalabilité optimisée et une réactivité fluide entre la base de données, l'API et l'interface utilisateur.

Il suffit de quelques exemples pour comprendre la logique globale de mon application, en voici quelques-uns :

### 2.4.1 Créer un nouveau stock

L'espace dédié aux stocks est crucial dans une application visant à créer des commandes, car il constitue le point central de gestion des informations nécessaires pour adapter chaque commande aux besoins spécifiques de chaque utilisateur.

Sans cet espace, il serait impossible d'associer correctement du matériel aux besoins spécifiques de chaque utilisateur, ce qui rendrait les commandes obsolètes. Il permet aussi d'accéder facilement aux informations du stock disponible.

Cet espace est donc organisé en plusieurs fonctionnalités, comme l'ajout d'un stock, et la modification de sa quantité. Ici, je vais prendre l'exemple de la fonctionnalité d'ajout d'un stock. Cette logique d'ajout est similaire à celle utilisée pour d'autres espaces, comme celui des commandes par exemple, leur fonctionnement est presque identique.

```
app > api > stocks > TS route.ts > ...
13 export async function POST(req: NextRequest) {
14   try {
15     const body = await req.json();
16     const newStock = await CreateStocks({
17       nom: body.nom,
18       description: body.description,
19       type: body.type,
20     });
21     return NextResponse.json(newStock, { status: 201 });
22   } catch (error) {
23     console.error("Error creating stock:", error);
24     return NextResponse.json(
25       { error: "Failed to create stock" },
26       { status: 500 }
27     );
28   }
29 }
30 |
```

*Capture 1*

La fonctionnalité d'ajout repose sur deux espaces : une méthode dans l'api (*Capture 1*) et une autre dans les services (*Capture 2*).

Le premier élément du code est la déclaration d'une fonction asynchrone POST qui prend en paramètre un objet « req ». Cette fonction est de type POST, c'est-à-dire ici qui a pour but de permettre l'ajout d'un nouvel élément dans une base de données (rectangle Jaune *Capture 1*).

Dans ce bloc, la fonction commence par extraire le corps de la requête (`await req.json()`). Cette opération permet de récupérer les données envoyées par le client sous forme de JSON (Format commun pour manipuler des données entre un serveur et un client). Ensuite, la fonction crée un nouvel objet « stock » avec plusieurs propriétés (nom, description et type), qui sont directement extraites du corps de la requête (`body.nom`, `body.description`, `body.type`) ; pour créer cet objet, la méthode du service « `CreateStocks` » est appelée (Voir *Capture 2*).

Si tout se déroule correctement, la fonction renvoie une réponse HTTP au client, réponse qui indique que la requête a réussi et qu'une nouvelle ressource a bien été ajoutée à la base de données (rectangle bleu *Capture 1*).

Si, au contraire, un problème est rencontré, le rectangle rouge contient les lignes de la gestion d'erreur, qui permet à l'application de ne pas planter en cas de problème. Si une erreur survient à n'importe quelle étape du processus (par exemple, si la base de données est inaccessible ou si les données envoyées sont incorrectes), le programme exécute immédiatement le bloc `catch(error)` qui renvoie l'erreur dans la console du serveur.

```
services > TS stockService.ts > CreateStocks
31 export async function CreateStocks(data: {
32     nom: string;
33     description: string;
34     type: $Enums.stock_type;
35 }): Promise<SerializedStocks | null> {
36     try {
37         const stock = await prisma.stocks.create({
38             data: {
39                 nom: data.nom,
40                 description: data.description,
41                 quantite_disponible: parseInt("0", 10),
42                 type: data.type,
43             },
44         });
45         const serializedStocks: SerializedStocks = JSON.parse(
46             JSONbig.stringify(stock)
47         );
48         return serializedStocks;
49     } catch (error) {
50         console.error("Erreur lors de la création de la commande:", error);
51         throw new Error("Echec de la création de la commande");
52     }
53 }
```

Capture 2

La fonction `CreateStocks` prend un paramètre `data`, qui est un objet contenant trois propriétés : `nom`, `description` et `type`, qui définissent les caractéristiques du stock que l'on souhaite créer. (rectangle vert *Capture 2*)

Une fois les données reçues en paramètre, la fonction tente de créer un nouvel enregistrement dans la base de données en utilisant la méthode `prisma.stocks.create`. Les données insérées dans cette table sont extraites de l'objet `data` (`nom`, `description`, `type`), et la quantité disponible est définie à une valeur de base qui est 0 (rectangle bleu *Capture 2*).

Après que le nouvel élément a été ajouté à la base de données, Prisma retourne l'objet `stock`, contenant toutes les informations du stock fraîchement créé. Toutefois, avant de renvoyer cet objet à l'appelant de la fonction, une étape de sérialisation est effectuée. Cette étape permet de transformer l'objet `stock` en une version standardisée en JSON, le but principal est de s'assurer que les données renvoyées sont dans un format propre et exploitable par le reste de l'application, notamment pour éviter les erreurs liées à des formats de nombres qui peuvent ne pas être gérés nativement par JavaScript. Finalement, la fonction retourne `serializedStocks`, qui est la version transformée et propre des données enregistrées. (rectangle jaune *Capture 2*)

Dans le rectangle rouge se trouve la gestion d'erreur de cette méthode, qui renvoie un message accompagné de l'erreur si une exception est rencontrée.

## 2.4.2 Retirer une commande en attente

L'espace dédié aux commandes permet d'associer correctement les matériaux ou médicaments désirés par un utilisateur à une toute nouvelle commande. Cet espace permet aussi d'accéder aux commandes toujours en attente.

Comme expliqué précédemment, le fonctionnement des différents espaces est très similaire. Cet espace est ainsi également organisé en plusieurs fonctionnalités, de la même manière qu'observé précédemment.

Ici, je vais prendre l'exemple de la fonctionnalité de suppression d'une commande en attente.

```
app > api > commandeAttente > [id] > TS route.ts > DELETE
59 export async function DELETE(
60   req: NextRequest,
61   context : routeContexte,
62 ) {
63   try {
64     const params = await context.params
65     const { id } = params;
66     const commandeDeleted = await DeleteCommande(id);
67
68     if (!id) {
69       return NextResponse.json(
70         { error: "Commande_id est requis." },
71         { status: 400 }
72       );
73     }
74     if (!commandeDeleted) {
75       return NextResponse.json({
76         error: "Erreur lors de la suppression commande.",
77       });
78     }
79     return NextResponse.json(
80       { message: "Commande supprimer avec succès." },
81       { status: 200 }
82     );
83   } catch (error) {
84     return NextResponse.json(
85       { error: "erreur lors de la suppression commande." },
86       { status: 500 }
87     );
88   }
89 }
```

*Capture 1*

Comme pour la fonctionnalité d'ajout, la suppression de commande repose sur deux espaces : une méthode dans l'api (*Capture 1*) et une autre dans les services (*Capture 2*).



D'abord la méthode extrait l'identifiant de la commande à supprimer à partir des paramètres de la requête, params est récupéré depuis context.params, puis l'ID est extrait de params. Ensuite, commandeDeleted est défini en appelant DeleteCommande(id) (*Capture 2*). (rectangle bleu *Capture 1*)

Cette partie gère les cas où les conditions nécessaires à la suppression ne sont pas remplies.

- Si id est indéfini ou nul (if (!id)).
- Si la valeur de commandeDeleted est false (if (!commandeDeleted)).

Un message d'erreur est envoyé sur la console du serveur si un de ces deux cas est rencontré. (rectangle orange)

Enfin, si la méthode n'est pas interrompue par ces conditions, le code renvoie un JSON confirmant la suppression réussie de la commande. Il inclut un message "Commande supprimée avec succès." et un statut HTTP indiquant que l'opération a été exécutée correctement. (rectangle vert)

Le rectangle rouge correspond au bloc catch, qui, comme montré précédemment, capture toute erreur survenant lors de l'exécution du try. Il s'agit ainsi de la gestion d'erreur de la méthode. Si une exception est levée, elle est automatiquement interceptée par ce bloc.

```
198 export async function DeleteCommande(id: number): Promise<boolean> {  
199   try {  
200     const commande = await prisma.commandes.findUnique({  
201       where: { id_commande: BigInt(id) },  
202     });  
203  
204     if (!commande) return false;  
205  
206     await prisma.commandes.delete({  
207       where: { id_commande: BigInt(id) },  
208     });  
209  
210     return true;  
211   } catch (error) {  
212     return false;  
213   }  
214 }
```

*Capture 2*

D'abord, la fonction va trouver une commande spécifique dans la base de données, dont l'ID correspond à l'id récupéré dans l'api. Si aucune commande n'est trouvée, la fonction retourne immédiatement false (ce qui valide un des test d'erreur dans l'api). Ensuite, si la commande existe, elle est supprimée à l'aide de « prisma.commandes.delete » ; si tout se passe bien, la fonction retourne true, indiquant que la suppression a réussi.

Dans le second rectangle (rouge), comme vu précédemment, nous avons la gestion des erreurs via un bloc catch (error). Si une exception est levée à n'importe quel moment dans le try, cette erreur est capturée et la fonction retourne false.

### 2.4.3 Mouvements

L'espace des mouvements est important car il permet de consulter les allers et les venues dans le stock ainsi que les dates auxquelles les changements ont été effectués. Il permet de rester vigilant en cas de mouvement suspect ou simplement d'aider l'analyse de ces données dans le cadre d'un compte rendu par exemple.

A nouveau, de la même façon que montré précédemment, la fonctionnalité qui permet de récupérer tous les mouvement de la base de données dépend des deux espaces de l'api et du service.

```
app > api > mouvements > TS route.ts > GET
4 export async function GET() {
5   try {
6     const mouvements = await GetAllMouvements();
7     return NextResponse.json(mouvements, { status: 200 });
8   } catch (error) {
9     return NextResponse.json({ error: "Failed to fetch mouvements" }, { status: 500 });
10  }
11 }
```

Capture 1

La méthode GET est très simpliste, une première ligne déclare une constante mouvements qui attend la réponse de la fonction GetAllMouvements() du service. Ensuite, la méthode NextResponse.json() retourne les données récupérées sous forme de réponse JSON avec un code de statut HTTP 200, indiquant que la requête a été traitée avec succès ; la partie dans le rectangle rouge est la gestion d'erreur.

```
services > TS mouvementService.ts > GetAllMouvements
19 export async function GetAllMouvements(): Promise<SerializedMouvement[]> {
20   try {
21     const mouvements = await prisma.mouvements.findMany({
22       include: {
23         stocks: true,
24       },
25     });
26     const serializedMouvement: SerializedMouvement[] = JSON.parse(JSONbig.stringify(mouvements));
27     console.error("mouvements", serializedMouvement);
28     return serializedMouvement;
29   } catch (error) {
30     throw new Error("Aucun mouvements récupérés");
31   }
32 }
```

*Capture 2*

La méthode `GetAllMouvement` fait le plus gros du travail, elle commence par récupérer tous les enregistrements de la table `mouvements`. L'option `include: { stocks: true }` signifie que les stocks associés à chaque mouvement seront également récupérés. Ensuite, la variable `serializedMouvement` est définie. Comme vu précédemment, elle est créée en convertissant des données (ici `mouvements`) en une chaîne JSON avec `JSONbig.stringify` puis en la retransformant en objet JavaScript avec `JSON.parse()`. (rectangle bleu *Capture 2*)

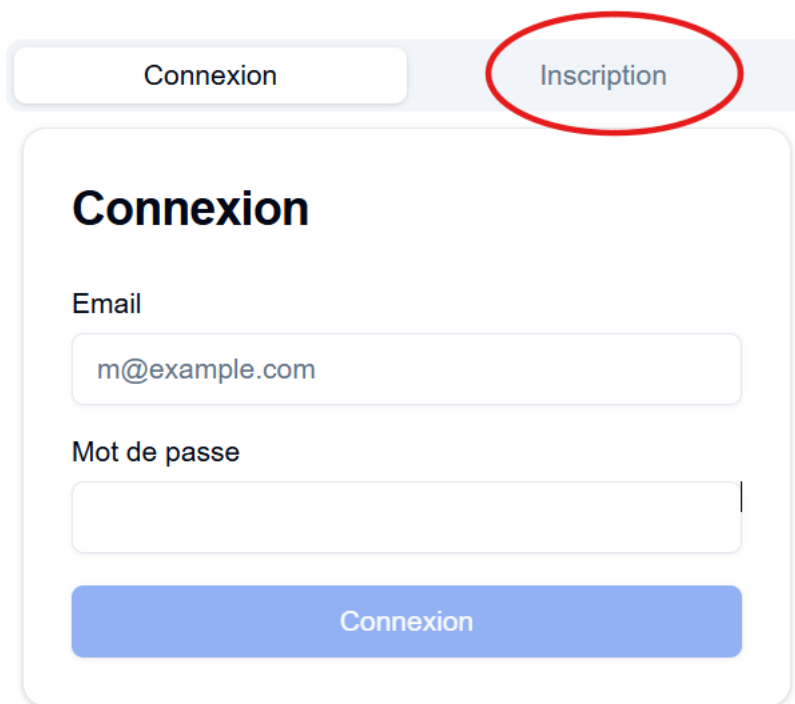
Enfin, comme toutes les fois précédentes, la section du rectangle rouge correspond au bloc `catch`, qui capture les erreurs survenant dans le `try`.

## 2.5 Présentation de l'application

Pour mieux comprendre le fonctionnement de l'application, je vais expliquer comment un utilisateur classique utilise les fonctionnalités que nous avons abordées précédemment.

### 2.5.1 Créer un nouveau compte

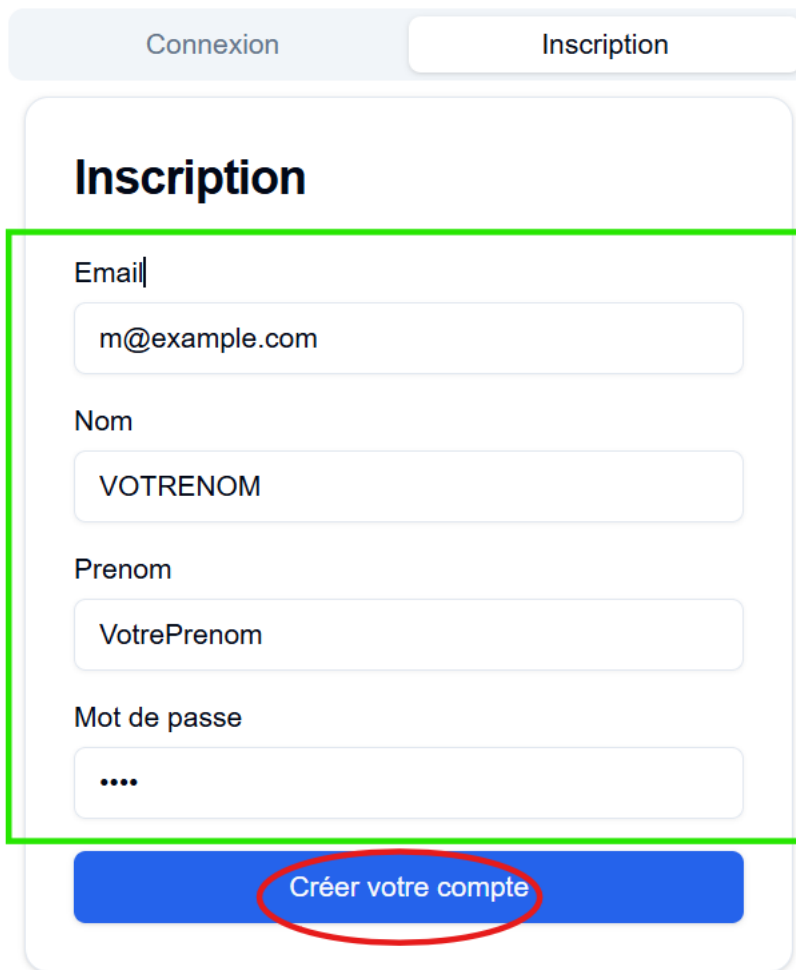
En partant du principe qu'un utilisateur est nouveau sur cette application, la première chose à faire est de créer un compte.



The screenshot shows a user interface with two tabs at the top: 'Connexion' and 'Inscription'. The 'Inscription' tab is highlighted with a red circle. Below the tabs is a form titled 'Connexion'. The form contains two input fields: 'Email' with the placeholder text 'm@example.com' and 'Mot de passe'. At the bottom of the form is a blue button labeled 'Connexion'.

*Capture 1*

Une fois sur la page d'accueil, il suffit de cliquer sur le bouton "Inscription", situé tout en haut à droite (cercle rouge capture 1).



Connexion    Inscription

## Inscription

Email  
m@example.com

Nom  
VOTRENOM

Prenom  
VotrePrenom

Mot de passe  
....

**Créer votre compte**

Capture 2

Figure 1:

Ensuite, sur la page du formulaire, l'utilisateur doit renseigner correctement les informations demandées dans chaque champ (rectangle vert capture 2), puis il faut cliquer sur le bouton permettant de valider les informations (cercle rouge capture 2). Il est important de noter que si les informations sont incorrectes, le mail de confirmation n'atteindra jamais sa destination.

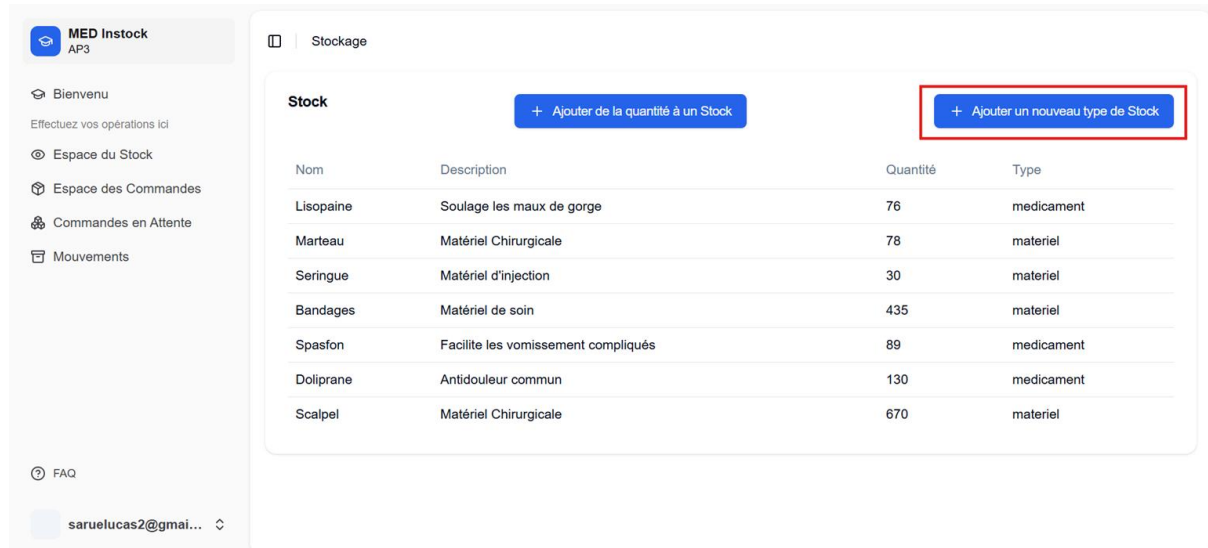
## 2.5.2 Ajouter un stock (Administrateur)

Les utilisateurs disposant de droits d'administrateurs sont libres de créer de nouveaux types de stock pour diversifier les commandes qu'il est possible de passer.



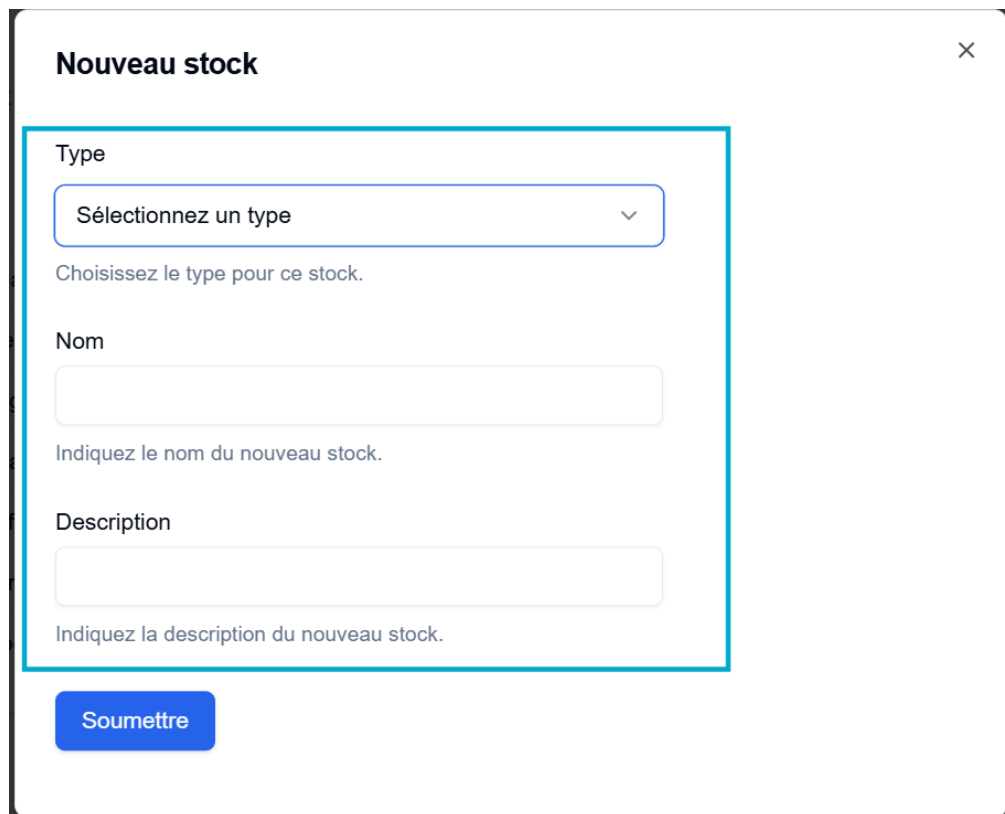
Capture 1

Pour ajouter un nouveau type de stock, commencez par accéder à l'espace du stock. Pour cela, cliquez sur le bouton à gauche de l'écran « Espace du Stock » (rectangle rouge *Capture 1*).



Capture 2

Une fois dans cet espace, il faut cliquer sur le bouton bleu en haut à droite, intitulé « Ajouter un nouveau type de stock », qui ouvre le formulaire de création de stock (rectangle rouge *Capture 2* ). On note que ce bouton n'est accessible que pour les administrateurs. Une fois dans le formulaire, remplir les critères demandés dans le rectangle bleu de la *capture 3* puis soumettre les informations est la dernière étape pour l'ajout d'un nouveau stock.



**Nouveau stock** ×

Type

Sélectionnez un type ▼

Choisissez le type pour ce stock.

Nom

Indiquez le nom du nouveau stock.

Description

Indiquez la description du nouveau stock.

**Soumettre**

*Capture 3*

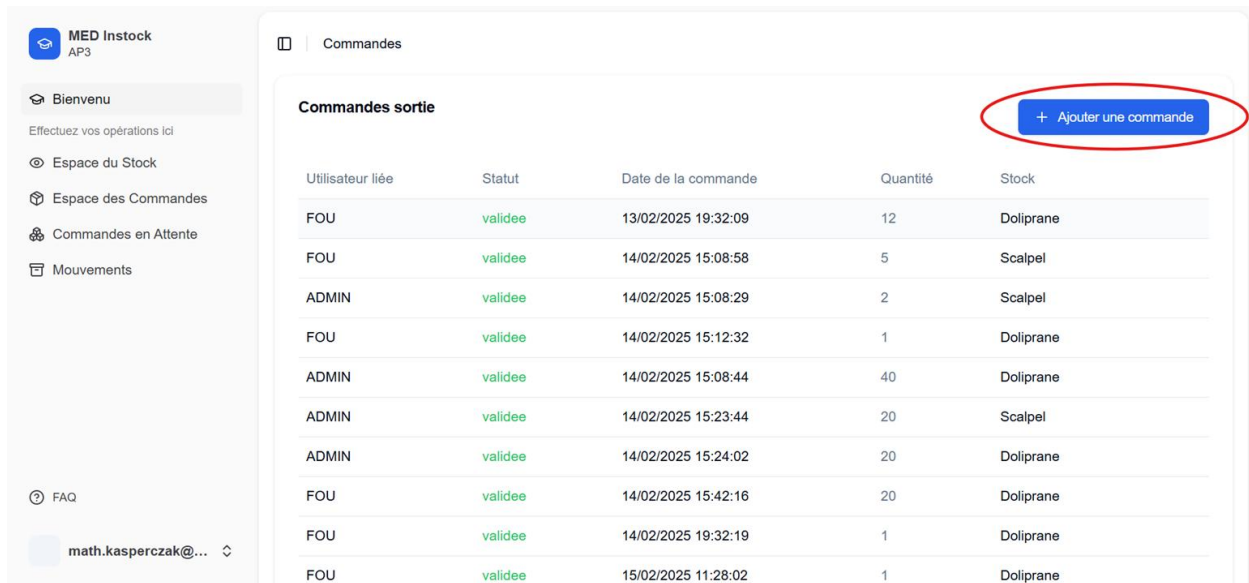
### 2.5.3 Formuler une commande

Finalement, la fonctionnalité la plus importante de ce projet est la formulation et la création d'une commande.



*Capture 1*

Pour formuler une nouvelle commande, commencer par accéder à l'espace des Commandes à l'aide du bouton à gauche de l'écran « Espace des Commandes » (rectangle rouge *Capture 1*).



*Capture 2*

Une fois dans cet espace, il faut cliquer sur le bouton bleu en haut à droite, intitulé « Ajouter une commande », qui ouvre le formulaire de formulation d'une nouvelle commande (rectangle rouge *Capture 2*). Une fois dans le formulaire, remplir les critères



demandés dans le rectangle bleu de la *capture 3* puis soumettre les informations, la nouvelle commande devrait apparaître avec les autres dans la liste !



The image shows a web form titled "Nouvelle commande" with a close button (X) in the top right corner. The form contains two main input fields, both of which are enclosed in a blue rectangular highlight. The first field is labeled "Quantité" and contains a single vertical bar character "|". Below it is the instruction "Indiquez la quantité de la commande." The second field is labeled "Stock concerné" and contains the text "Sélectionnez un stock" with a downward arrow icon. Below it is the instruction "Choisissez le stock pour cette commande." At the bottom left of the form is a blue button labeled "Soumettre".

*Capture 3*

# 3 CONCLUSION

## 3.1 Apprentissages personnels

### 3.1.1 Difficultés rencontrées

Lors de la création de cette application, plusieurs difficultés ont été rencontrées et ont dû être gérées.

Le projet devait être réalisé en deux mois, ce qui a mis une pression considérable sur le planning et l'organisation, d'autant que j'ai été dans l'incapacité d'y consacrer mon temps plein à cause de ma condition d'alternant : certaines périodes devaient être consacrées au travail de l'entreprise.

Travailler sous cette contrainte de temps signifiait que chaque étape du développement devait être optimisée et que les erreurs devaient être minimisées.

Un autre défi majeur a été la découverte du modèle de logique de Next.js : j'ai mis du temps pour devenir à l'aise avec cette nouvelle technologie et cette nouvelle logique .

Une mauvaise gestion de mon temps déjà limité a également eu pour effet de stresser un travail intensif sur deux semaines afin de pouvoir compléter cette application et ce rapport dans les temps impartis.

Malgré ces obstacles, chaque difficulté a offert une opportunité de croissance et d'amélioration des compétences en gestion de projet, et j'ai le sentiment de sortir grandi de ce projet.

### 3.1.2 Evolution personnelle

Au cours de la réalisation de ce projet, j'ai constaté une évolution personnelle dans plusieurs domaines :

L'une des premières évolutions notables a été ma capacité à gérer le temps et à établir des priorités. Avec deux mois pour réaliser ce projet, j'ai d'abord cru pouvoir me focaliser sur mon travail en entreprise, mais j'ai dû planifier minutieusement chaque étape et établir des priorités claires pour les tâches à accomplir lorsque je me suis rendu compte que j'avais fortement sous-estimé la charge demandée par l'application.

Cela m'a également appris à rester concentré sous pression et à maintenir une productivité constante tout au long du projet.

Aussi, avant de commencer ce projet, ma connaissance de Next.js était infime, car les seules connaissances que je possédais étaient lointaines et purement théoriques, les objectifs imposés par la réalisation de cette application furent difficile à mener à bout, mais ils m'ont formé et je peux maintenant affirmer que je suis confiant avec les technologies utilisées tout au long de ce projet !

Ce projet a été une expérience enrichissante qui a catalysé ma croissance personnelle et professionnelle !

## 3.2 Annexes

### 3.2.1 Annex 1 : Lien du Projet

<https://ap3-gestiondestock.kemyl.fr>

Lors du test de cette application, vous pouvez vous connectez en tant qu'administrateur à l'aide :

- de l'adresse mail : [sandidkemyl2003@gmail.com](mailto:sandidkemyl2003@gmail.com)

- et du mot de passe : rootroot

Autrement, vous pouvez créer un nouveau compte d'utilisateur.

### 3.2.2 Annexe 2 : Git

Ci-dessous un lien pour accéder à la page Git de mon Application complète :

[https://github.com/Kemyyl/AP3\\_GestionDeStock](https://github.com/Kemyyl/AP3_GestionDeStock)

### 3.2.3 Annexe 3 : BDD

Ci-dessous un lien pour accéder à la page de Supabase qui met à disposition ses ressources :

<https://supabase.com/>