DOSSIER DE PROJET

Concepteur Développeur d'Application
Titre CDA

ROMARIC LAMARE

Sommaire

1. Abstract

As part of my studies as an Application Developer at Wild Code School, I worked on a project to develop a web application for Oros, a fictional equipment rental company (skiing, hiking, climbing, etc.). The goal was to design a simple and intuitive solution to manage bookings and inventory through a web application.

The main features of the application include:

- Admin login with access to the back office,
- Back office: management of product references (name, description, image) and available stock,
- Back office: viewing reservations,
- Front-end: searching for available products by category, name, and date range,
- Front-end (all pages): a shopping cart with options to add, remove, or adjust quantities of selected products,
- Front-end (checkout): cart summary and billing information input, with order processing through a mock payment system.

The project followed the agile SCRUM methodology. At the beginning of each 1- to 2-weeks sprint, we defined tasks (tickets) to complete, ensuring we could present a functional prototype at the end of each cycle.

Following a requirements analysis, mockups, and a definition of the software architecture, the application was developed using Next.js and TypeScript on the client side, chosen for their ability to build performant and scalable web applications. On the server side, Express was used alongside Apollo Server to manage GraphQL queries, while TypeGraphQL helped define schemas and data types. TypeORM was used for data management, interfacing with a PostgreSQL database. Layout was handled in CSS, with Material-UI used for certain components.

The deployment of the application was secured through backend and frontend testing. The site is dynamic, adjusting its content based on user interactions, and is fully responsive across all screen sizes.

2. Compétences couvertes par le projet

Développer une application sécurisée

- Installer et configurer son environnement de travail en fonction du projet
- Développer des interfaces utilisateur
- Développer des composants métier
- Contribuer à la gestion d'un projet informatique

Concevoir et développer une application sécurisée organisée en couches

- Analyser les besoins et maquetter une application
- Définir l'architecture logicielle d'une application
- Concevoir et mettre en place une base de données relationnelle
- Développer des composants d'accès aux données SQL
- et NoSQL

Préparer le déploiement d'une application sécurisée

- Préparer et exécuter les plans de tests d'une application
- Préparer et documenter le déploiement d'une application
- Contribuer à la mise en production dans une démarche DevOps

3. Cahier des charges

Les exigences du projet définies par le client étaient les suivantes :

- L'application web doit être simple et intuitive.

- L'administrateur doit pouvoir se connecter de manière sécurisée et accéder à un back-office permettant de consulter les réservations et de gérer les références produits (nom, description, image) et des stocks.

- Le visiteur (non connecté) doit pouvoir naviguer sur l'ensemble du site.

- Un menu de navigation est accessible sur toutes les pages.

- Le logo doit renvoyer sur la page d'accueil.

- Le visiteur doit pouvoir cliquer sur une catégorie d'articles et consulter les articles disponibles dans cette catégorie précise.

- Le visiteur doit pouvoir consulter la fiche détaillée d'un article et l'ajouter à son panier.

- Le visiteur doit pouvoir rechercher les articles disponibles par nom.

- Le visiteur doit pouvoir accéder à son panier sur toutes les pages, avec la possibilité d’ajouter, de retirer ou de changer la quantité des articles choisis.

- Le visiteur doit pouvoir accéder à un récapitulatif de son panier

4. Spécifications techniques

4.1- Technologies utilisées :

Next.js : framework de développement web JavaScript basé sur React, utilisé pour créer des applications web modernes et performantes. Il simplifie la gestion des routes grâce à son système de routage intégré..

TypeScript : Langage de programmation open source, sur-ensemble syntaxique strict de JavaScript, qui améliore la robustesse et la maintenabilité du code. Le code TypeScript est compilé en JavaScript pour être exécuté dans le navigateur ou sur un serveur.

CSS pour la mise en page.

Material-UI : Bibliothèque de composants React respectant les principes du Material Design, un ensemble de règles de design proposé par Google en 2014, pour unifier l'apparence et l'expérience utilisateur.

NodeJS : Notre back-end a été développé avec **Node.js**, environnement d'exécution JavaScript côté serveur, basé sur le moteur V8 de Google Chrome, permettant de développer des applications serveur performantes et évolutives.

Express : Utilisé comme serveur HTTP principal pour gérer les requêtes et les middlewares. Apollo Server est intégré à **Express** pour traiter les requêtes **GraphQL**, en profitant de la structure d'Express pour ajouter des fonctionnalités comme la gestion des cookies, des en-têtes HTTP, et des sessions utilisateur.

Apollo Studio : Ensemble d'outils dédiés à la gestion et au monitoring des requêtes GraphQL. Apollo Client (côté client) facilite la consommation des APIs GraphQL, tandis qu'Apollo Server (côté serveur) gère l'interface avec notre back-end pour traiter les requêtes GraphQL. Dans le projet Oros, Apollo Client et Apollo Server ont été utilisés pour gérer la communication entre le front-end et le back-end.

GraphQL : Pour la création de notre API, nous avons choisi GraphQL. Ce langage de requête et environnement d'exécution côté serveur permet aux clients de demander uniquement les données dont ils ont besoin. Contrairement aux APIs REST, GraphQL structure les requêtes pour extraire des données de plusieurs sources via un seul appel API. Il utilise un schéma pour décrire les données disponibles et des résolveurs pour produire les valeurs associées aux requêtes.

Côté client, les deux opérations principales de GraphQL sont :

- Queries : elles permettent de lire les données (équivalent du "Read" dans le modèle CRUD).
- Mutations : elles permettent de créer, modifier ou supprimer des données (équivalentes aux opérations "Create", "Update" et "Delete" dans le modèle CRUD).

TypeGraphQL qui est un framework conçu pour implémenter des APIs GraphQL dans Node.js. Il permet de définir le schéma GraphQL en utilisant des classes et des décorateurs. Par exemple, on utilise le décorateur `@ObjectType` pour définir une classe comme un type GraphQL, et `@Field` pour déclarer les propriétés de cette classe qui seront mappées aux champs GraphQL.

De plus, Type-GraphQL simplifie la création de requêtes (queries), de mutations, et de champs en les définissant comme des méthodes de classe ordinaires, similaires aux contrôleurs REST. Cela permet une séparation claire entre la logique métier et la couche de transport, et facilite également les tests unitaires des résolveurs, qui peuvent être traités comme de simples services.

PostgreSQL : Système de gestion de base de données relationnelle open source, utilisé pour sa robustesse et sa gestion avancée des transactions. Intégré via **TypeORM**, il permet des opérations CRUD simplifiées et une gestion efficace des relations entre entités.

TypeORM pour la communication avec la base de données : Il s'agit d'un ORM (Object-Relational Mapping) qui permet de convertir les données d'une base de données relationnelle en objets dans un programme orienté objet. Il facilite l'interaction avec la base de données, en masquant les requêtes SQL complexes derrière des méthodes orientées objet, notamment pour les opérations CRUD.

Docker pour créer des environnements isolés pour les applications en conteneurs, ce qui facilite la gestion des dépendances et la cohérence entre les environnements de développement, de staging et de production.

Docker Compose : Outil permettant d'orchestrer plusieurs conteneurs Docker, en définissant les services, réseaux, et volumes dans un fichier YAML. Il gère l'ordre d'exécution des services via la directive *depends_on* et inclut un mécanisme de **healthcheck** pour surveiller la santé des conteneurs, garantissant leur bon fonctionnement avant de lancer les services dépendants. assurant ainsi un démarrage synchronisé et fiable de l'application.

DockerHub : Utilisé comme registre centralisé pour stocker et partager les images Docker, facilitant ainsi le déploiement des conteneurs dans différents environnements.

Jest : **framework de test** simple et léger qui offre beaucoup de fonctionnalités de test pour les projets JavaScript et TypeScript.

Visual Studio Code pour l'environnement de développement

Pour l'hébergement de l'application, nous disposions d'un serveur VPS fourni par l'école, garantissant un accès sécurisé aux ressources internes. Nous avons utilisé le serveur web Caddy pour gérer les requêtes HTTP entrantes, bénéficiant de son support natif du HTTPS, ce qui a simplifié la configuration des certificats de sécurité. Docker a été employé afin de standardiser les environnements, faciliter le déploiement du projet, et mettre en place un environnement de staging pour un déploiement continu. En complément, nous avons également utilisé Nginx pour équilibrer les charges et améliorer la gestion des requêtes.

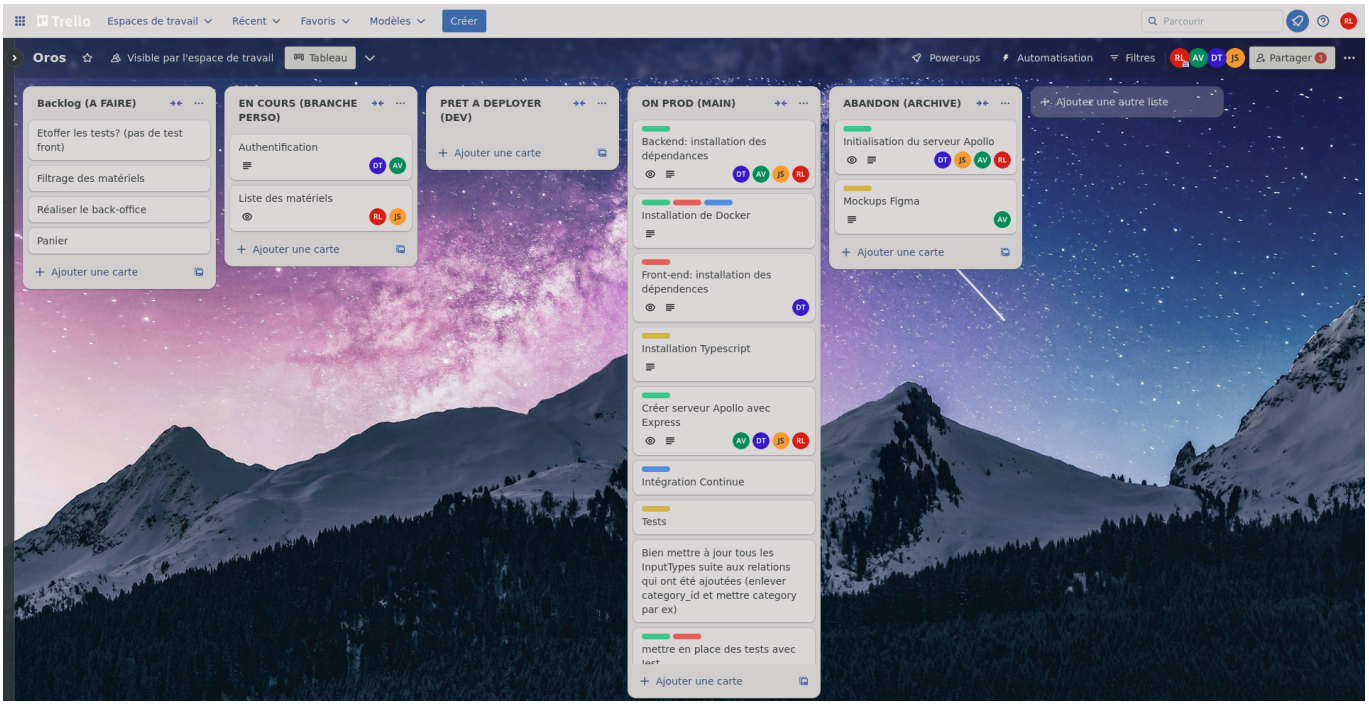
4.2- Architecture

Nous avons utilisé l'architecture **MVC** (Modèle-Vue-Contrôleur), qui est un modèle de conception servant à organiser et structurer les applications à l'aide de trois composants : le Modèle, la Vue et les Contrôleurs (cf. p29).

5. Gestion de projet

5.1- Méthode agile

Nous avons suivi la méthodologie Agile **Scrum**. L'équipe de développement a établi un **backlog** des fonctionnalités à implémenter à l'aide d'un outil de suivi en ligne qui s'appelle **Trello**. Les **tickets** sont programmés et assignés à un développeur, ce qui permet d'avoir une visibilité sur le travail en cours et à venir de chacun.



Le projet étant réalisé sur les temps de formation, l'équipe a travaillé, au cours des sprints successifs, sur des tâches spécifiques, effectuant des revues fréquentes pour s'assurer que les spécifications techniques étaient en adéquation avec les besoins du client et que le produit final répondait aux attentes tout en étant **évolutif**.
Chaque jour, nous faisions un tour de table (**daily toast**) au cours duquel chaque développeur s'exprimait sur le travail réalisé, les difficultés potentielles rencontrées et l'objectif de la journée.

A l'issue de chaque sprint, nous proposions au client (nous présentions notre travail au groupe de formation)) un produit intermédiaire mais fonctionnel (**MVP** : Minimum Viable Product), tout en étant dans la capacité de **s'adapter** à d'éventuels changements de directives.

Un **Scrum Master** était également désigné à chaque tour afin de guider l'équipe Agile, résoudre les obstacles, favoriser l'amélioration continue et assurer l'application du Scrum.

5.2- Outils utilisés

- **Git / GitHub** pour le versionnage :

  Git est un système de contrôle de version qui permet de suivre les modifications du code en local.

  GitHub est une plateforme en ligne basée sur Git, offrant des fonctionnalités de collaboration, d'**intégration** via les **GitHub Actions**, et de **déploiement continue** à l'aide de **Webhooks**.

  Dans notre projet, nous avons configuré les GitHub Actions pour automatiser des étapes clés. Lors d'un push sur la branche *dev*, les tests sont automatiquement exécutés, suivis d'un build **Docker** si ces tests réussissent. Par la suite, des webhooks déclenchent un script de déploiement automatique appelé *fetch and deploy*, qui met à jour l'environnement de **staging** après un push sur *dev*, et l'environnement de **production** après un push sur *main*. Cela assure une livraison continue du code et un flux de travail fluide.

- **Trello** pour la gestion de projet agile (élaboration des sprints et des tickets).

- **Miro** pour la modélisation de la base de données et UML.

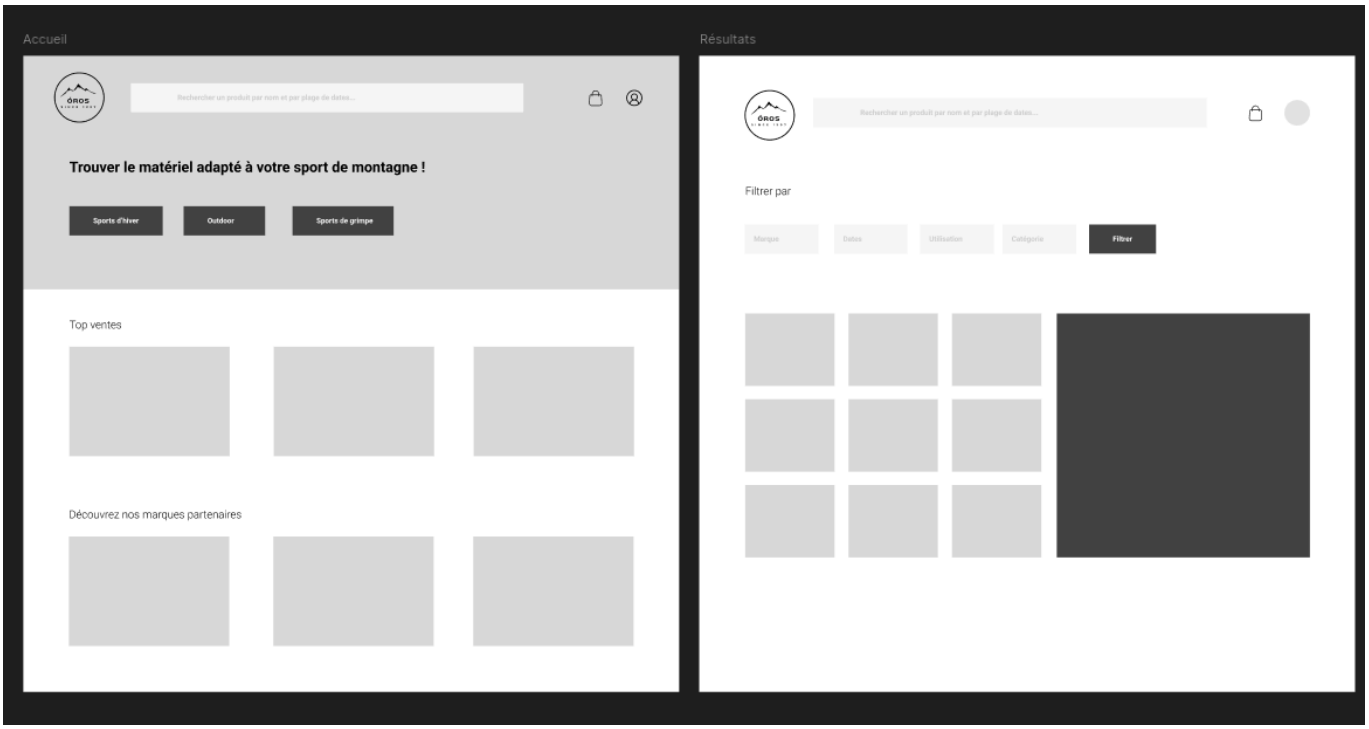
- **Figma** pour la conception du wireframe et de la maquette de l'application

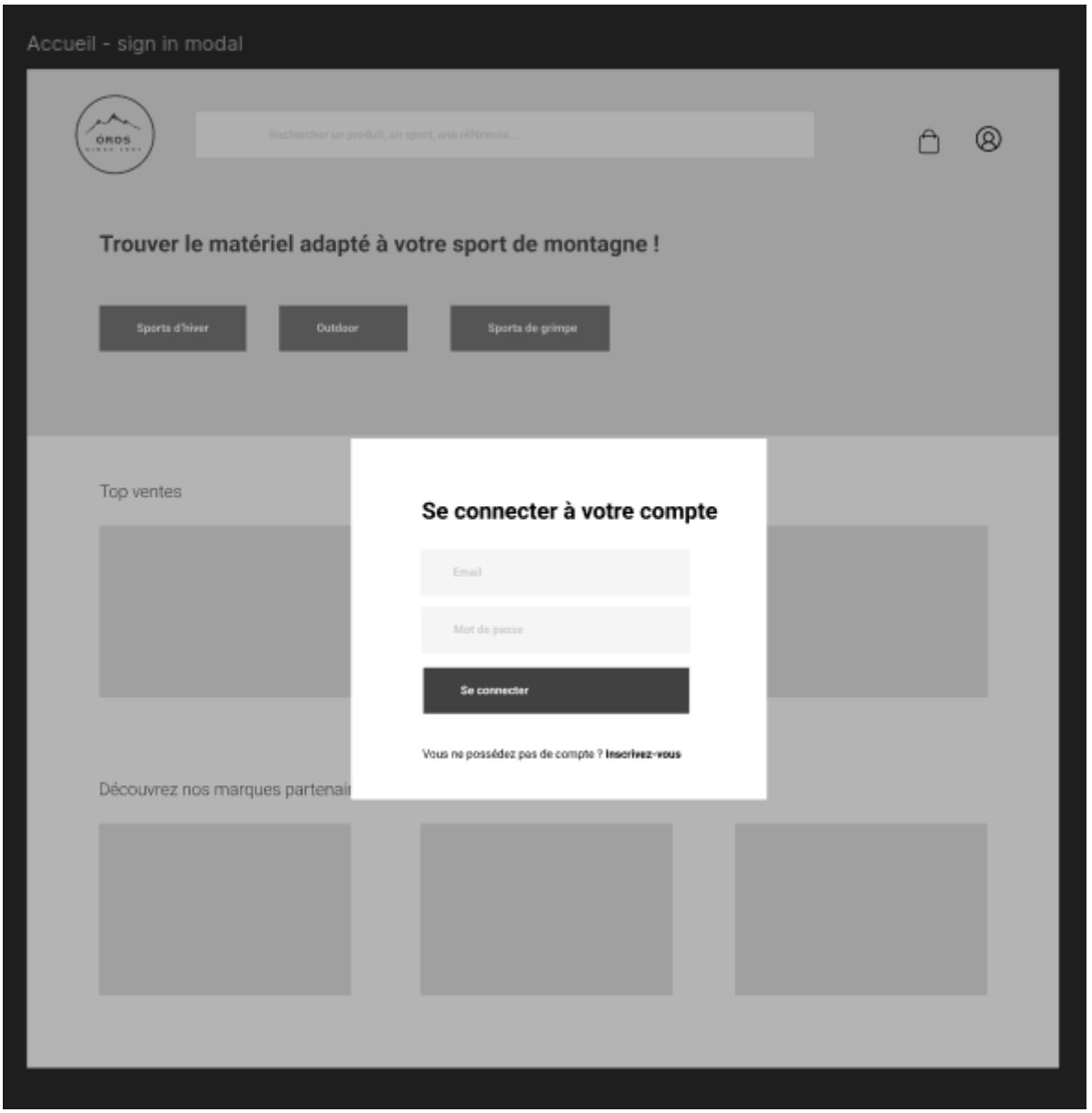
6. Le projet

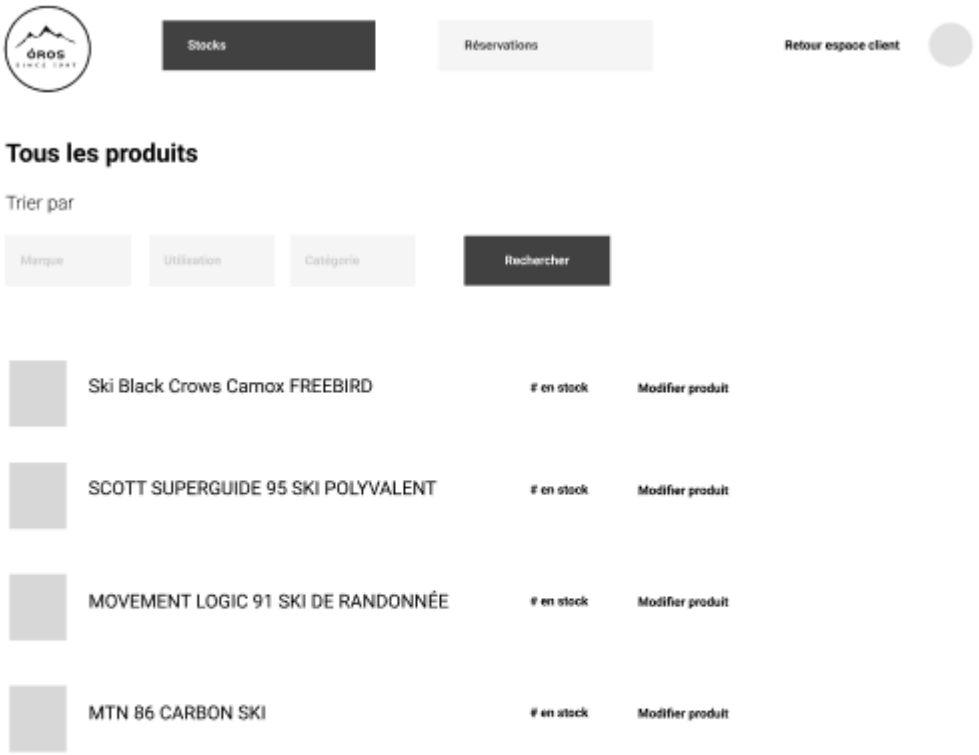
6.1- Le maquettage

Après avoir effectué un **brainstorming** concernant notre vision de l'application en accord avec les éléments transmis par le client dans le **cahier des charges**, nous avons entrepris de réaliser un wireframe à l'aide de l'outil **Figma**. Le **wireframe** nous

permet d'avoir une vue d’ensemble, **schématique** et **simplifiée** de l’application et de visualiser rapidement la structure, l’arborescence et l'organisation des composants et des interactions. Cela constitue un point de départ afin de se lancer dans l’élaboration de la maquette.




Accueil - sign in modal
ÖROS
Trouver le matériel adapté à votre sport de montagne !
Sports d'hiver
Outdoor
Sports de grimpe
Top ventes
Découvrez nos marques partenair
Se connecter à votre compte
Email
Mot de passe
Se connecter
Vous ne possédez pas de compte ? Inscrivez-vous



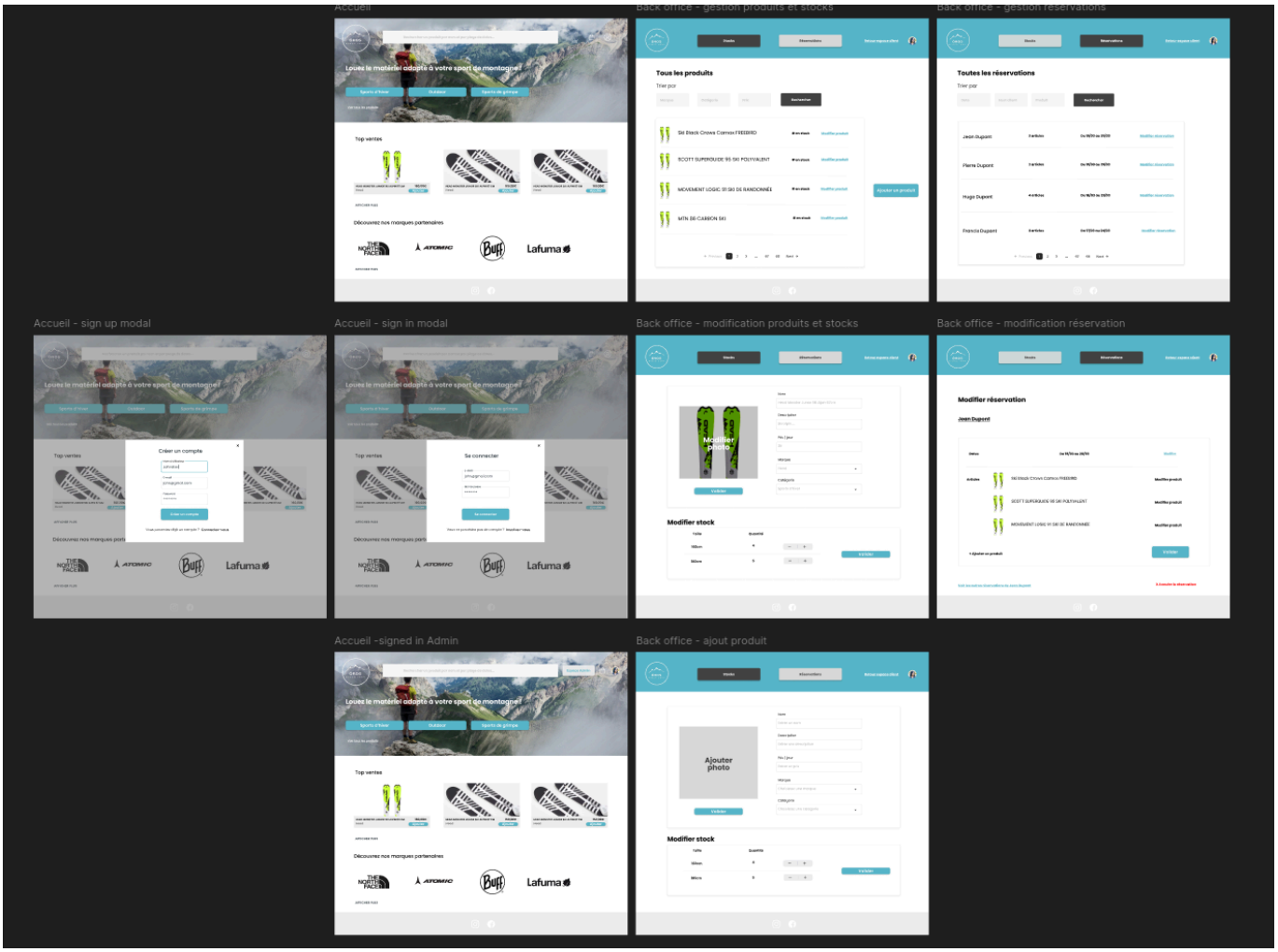
Puis nous avons réalisé un moodboard afin de choisir l'orientation design du site.

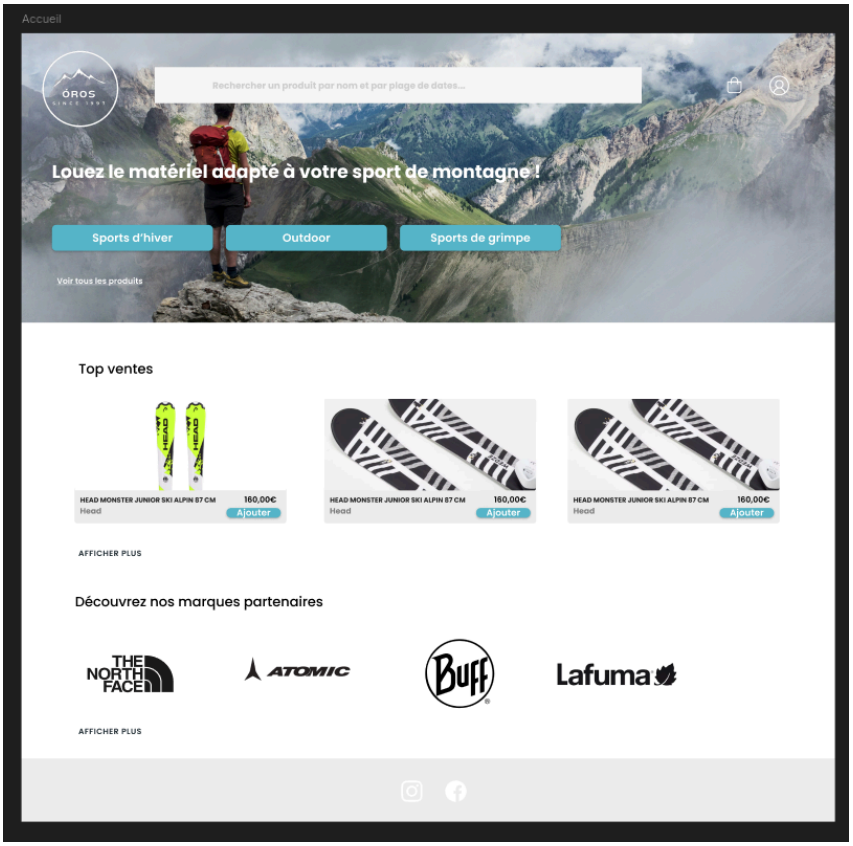
Nous avons réfléchi à l'UX/UI afin d'avoir une expérience efficace, agréable et fonctionnelle et répondre aux désirs du client.

Les maquettes incluent les détails visuels tels que les couleurs, les typographies, les icônes et les images pour refléter le style et l'identité visuelle de l'application. Des éléments d'interaction tels que les boutons et les formulaires ont été intégrés aux maquettes pour représenter les fonctionnalités de l'application.

Les maquettes finales ont servi de référence visuelle pendant le processus de développement et ont facilité la communication entre les différentes parties impliquées dans le projet.

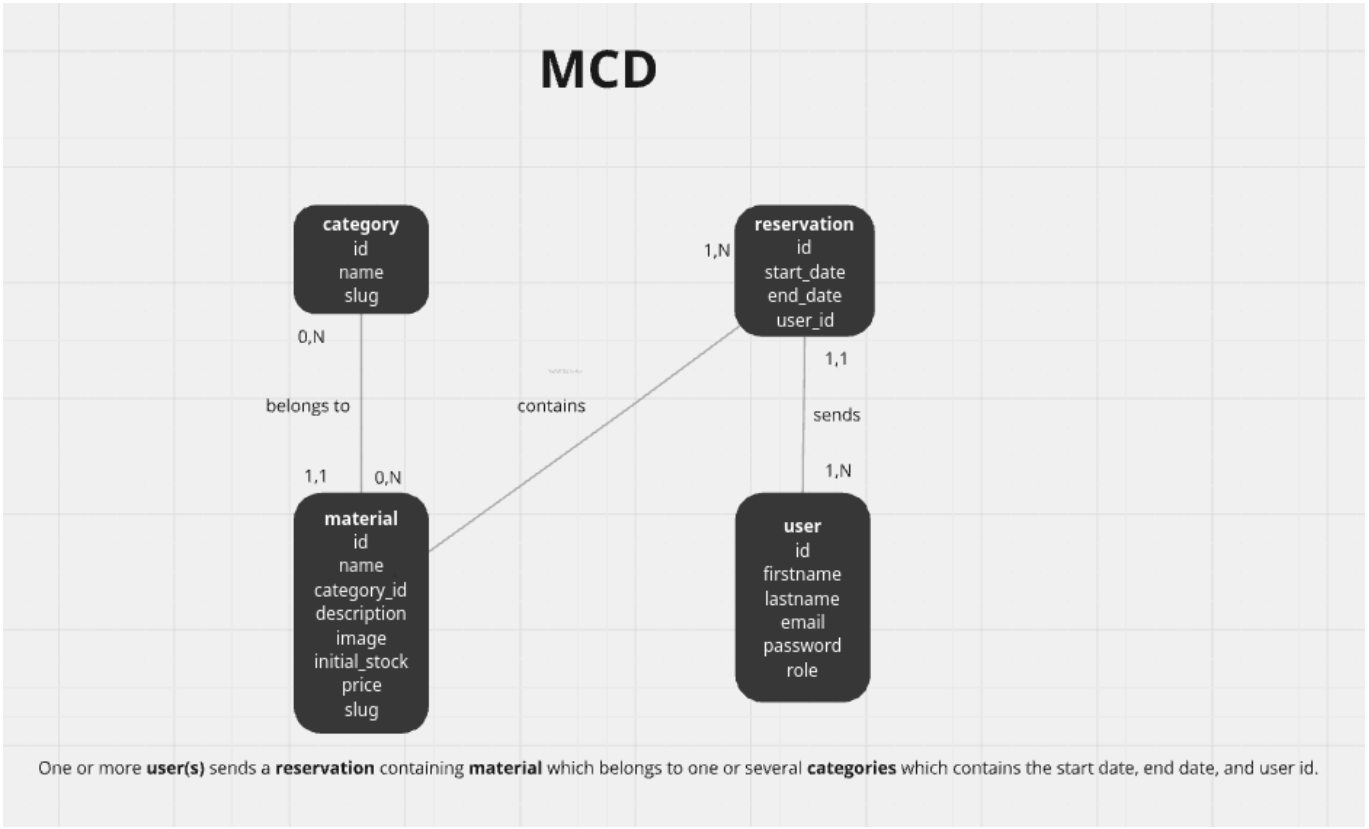
Le maquettage sur Figma a joué un rôle essentiel dans la planification et la réalisation du projet, en aidant à visualiser et à valider le concept avant sa mise en œuvre.


Accueil
Louez le matériel adapté à votre sport de montagne !
Back office - gestion produits et stocks
Tous les produits
Back office - gestion réservations
Toutes les réservations
Accueil - sign up modal
Accueil - sign in modal
Back office - modification produits et stocks
Modifier stock
Back office - modification réservation
Modifier réservation
Accueil -signed in Admin
Back office - ajout produit
Ajouter photo


Accueil
ÖROS
Rechercher un produit par nom et par plage de dates...
Louez le matériel adapté à votre sport de montagne !
Sports d'hiver
Outdoor
Sports de grimpe
Voir tous les produits
Top ventes
HEAD MONSTER JUNIOR SKI ALPIN 87 CM
Head
160,00€
Ajouter
AFFICHER PLUS
Découvrez nos marques partenaires
THE NORTH FACE
ATOMIC
Buff
Lafuma
AFFICHER PLUS

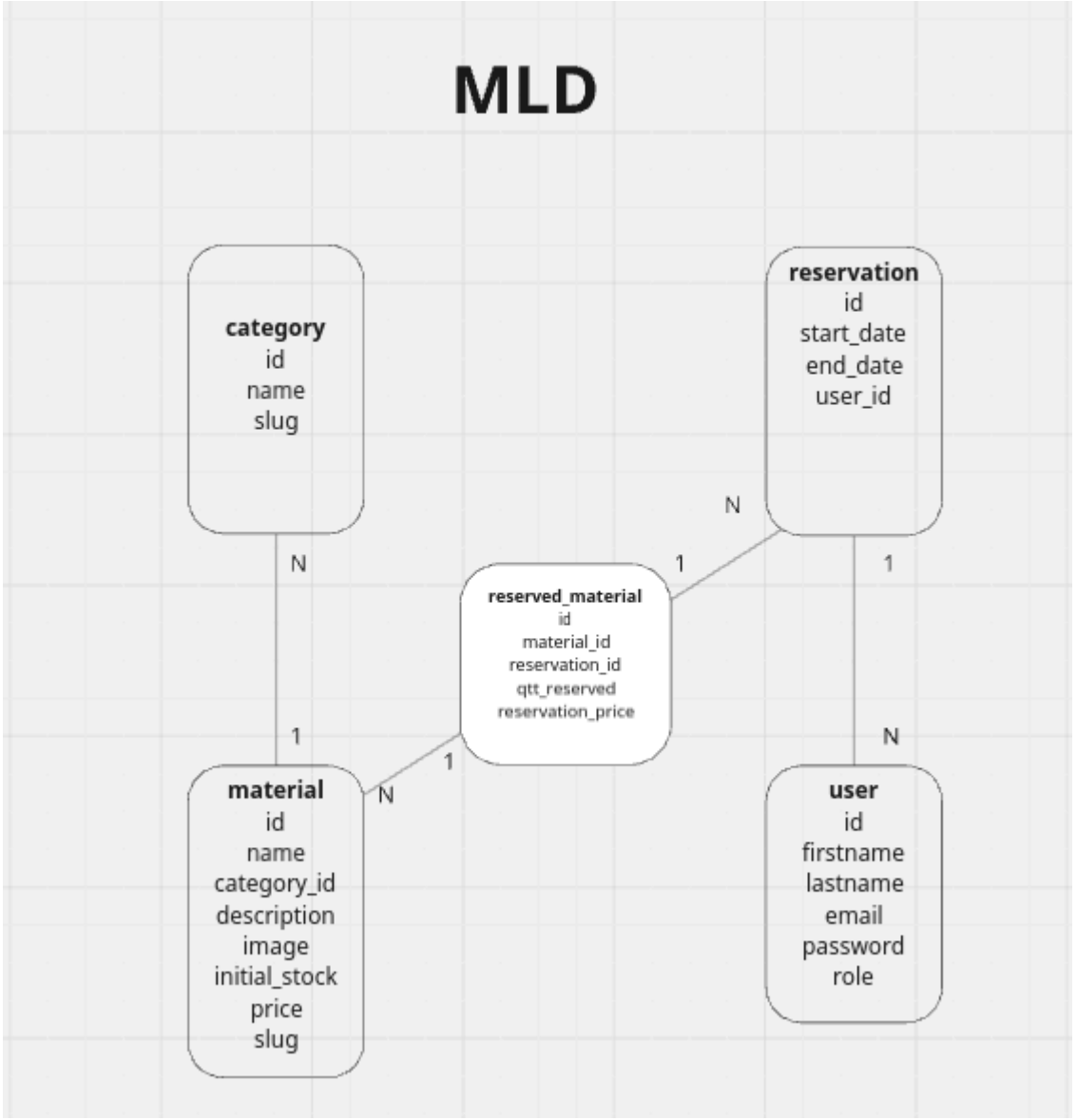
6.2- Conception de la base de données

La conception de la base de données est une étape indispensable, permettant, en amont de l'écriture de celle-ci, de la modéliser et d'en définir les composantes. Pour cela, dans le cadre du projet **Oros**, nous avons utilisé la méthode Merise pour définir les modèles de données.
notamment à travers le **Modèle Conceptuel de Données** (MCD), afin de structurer les entités principales telles que les utilisateurs, le matériel, les réservations et les catégories, de bien établir leurs relations et d'en définir les attributs afin d'avoir une vue complète du système.

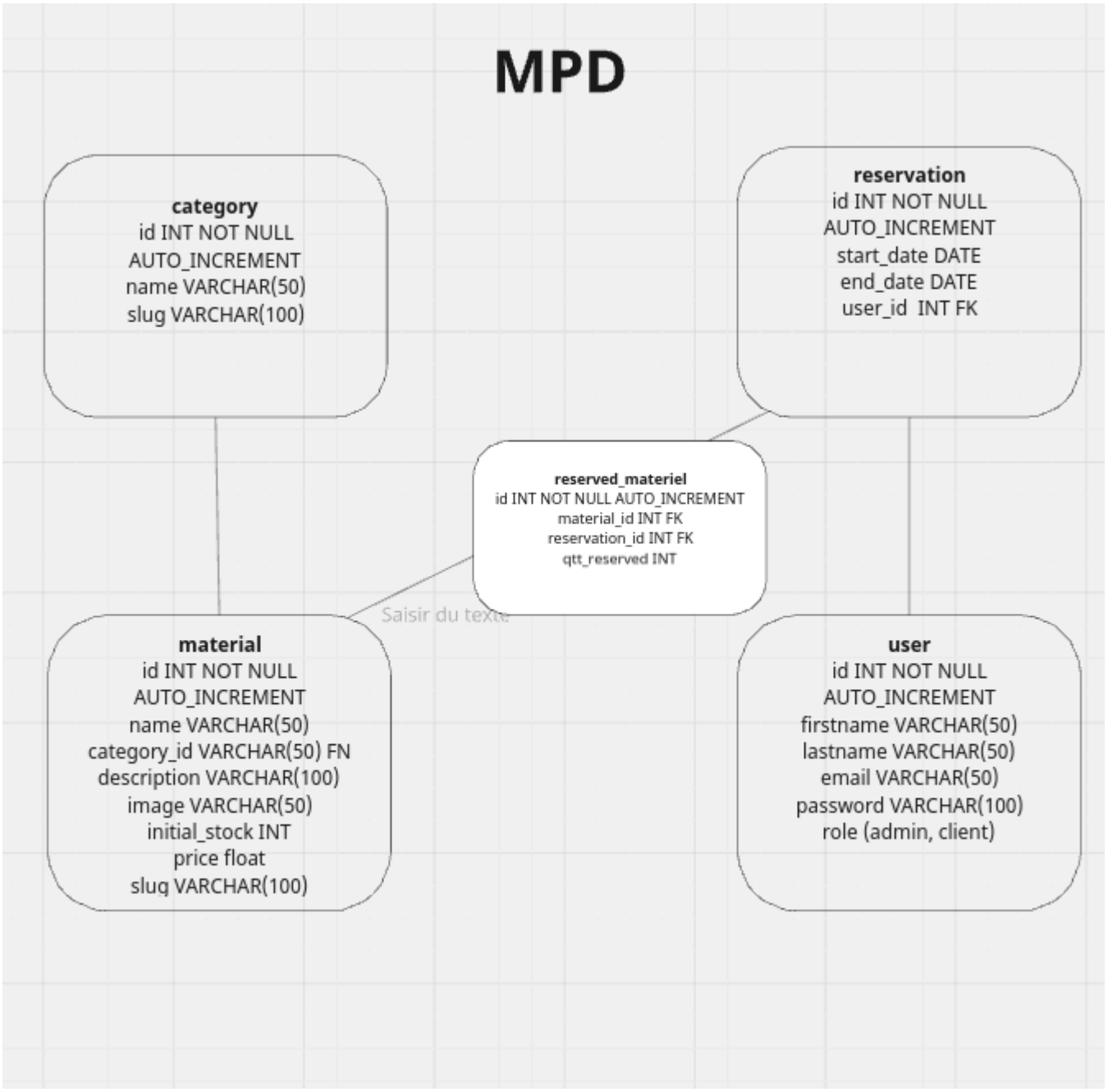


Ensuite, nous avons converti notre MCD en un ensemble compréhensible pour un SGBD, le **Modèle Logique de Donnée**s (MLD). Les entités sont remplacées par des éléments de bases de données : les tables.

Les verbes d'associations ont donné lieu aux **clés étrangères** (la clé primaire d'une table pourra devenir la clé étrangère d'une autre table avec laquelle elle a une relation par exemple). On ne garde plus que la **cardinalité** maximum (celle de droite dans chaque cardinalité). En cas de **ManyToMany**, nous définissons une table intermédiaire.


MLD
category
id
name
slug
reservation
id
start_date
end_date
user_id
reserved_material
id
material_id
reservation_id
qtt_reserved
reservation_price
material
id
name
category_id
description
image
initial_stock
price
slug
user
id
firstname
lastname
email
password
role
N
1
1
N
1
N
1
N

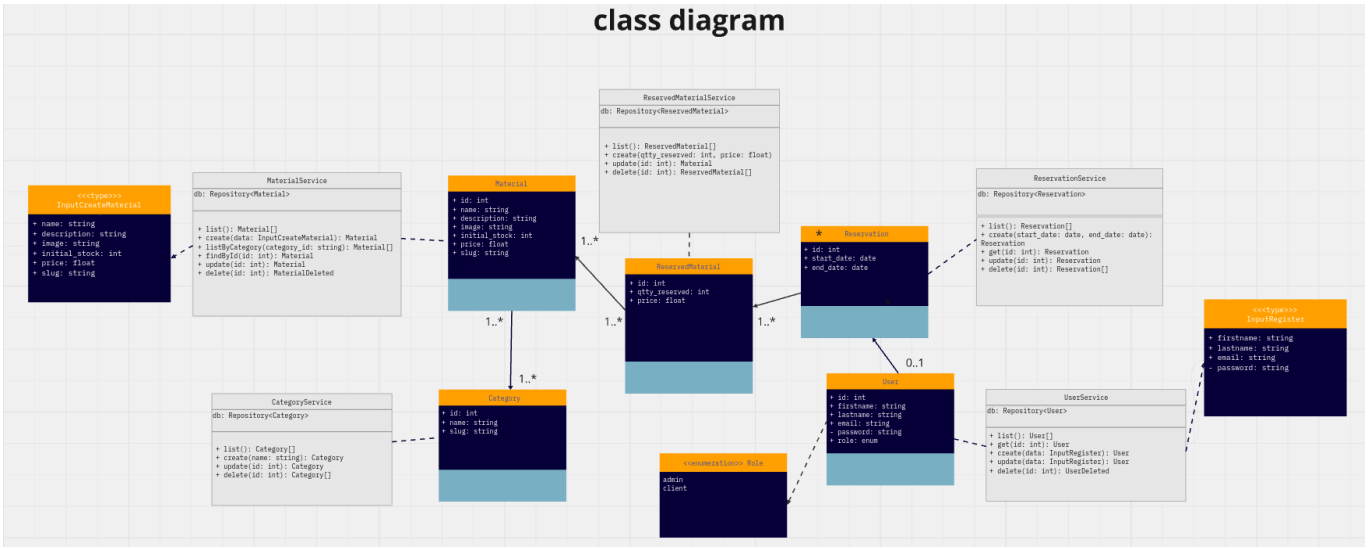
Enfin, nous avons généré le **Modèle Physique de Données** (MPD), spécifiant la structure des tables et les contraintes (Clé primaire, clé étrangère, contrainte de non-nullité, types, etc) de la base de données **PostgreSQL**, utilisée pour le projet.

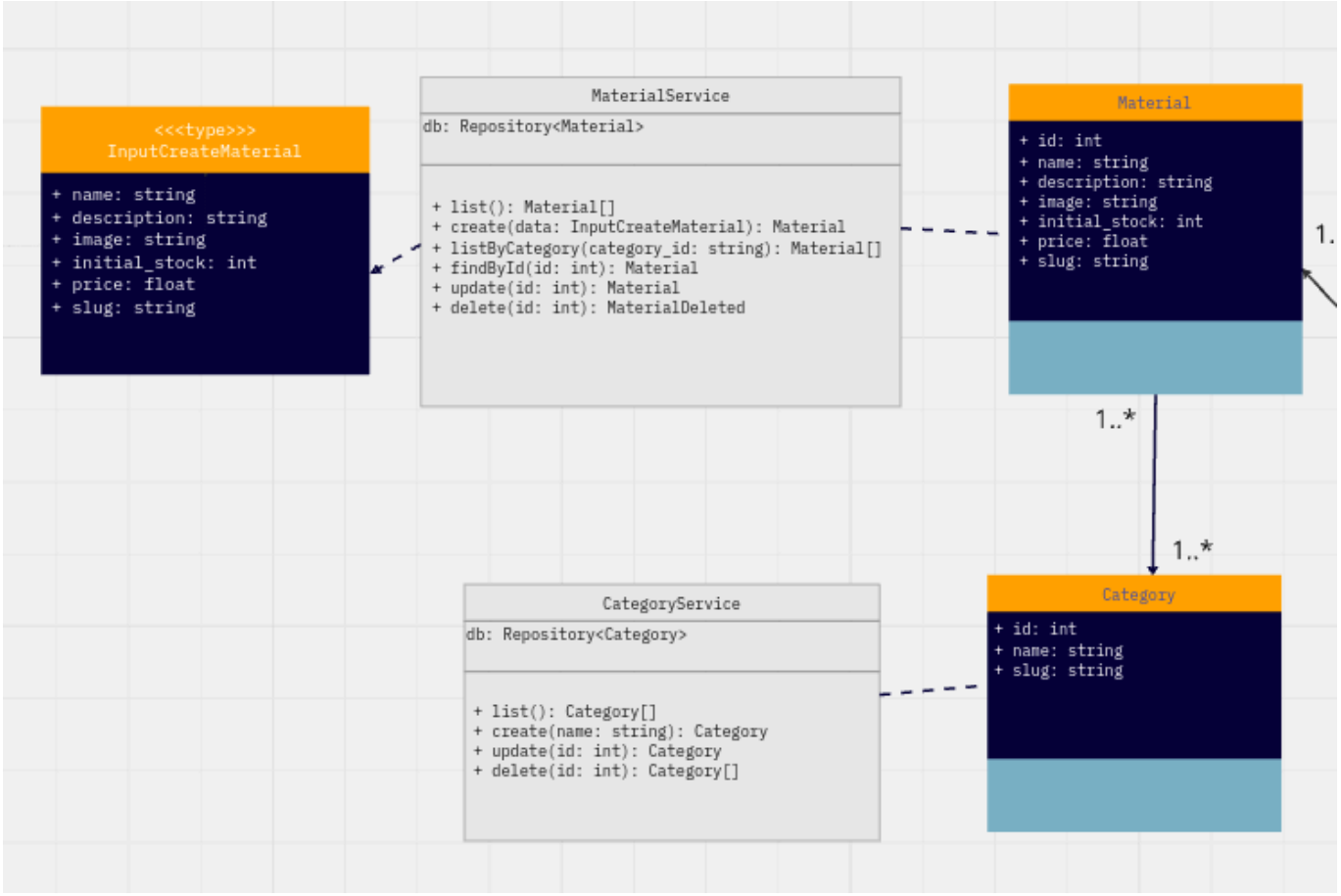


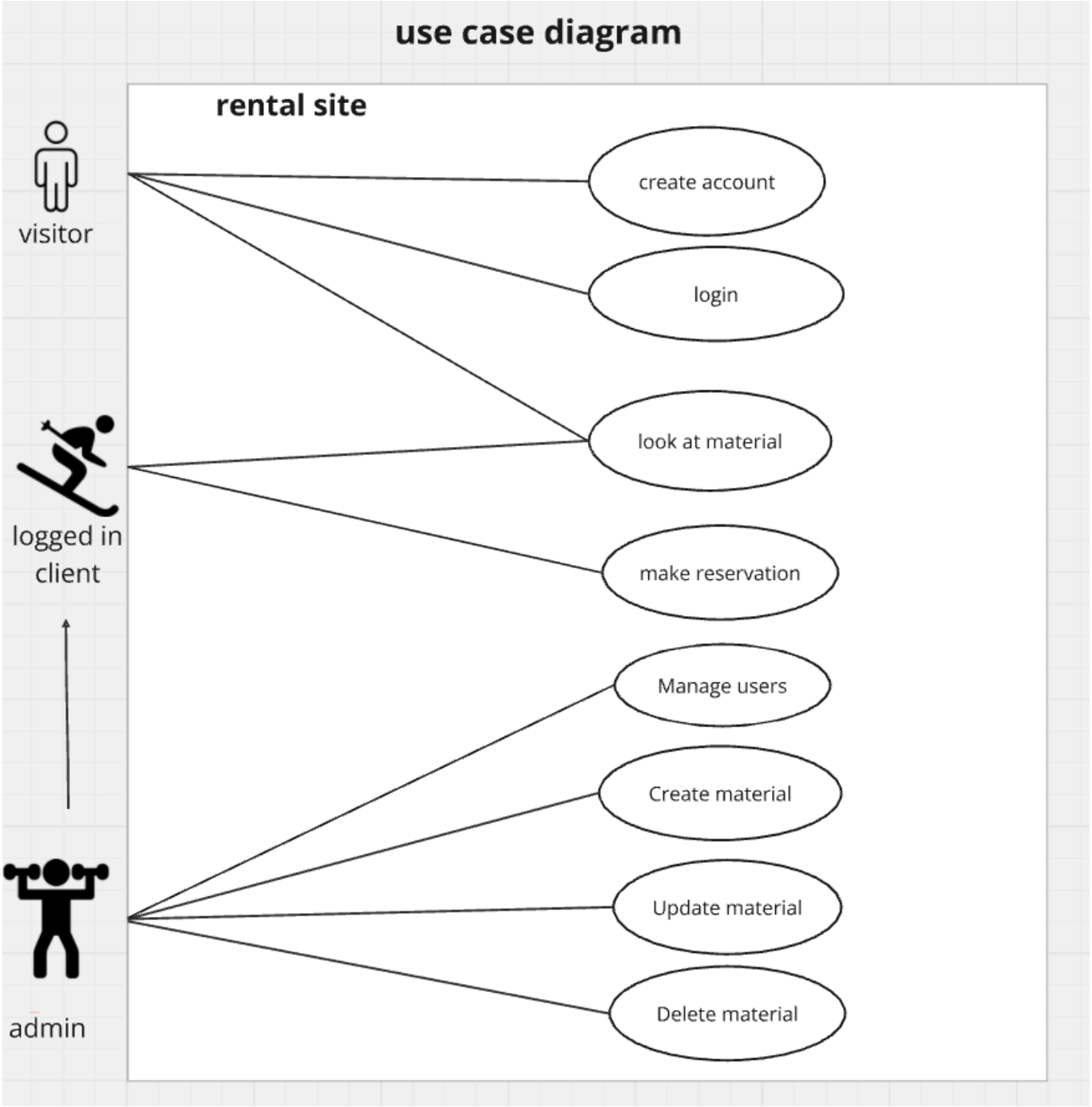
En complément, **l'UML (Unified Modelling Language)** a été utilisé pour définir les interactions et processus à travers un **diagramme de classes**, qui reflète la définition des classes transformées en entités via **GraphQL**, les services et méthodes intégrés avec **TypeORM**, ainsi que les inputs pour les interactions avec le front-end. Ce diagramme couvre également les **cas d'utilisation (Use Cases)** liés aux principales fonctionnalités de l'application, telles que la gestion des utilisateurs, la réservation de matériel et la catégorisation des produits. Grâce à cette modélisation, nous avons pu

obtenir une vue d'ensemble à la fois fonctionnelle et technique, facilitant l'implémentation avec **TypeORM** pour gérer les entités et les relations dans PostgreSQL.

Ainsi, cette approche combinée (Merise pour la modélisation conceptuelle et UML pour la représentation des interactions système) a été essentielle pour assurer une parfaite adéquation entre les besoins métier et la structure technique de la base de données.




<<<type>>>
InputCreateMaterial
+ name: string
+ description: string
+ image: string
+ initial_stock: int
+ price: float
+ slug: string
MaterialService
db: Repository<Material>
+ list(): Material[]
+ create(data: InputCreateMaterial): Material
+ listByCategory(category_id: string): Material[]
+ findById(id: int): Material
+ update(id: int): Material
+ delete(id: int): MaterialDeleted
Material
+ id: int
+ name: string
+ description: string
+ image: string
+ initial_stock: int
+ price: float
+ slug: string
1..*
1..*
CategoryService
db: Repository<Category>
+ list(): Category[]
+ create(name: string): Category
+ update(id: int): Category
+ delete(id: int): Category[]
Category
+ id: int
+ name: string
+ slug: string



6.3- Connexion à la base de données et accès aux données

Une fois la modélisation faite, j'ai mis en place la base de données **PostgreSQL**, en suivant les étapes suivantes :

1. Configuration de TypeORM

J'ai configuré **TypeORM** en définissant les paramètres de connexion à la base de données dans un fichier de configuration. Ce fichier contient des informations

comme le type de base de données (PostgreSQL), l'hôte, le port, l'utilisateur, le mot de passe, et spécifie également les entités utilisées dans le projet (cf. point suivant). Voici ma configuration dans le fichier *DataSource.ts* :

src > lib > TS datasource.ts > ...

```ts
import { DataSource } from 'typeorm';
import Material from '../entities/Material.entity';
import User from '../entities/User.entity';
import Reservation from '../entities/Reservation.entity';
import Category from '../entities/Category.entity';

export default new DataSource({
  type: 'postgres',
  host: 'db',
  port: 5432,
  database: process.env.POSTGRES_DB,
  username: process.env.POSTGRES_USER,
  password: process.env.POSTGRES_PASSWORD,
  entities: [Material, User, Reservation, Category],
  synchronize: true, //à ne pas utiliser en production
  logging: ['error', 'query'], //à ne pas utiliser en production
});

```

2. Définition des Entités avec TypeORM et TypeGraphQL

J'ai créé des entités pour représenter les tables de la base de données. Chaque entité est définie en tant que classe, avec des décorateurs **TypeORM** pour spécifier les colonnes, les types de données, et les relations entre les entités, et des décorateurs **TypeGraphQL** pour exposer ces entités via l'API GraphQL.

Par exemple, l'entité **Category** représente la table des catégories dans la base de données. Le décorateur **@ObjectType** permet de définir la classe comme un type GraphQL, tandis que **@PrimaryGeneratedColumn** et **@Column** définissent les colonnes de la base de données. **@OneToMany** est utilisé pour indiquer la relation entre les entités **Category** et **Material**. Des types d'entrée (**input types**) sont aussi créés pour permettre la création et la mise à jour des catégories via des mutations GraphQL :

src > entities > TS Category.entity.ts > InputCreateCategory > name

```ts
import { Field, ID, ObjectType, InputType } from 'type-graphql';
import { Column, Entity, OneToMany, PrimaryGeneratedColumn } from 'typeorm';
import Material from './Material.entity';

@ObjectType()
@Entity()
class Category {
  @Field(() => ID)
  @PrimaryGeneratedColumn()
  id: number;

  @Field()
  @Column()
  name: string;

  @Field(() => [Material], { nullable: true })
  @OneToMany(() => Material, (m) => m.category, { nullable: true })
  materials: Material[];

  @Field()
  @Column()
  slug: string;
}

@InputType()
export class InputCreateCategory {
  @Field()
  @Column()
  name: string;

  @Field()
  @Column()
  slug: string;
}
```

```
@InputType()
export class InputUpdateCategory {
  @Field()
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Field({ nullable: true })
  name: string;

  @Field({ nullable: true })
  slug: string;
}

@ObjectType()
export class CategoryDeleted {
  @Field()
  @Column()
  name: string;

  @Field()
  @Column()
  slug: string;
}

export default Category;
```

3. Opérations CRUD avec TypeORM

Les opérations de création, lecture, mise à jour et suppression des données sont réalisées dans les **services** à l'aide des méthodes fournies par **TypeORM**. Cela permet de manipuler efficacement les données dans **PostgreSQL** tout en gardant la logique métier séparée de l'implémentation des requêtes.

Prenons l'exemple du service **CategoryServices** qui gère les entités **Category**.

Le service utilise **TypeORM** pour interagir avec la base de données **PostgreSQL**, en exposant des méthodes comme la création (**create**), la lecture (**findById** et **list**), la mise à jour (**update**), et la suppression (**delete**) d'une catégorie. Le service est

également conçu pour gérer les relations entre les catégories et les matériaux associés via les décorateurs **TypeORM**, comme **@OneToMany**.

Voici un exemple des **méthodes CRUD** dans le fichier **CategoryServices** :

```
  async update(id: number, data: Omit<InputUpdateCategory, 'id'>) {
    const findCategory = await this.db.findOne({
      where: { id },
    });

    if (findCategory) {
      const materialToSave = this.db.merge(findCategory, { ...data });
      return await this.db.save(materialToSave);
    }
  }

  async delete(id: number) {
    const categoryToDelete = await this.db.findOne({
      where: { id },
    });
                                    const categoryToDelete: Category | null
    console.log('categoryToDelete', categoryToDelete);
    if (!categoryToDelete) {
      throw new Error("Le matériel n'existe pas!");
    }

    return await this.db.remove(categoryToDelete);
  }
}

export default CategoryServices;
```

Ici, **TypeORM** permet de manipuler les données de manière efficace grâce à des méthodes comme **find, findOne, create, save, merge, et remove**. Cela permet de gérer les opérations CRUD de manière fluide et cohérente avec les entités définies, tout en respectant les relations entre les tables.

4. Intégration avec GraphQL et Apollo

Enfin, **TypeORM** est intégré avec **GraphQL** et **Apollo Server** pour faciliter les requêtes et mutations sur la base de données.

Les **resolvers** GraphQL sont utilisés pour gérer ces opérations côté serveur. Ils permettent de récupérer les entités définies via **TypeORM** pour les requêtes (queries) et de manipuler les données pour les mutations (création, mise à jour et suppression). Chaque resolver est lié à une opération spécifique de l'API GraphQL, assurant une interaction fluide entre le front-end et le back-end.

Par exemple, le **resolver** pour la gestion des catégories définit des requêtes comme **listCategories** pour récupérer toutes les catégories, ou **findCategoryById** pour en obtenir une par son identifiant. De plus, les mutations **createCategory**, **updateCategory** et **deleteCategory** permettent de gérer la création, la mise à jour et la suppression des catégories en interagissant avec les méthodes CRUD fournies par

le service **CategoryServices**.

src > resolvers > TS category.resolver.ts > CategoryResolver

```ts
import { Arg, Mutation, Query, Resolver } from 'type-graphql';
import {
  InputCreateCategory,
  InputUpdateCategory,
} from '../entities/Category.entity';
import CategoryServices from '../services/category.service';
import Category from '../entities/Category.entity';

@Resolver()
export default class CategoryResolver {
  @Query(() => [Category])
  async listCategories() {
    const category: Category[] = await new CategoryServices().list();
    return category;
  }

  @Query(() => Category)
  async findCategoryById(@Arg('id') id: string) {
    const categories: Category = await new CategoryServices().findById(+id);
    return categories;
  }

  @Mutation(() => Category)
  async createCategory(@Arg('infos') infos: InputCreateCategory) {
    const result: Category = await new CategoryServices().create(infos);
    console.log('RESULT', result);
    return result;
  }

  @Mutation(() => Category)
  async updateCategory(@Arg('infos') infos: InputUpdateCategory) {
    const { id, ...otherData } = infos;
    const categoryToUpdate = await new CategoryServices().update(
      +id,
      otherData,
    );
    return categoryToUpdate;
  }

  @Mutation(() => Category)
  async deleteCategory(@Arg('id') id: string) {
    const categories: Category = await new CategoryServices().delete(+id);
    return categories;
  }
}
```

Cela permet une interaction transparente entre le front-end et le back-end via l'API

GraphQL, tout en facilitant la gestion des entités et des données dans la base de données **PostgreSQL** à l'aide de **TypeORM**.

Architecture MVC

Mon projet suit une architecture **MVC**, qui permet de structurer et d'organiser le code en trois couches distinctes : le Modèle, la Vue et le Contrôleur.

Modèle

Le Modèle est responsable de la gestion des données et de la logique métier. Dans cette architecture, cette partie est gérée principalement via **GraphQL**, **TypeGraphQL**, et les **Resolvers** :

- Les **Resolvers** jouent un rôle central en **TypeGraphQL**. En tant que classes définies dans le Modèle, ils encapsulent la logique pour interagir avec la base de données, gérer les opérations CRUD (création, lecture, mise à jour, suppression) et définir les schémas GraphQL.
- Les **Entités** (comme **User**) définissent la structure des données, tandis que les **Services** (comme **UserServices**) implémentent la logique métier nécessaire pour manipuler ces données.

Vue

La Vue est représentée par **Next.js**, qui gère l'interface utilisateur :

- **Next.js** est responsable de l'affichage des pages et des composants React, ainsi que de la mise à jour dynamique des données en fonction des réponses des requêtes GraphQL.
- La Vue capture également les actions de l'utilisateur, telles que les soumissions de **formulaires**, et envoie ces données au serveur pour traitement.

Contrôleur

- Le Contrôleur orchestre les interactions entre la Vue et le Modèle, et est géré par **Express** :
- **Express** prend en charge les requêtes HTTP, les middlewares, ainsi que la gestion des sessions et des cookies JWT. Il assure que les requêtes sont correctement dirigées vers les resolvers et que les réponses appropriées sont renvoyées au client.

Cette séparation en MVC permet une organisation claire et une maintenance facilitée de l'application en isolant les responsabilités liées aux données (Modèle), à l'affichage (Vue), et aux interactions avec le client (Contrôleur).

6.4- Authentification

L'authentification dans l'application est mise en œuvre en utilisant plusieurs techniques de sécurité essentielles, notamment le **hachage** des mots de passe, l'utilisation de **JWT** (JSON Web Token), la gestion des cookies, et l'établissement d'un contexte utilisateur.

Hachage des mots de passe

Les mots de passe des utilisateurs sont sécurisés grâce à la librairie **argon2** pour le hachage avant leur stockage dans la base de données. Le hachage est appliqué à l'aide de middlewares **TypeORM**, tels que **@BeforeInsert** et **@BeforeUpdate**, qui exécutent le hachage chaque fois qu'un nouveau mot de passe est inséré ou mis à jour. Cela permet de garantir que même en cas de compromission de la base de données, les mots de passe restent illisibles.

User.entity.ts :

```
type ROLE = 'ADMIN' | 'CLIENT';

@ObjectType()
@Entity()
export default class User {
  @BeforeInsert()
  @BeforeUpdate()
  protected async hashPassword() {
    if (!this.password.startsWith('$argon2')) {
      this.password = await argon2.hash(this.password);
    }
  }
```

Le hachage d'un mot de passe consiste à transformer un mot de passe en clair en une chaîne de longueur fixe à l'aide d'un algorithme de hachage. Cette transformation est toujours identique pour une même entrée, ce qui rend les mots de passe vulnérables aux attaques, comme celles par **force brute**. Le hachage seul ne suffit donc pas à protéger efficacement un mot de passe.

De plus, le hachage sécurise uniquement les mots de passe stockés (dans la base de données donc), mais pas leur transmission lors de l'envoi par l'utilisateur. Des algorithmes comme MD5 et SHA1, conçus pour être rapides, sont aujourd'hui obsolètes et facilement cassables par des attaques par force brute.

Amélioration possible :

Pour renforcer la sécurité, on utilise un **salt**, une donnée aléatoire ajoutée au mot de passe avant le hachage. Cela rend le mot de passe beaucoup plus difficile à cracker via des attaques par dictionnaire ou des tables arc-en-ciel (Rainbow tables), qui tentent de faire correspondre des hachages prédéfinis à des mots de passe. En générant un salt unique pour chaque utilisateur, même deux utilisateurs avec le

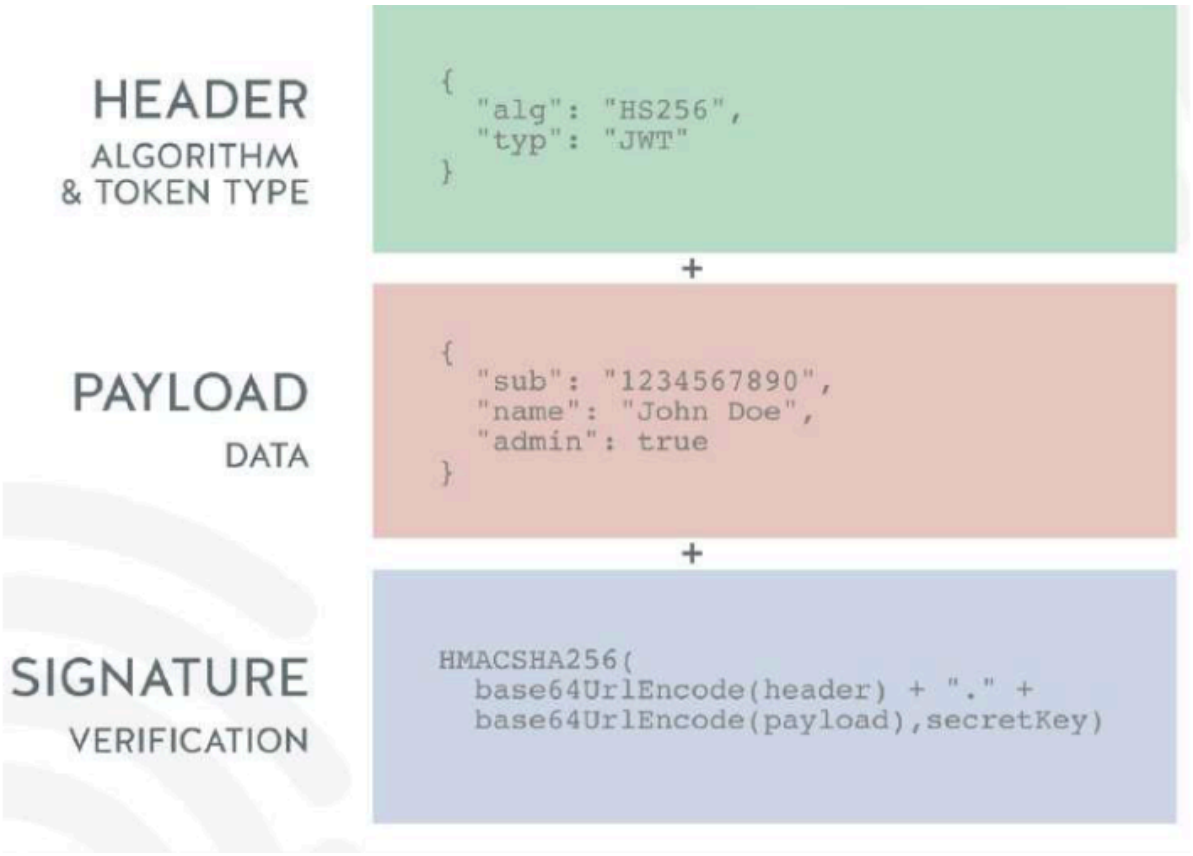
même mot de passe auront des hachages totalement différents dans la base de données.

L'algorithme **bcrypt**, par exemple, intègre un salt et est "adaptatif", permettant d'augmenter progressivement les itérations pour ralentir le processus de hachage. Cela rend bcrypt particulièrement efficace contre les attaques par force brute, car plus les itérations augmentent, plus le hachage devient difficile à casser.

JWT et gestion des cookies

Les **JSON Web Tokens (JWT)** sont utilisés pour **authentifier** les utilisateurs lors de leur connexion et tout au long de leur session. Après la vérification du mot de passe via **argon2**, un JWT est généré en utilisant la bibliothèque **jose**. Ce token contient des informations non sensibles, telles que l'adresse email et le rôle de l'utilisateur, qui sont placées dans le payload du JWT. Cela permet d'éviter de vérifier à chaque requête le rôle en base de données, réduisant ainsi la charge de travail du serveur.

Le JWT est composé de trois parties : le **header**, le **payload** et la **signature**. Le header est automatiquement fourni par la librairie **jose**, spécifiant l'**algorithme de chiffrement** (dans ce cas HS256). La signature est générée en combinant le header, le payload et une clé secrète (**process.env.SECRET_KEY**). Cette clé secrète est essentielle à la sécurité du JWT, car elle permet de vérifier que le token n'a pas été altéré. Elle doit rester strictement confidentielle et ne jamais être divulguée.



Le JWT est ensuite stocké dans un **cookie HTTP-only**, ce qui empêche son accès par des **scripts JavaScript malveillant**s, protégeant ainsi l'application des attaques **XSS**

(Cross-Site Scripting). Grâce à ce cookie, l'utilisateur reste authentifié à chaque requête HTTP, et si un token valide est présent, les informations qu'il contient (comme l'email et le rôle) sont extraites pour définir le **contexte de l'utilisateur**.

Les JWT sont "**stateless**", c'est-à-dire que les informations de session ne sont pas stockées côté serveur. Cela offre l'avantage de ne pas consommer de mémoire serveur pour la gestion des sessions et permet une authentification fluide dans une architecture en microservices. En partageant la même clé secrète entre différentes applications, il est possible d'utiliser un seul token pour s'authentifier sur plusieurs services sans que chaque service n'ait besoin de recréer une session utilisateur.

Enfin, une date d'expiration est ajoutée au JWT avec la **méthode .setExpirationTime()** pour limiter sa durée de validité (ici à 2 heures).

user.resolver.ts

```ts
export interface Payload {
  email: string;
}
```

```ts
@Query(() => Message)
async login(@Arg('infos') infos: InputLogin, @Ctx() ctx: MyContext) {
  const user = await new UserServices().findUserByEmail(infos.email);
  if (!user) {
    throw new Error('Vérify your login information');
  }

  const isPasswordValid = await argon2.verify(user.password, infos.password);
  const m = new Message();
  if (isPasswordValid) {
    const token = await new SignJWT({ email: user.email, role: user.role })
      .setProtectedHeader({ alg: 'HS256', typ: 'jwt' })
      .setExpirationTime('2h')
      .sign(new TextEncoder().encode(`${process.env.SECRET_KEY}`));

    const cookies = new Cookies(ctx.req, ctx.res);
    cookies.set('token', token, { httpOnly: true });

    console.log(token);
    console.log(cookies);
    m.message = 'Welcome!';
    m.success = true;
  } else {
    m.message = 'Verify your login information';
    m.success = false;
  }
  return m;
}

@Query(() => Message)
async logout(@Ctx() ctx: MyContext) {
  if (ctx.user) {
    const cookies = new Cookies(ctx.req, ctx.res);
    cookies.set('token'); //sans valeur, le cookie token sera supprimé
  }
  const m = new Message();
  m.message = 'Vous avez été déconnecté';
  m.success = true;

  return m;
}
```

Contexte de l'utilisateur

Le **contexte** utilisateur est déterminé dans le **middleware Express**, où le JWT est extrait des cookies, vérifié, et l'utilisateur correspondant est récupéré à partir de la base de données. Ce contexte est ensuite accessible dans les résolveurs GraphQL pour restreindre l'accès à certaines ressources ou pour personnaliser les réponses en fonction de l'utilisateur connecté.

index.ts

```ts
export interface MyContext {
  req: express.Request;
  res: express.Response;
  user: User | null;
}
```

```ts
expressMiddleware(server, {
  context: async ({ req, res }) => {
    let user: User | null = null;
    const cookies = new Cookies(req, res);
    const token = cookies.get('token');
    console.log("Token from cookies:", token);  // Log du token récupéré

    if (token) {
      try {
        const verify = await jwtVerify<Payload>(
          token,
          new TextEncoder().encode(process.env.SECRET_KEY),
        );
        console.log("Token verified:", verify.payload);  // Log du payload du token
        user = await new UserService().findUserByEmail(verify.payload.email);
        console.log("User found:", user);  // Log de l'utilisateur récupéré
      } catch (err) {
        console.log("Error verifying token:", err);
      }
    } else {
      console.log("No token found in cookies");
    }

    return { req, res, user };
  },
}),
```

Autorisation avec TypeGraphQL

En complément, pour plus de sécurité, nous avons mis en place un système d'authentification et de gestion des autorisations à l'aide du décorateur **@Authorized** de **TypeGraphQL** et du **customAuthChecker**. Ce système permet de sécuriser les

résolveurs et de contrôler l'accès aux différentes fonctionnalités de l'application en fonction des rôles des utilisateurs.

Le décorateur @**Authorized** est utilisé pour restreindre l'accès à certains résolveurs ou champs. Par exemple, nous avons défini **@Authorized("ADMIN")** sur certains résolveurs sensibles, comme la **création, édition ou suppression de matériel**, afin que seuls les administrateurs puissent y accéder. Dans d'autres cas, comme la consultation des réservations, nous avons utilisé @Authorized("USER", "ADMIN") pour autoriser à la fois les utilisateurs et les administrateurs. Si un résolveur n'a pas besoin de restriction spécifique, mais qu'il doit s'assurer que l'utilisateur est simplement authentifié, nous utilisons @Authorized() pour garantir que l'accès est réservé aux utilisateurs connectés.

```
@Authorized('ADMIN')
@Mutation(() => Material)
async createMaterial(@Arg('infos') infos: InputCreateMaterial) {
  const result: Material = await new MaterialServices().create(infos);
  console.log('RESULT', result);
  return result;
}

@Authorized('ADMIN')
@Mutation(() => Material)
async updateMaterial(@Arg('infos') infos: InputUpdateMaterial) {
  const { id, ...otherData } = infos;
  const materialToUpdate = await new MaterialServices().update(id, otherData);
  return materialToUpdate;
}
@Authorized('ADMIN')
@Mutation(() => Material)
async deleteMaterial(@Arg('id') id: string) {
  const material: Material = await new MaterialServices().delete(id);
  return material;
}
```

```
@Resolver()
export default class ReservationResolver {
  @Authorized('USER', 'ADMIN')
  @Query(() => [Reservation])
  async listReservations() {
    const users: Reservation[] = await new ReservationServices().list();
    return users;
  }
```

Le décorateur fonctionne en tandem avec le **customAuthChecker**, un **middleware créé avec le AuthChecker de TypeGraphQL,** qui vérifie si l'utilisateur est connecté et possède les rôles nécessaires. Si l'utilisateur est connecté mais que son rôle ne correspond pas à ceux attendus par le décorateur, l'accès est refusé. Par exemple, dans le cas où le décorateur @Authorized("ADMIN") est utilisé, le customAuthChecker s'assure que le rôle de l'utilisateur est bien "ADMIN". Si aucun rôle spécifique n'est requis (par exemple pour l'accès à un profil utilisateur via @Authorized()), le customAuthChecker laisse passer tout utilisateur authentifié.

index.ts

```ts
import { customAuthChecker } from "./lib/authChecker";

export interface MyContext {
  req: express.Request;
  res: express.Response;
  user: User | null;
}

export interface Payload {
  email: string;
}

const app = express();
const httpServer = http.createServer(app);

async function main() {
  const schema = await buildSchema({
    resolvers: [
      MaterialResolver,
      UserResolver,
      ReservationResolver,
      CategoryResolver,
    ],
    validate: false,
    authChecker: customAuthChecker,
  });
```

CORS (Cross-Origin Resource Sharing)

Afin de sécuriser les communications entre le client et le serveur, les paramètres **CORS** sont configurés pour restreindre l'origine des requêtes à une liste spécifique de domaines de confiance, garantissant ainsi que seules des applications approuvées peuvent interagir avec l'API.

```ts
app.use(
  '/',
  cors<cors.CorsRequest>({
    origin:
      [
        'http://localhost:3000',
        'http://localhost:4005',
        'https://studio.apollographql.com',
        "https://1123-jaune-1.wns.wilders.dev/",
        "https://staging.1123-jaune-1.wns.wilders.dev/"
      ],
    credentials: true,
  }),
```

6.5-Front-end

Next

Pour commencer, il est nécessaire de faire un zoom sur **Next.js**, le framework utilisé pour notre projet.

Next.js est une extension de **React.js**, conçue pour construire des interfaces utilisateur **interactives** et **réactives** pour les applications web :

- Repose sur le concept de **composants réutilisables**, permettant de diviser l'interface utilisateur en petits morceaux indépendants.

- Permet également le développement d'applications à **état**, où les composants peuvent être manipulés et mis à jour en réponse aux événements utilisateur.

- Utilise le '**Virtual DOM**' pour une mise à jour efficiente de l'interface utilisateur. Lorsqu'un changement survient, il compare cette version virtuelle avec le **DOM réel**, mettant à jour uniquement les différences et évitant ainsi le rechargement complet de la page. Bien que Next.js soit souvent utilisé pour créer des applications multi-pages avec des fonctionnalités avancées comme le rendu côté serveur (**SSR**) et le rendu statique (**SSG**), nous avons choisi de rester en mode de rendu côté client (**CSR**) pour notre projet. Cette approche nous a permis de bénéficier d'une navigation fluide et rapide, tout en simplifiant la structure de notre application grâce à **la gestion automatique des routes basée sur les fichiers et les dossiers**, ce qui a été un facteur déterminant dans notre choix.

- Syntaxe JavaScript plus simple grâce au format **JSX** (pour JavaScript XML), permettant l'intégration d'éléments HTML et de composants React dans un seul code, qui est ensuite traduit en JavaScript pur lors de la compilation pour être exécuté dans le navigateur.

Pour renforcer la robustesse et la maintenabilité de notre code, nous avons également utilisé **TypeScript**. TypeScript apporte un typage statique à JavaScript, ce

qui permet de détecter les erreurs de type à la compilation et d'améliorer la **lisibilité, la fiabilité** et la **documentation** du code.

La configuration

Pour configurer le front-end de notre application Next.js, nous avons utilisé trois fichiers principaux : **index.tsx**, **_app.tsx**, et **_document.tsx**.

- Dans index.tsx, nous avons défini la page d'accueil de notre application en important et en utilisant le composant **HomepageConten**t, qui contient le contenu principal de la page sur laquelle arrivera l'utilisateur quand il accèdera à l'application.

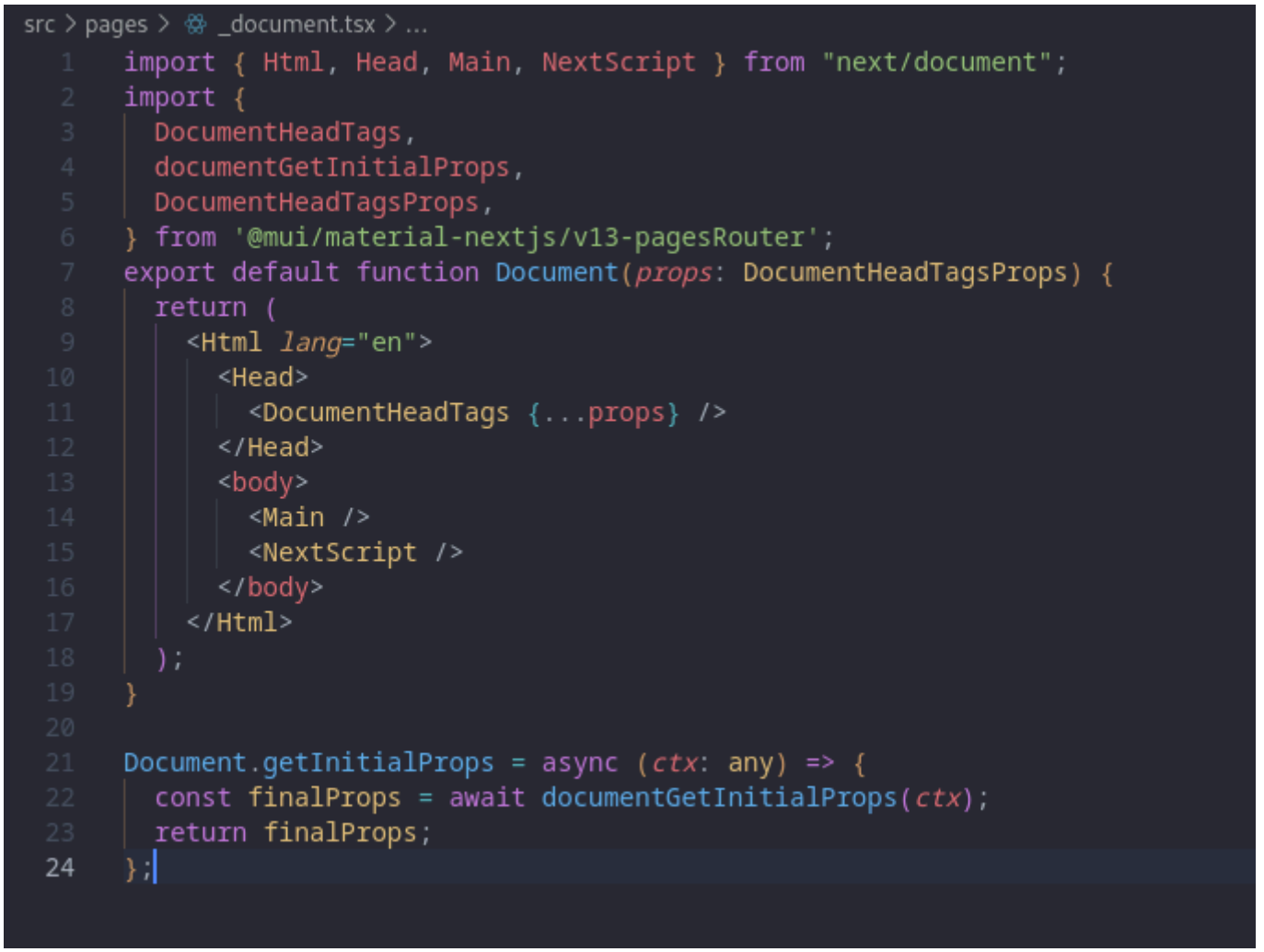
```
import HomepageContent from "@/pages/HomepageContent";

export default function Home() {
  return (
    <main>

      <div>
        {/*<MainNav />*/}
        <HomepageContent />
      </div>
    </main>
  );
}
```

- index.tsx est rendu à l'intérieur du composant **App,** défini dans _app.tsx, qui gère la configuration globale de l'application. Dans _app.tsx, nous avons configuré un **client Apollo** pour gérer les **requêtes GraphQL** et utilisé **AppCacheProvider** pour optimiser les performances des composants Material-UI. Nous avons également **désactivé le rendu côté serveur (SSR)** pour cette application. Nous avons inclus le composant **MainNav** dans _app.tsx pour garantir que la navigation principale soit présente sur toutes les pages de notre application, tout en permettant de définir si on se trouve sur la page d'accueil ou non, ce qui nous servira pour adapter la Nav en fonction du besoin.

src > pages > _app.tsx > App

```tsx
import "@/styles/globals.css";
import '@fontsource/roboto/300.css';
import '@fontsource/roboto/400.css';
import '@fontsource/roboto/500.css';
import '@fontsource/roboto/700.css';

import type { AppProps } from "next/app";
import { ApolloClient, InMemoryCache, ApolloProvider } from "@apollo/client";
import { AppCacheProvider } from '@mui/material-nextjs/v13-pagesRouter';
import dynamic from "next/dynamic";
import MainNav from "@/components/MainNav/MainNav";
import { useRouter } from "next/router";

const client = new ApolloClient({
  uri: "http://localhost:4005",
  cache: new InMemoryCache({
    addTypename: false
  }),
});



function App({ Component, pageProps, ...props }: AppProps) {
  const router = useRouter();
  {/* Vérifie si on est sur la page d'acceuil */}
  const isHomePage = router.pathname === '/';

  return (
    <>
    <AppCacheProvider {...props}>
      <ApolloProvider client={client}>
        {/* On passe la prop isHomePage à MainNav */}
        <MainNav isHomePage={isHomePage} />
          <Component {...pageProps} />
      </ApolloProvider>
      </AppCacheProvider>
    </>
  );
}

// Disabling SSR
export default dynamic(() => Promise.resolve(App), { ssr: false });
```

- Enfin, dans **_document.tsx**, nous avons personnalisé le **document HTML** généré par Next.js en injectant les balises nécessaires pour **Material-UI** dans la section <head> du document.



Apollo Client et GraphQL-Codegen

Nous avons utilisé **Apollo Client** et **GraphQL Codegen** pour simplifier et optimiser la gestion des données entre le front-end et notre API GraphQL.

- **Apollo Client** est une bibliothèque qui nous permet de communiquer efficacement avec le serveur GraphQL. Elle facilite l'envoi des requêtes et des mutations, tout en intégrant des états de chargement, de succès et d'erreur directement dans nos composants.
- **GraphQL Codegen**, quant à lui, génère automatiquement des types TypeScript et des hooks basés sur nos schémas GraphQL, garantissant la **fiabilité** du typage en **réduisant les erreurs** tout en améliorant la **productivité**.

Tout d’abord, nous définissons nos requêtes et nos mutations dans des fichiers dédiés, servant de base pour Apollo Client et Codegen, comme par exemple ici :

src/request/queries/auth.queries.ts

src/requests/queries/categories.queries

```
src > requests > queries > TS auth.queries.ts > ...
import { gql } from "@apollo/client";

export const LOGIN = gql`
  query Login($infos: InputLogin!) {
    login(infos: $infos) {
      success
      message
    }
  }
`;

export const LOGOUT = gql`
  query Logout {
    logout {
      success
      message
    }
  }
`;
```

```
src > requests > queries > TS categories.queries.ts > ...
import { gql } from "@apollo/client";

export const LIST_CATEGORIES = gql`
  query ListCategories {
    listCategories {
      id
      name
      slug
    }
  }
`;

export const FIND_CATEGORY = gql`
  query FindCategoryById($id: String!) {
    findCategoryById(id: $id) {
      id
      materials {
        id
        name
        price
        image
        slug
      }
    }
  }
`;
```

Ensuite nous configurons Codegen afin qu'il utilise à la fois nos schémas disponibles sur le serveur et dans les requêtes et mutations définis dans nos fichiers *.queries.ts* et *.mutation.ts*.

Codegen génère alors un fichier contenant les types et les hooks associés.

Prenons en exemple le composant **Login** pour la connexion et la **MainNav** pour la navigation.

Login.tsx

Ici, nous utilisons le hook **useLazyQuery** avec la requête **LOGIN** (définie dans notre fichier **auth.queries.ts**) pour authentifier les utilisateurs. Lors de la soumission du formulaire, **useLazyQuery** déclenche la requête avec les informations d'identification (email et mot de passe) et renvoie la réponse du serveur. Cela nous permet de gérer l'état d'authentification de manière **asynchrone**, en affichant des messages de succès ou d'erreur en fonction de la réponse reçue. Contrairement à **useQuery**, qui exécute la requête dès que le composant est monté, **useLazyQuery attend une action explicite** (ici la soumission du formulaire).

```tsx
src > pages > auth > login.tsx > LoginProps > showRegister
import { Box, Button, TextField, Typography } from '@mui/material';
import { useLazyQuery } from '@apollo/client';
import { useState } from 'react';
import { LOGIN } from '@/requests/queries/auth.queries';

interface LoginProps {
  showRegister: () => void;
  closeModals: () => void;
}

const Login: React.FC<LoginProps> = ({ showRegister, closeModals }) => {

  const [formState, setFormState] = useState({
    email: '',
    password: '',
  });

  const [loginUser, { data, loading, error }] = useLazyQuery(LOGIN, {
    onCompleted: (data) => {
      console.log("Mutation completed:", data);
      closeModals();
    },
    onError: (error) => {
      console.error("Mutation error:", error);
    },
  });
```

```tsx
const handleSubmit = async (e: React.FormEvent) => {
  e.preventDefault();
  try {
    const response = await loginUser({
      variables: {
        infos: {
          email: formState.email,
          password: formState.password,
        },
      },
    });
    console.log("Response from server:", response);
  } catch (e) {
    console.error("Submission error:", e);
  }
};
```

MainNav.tsx

Dans ce composant, nous utilisons des **hooks** générés par **GraphQL-Codegen**, tels que **useListCategoriesQuery** et **useListMaterialsQuery**, pour récupérer les données

des catégories et des matériels. Ces hooks sont **automatiquement** créés à partir de nos **schémas** et **requêtes** GraphQL, garantissant un **typage** TypeScript **strict** et une intégration fluide avec **Apollo Client**. En utilisant ces hooks, nous pouvons facilement récupérer et afficher la liste de matériels dans le champ de recherche et les catégories sous forme de boutons de navigation dynamiques. Cela permet à notre application de gérer efficacement les données.

```
  //On utilise les hooks de codegen pour récupérer les données

const { loading: loadingMaterial, error: errorMaterial, data: dataMaterial } = useListMaterialsQuery({
  fetchPolicy: "no-cache"
});

const { loading: loadingCategories, error: errorCategories, data: dataCategories } = useListCategoriesQuery({
  fetchPolicy: "no-cache"
});

if (loadingMaterial) return <p>Loading...</p>;
if (errorMaterial) return <p>Error: {errorMaterial.message}</p>;

if (loadingCategories) return <p>Loading...</p>;
if (errorCategories) return <p>Error: {errorCategories.message}</p>;

//variable pour itérer sur les matériels et les afficher dans l'autocomplete
const materials = dataMaterial?.listMaterials?.map((material) => ({
  label: material.name,
  id: material.id,
})) || [];
```

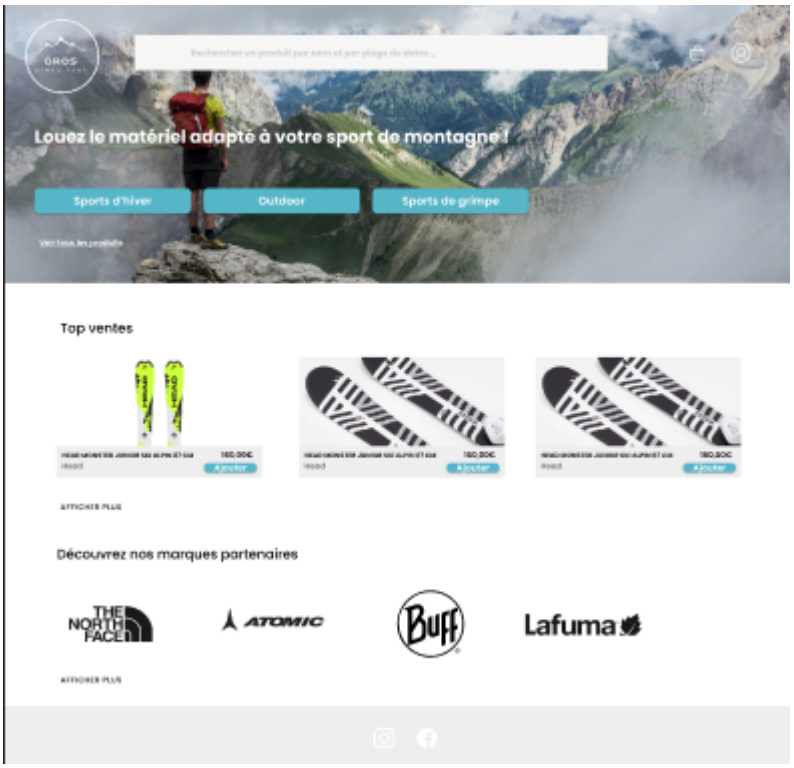
Dans cette navigation principale, nous adaptons **dynamiquement** l'interface en fonction de la page sur laquelle l'utilisateur se trouve. Un **état** local **isHomePage** (de type booléen) est défini en fonction du **pathname** fourni par le **routeur**. Le hook **useEffect** surveille les changements de route et **met à jour cet état**. Cela permet d'afficher ou de masquer les boutons de catégories selon que l'utilisateur est sur la page d'accueil ou non, en s'appuyant sur les données récupérées par **useListCategoriesQuery**.

MainNav.tsx

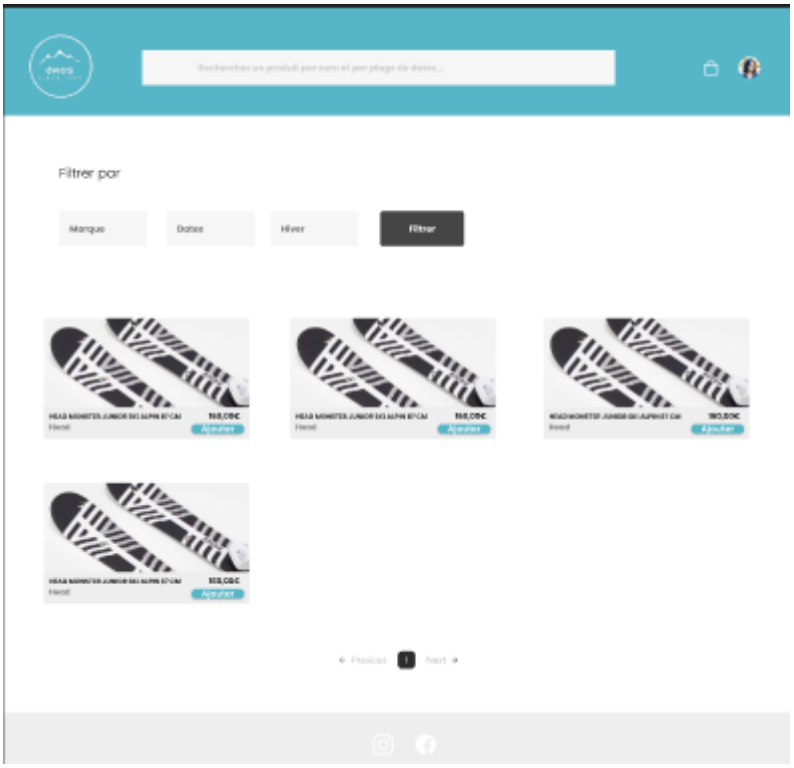
```
const [isHomePage, setIsHomePage] = useState<boolean>(true); // State pour savoir si on est sur la page d'accueil

const router = useRouter();

// Hook pour définir si on est sur la page d'accueil et mettre à jour le state
useEffect(() => {
  if (router.pathname === "/") {
    setIsHomePage(true);
  }
  else {
    setIsHomePage(false);
  }
}, [router]);
```

```jsx
{/* Condition pour afficher les boutons de navigation des catégories, selon si on est sur la page d'accueil ou non */}
{isHomePage && (
  <>
{/* Boutons de navigation dynamiques en itérant sur les catégories et en utilisant un composant réutilisable */}
    <div className={styles.navButtons}>
      <ul>
      {dataCategories?.listCategories?.map((category: Category) => (
        <li key={category.id}>
          <CategoryButton name={category.name} id={category.id} />
        </li>
      ) )}
      </ul>
    </div>
  </>
)}
```

Accueil



Catégorie “Sports d’hiver”



Lorsque l'utilisateur est sur la page d'accueil, ces boutons de catégorie sont affichés, permettant une **navigation directe** vers les différentes catégories de matériels. Ces boutons sont générés **dynamiquement** en **itérant** sur les catégories récupérées via la **requête GraphQL**. Chaque bouton redirige l'utilisateur vers une page de liste de matériels spécifique à la catégorie sélectionnée.

Le fichier **category/[id].tsx** dans Next.js utilise le routage dynamique pour afficher les détails d'une catégorie spécifique. Le segment [id] dans le chemin de l'URL permet de capturer l'identifiant de la catégorie et de le passer en tant que paramètre à la page. Cette page utilise une requête GraphQL pour récupérer les matériels associés à la catégorie sélectionnée et les affiche dans une liste. Chaque matériel est rendu à l'aide d'un **composant MaterialCard**, qui présente les informations détaillées du matériel.

src > pages > categories > [id].tsx > ...

```tsx
import MaterialCard from "@/components/MaterialCard";
import { Material, useFindCategoryByIdQuery } from "@/types/graphql";

import { useRouter } from "next/router";
import React from "react";

const CategoryPage = () => {
  const router = useRouter();
  const { id } = router.query;

  const { loading, error, data } = useFindCategoryByIdQuery({
    variables: { id: id as string },
    skip: !id,
  });
  console.log("coucou", data);
  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error.message}</p>;

  return (
    <div>
      <h1>Liste des matériels</h1>
      <ul>
        {data?.findCategoryById?.materials?.map((material) => (
          <li key={material.id}>
            <MaterialCard material={material} />
          </li>
        ))}
      </ul>
    </div>
  );
};

export default CategoryPage;
```

Les boutons de catégorie sont implémentés à l'aide d'un composant réutilisable **CategoryButton**, qui redirige l'utilisateur vers la page de la catégorie correspondante.

src > components > CategoryButton.tsx > default

```tsx
import { Button } from '@mui/material';
import React from 'react';

function CategoryButton({name, id}: {name: string, id: string}) {
    return (
        <div>
          <Button variant="contained" href={`/categories/${id}`}>{name}</Button>
        </div>
    );
}

export default CategoryButton;
```

Material-UI

Material-UI (MUI) est une **bibliothèque** de **composants React** qui permet de concevoir des interfaces utilisateur modernes et responsives. Conçue selon les principes du **Material Design de Google**, MUI fournit une large gamme de composants pré-stylisés comme des boutons, des champs de texte, des modales, et des icônes, tout en permettant de personnaliser le thème pour répondre aux besoins spécifiques de chaque projet. En facilitant la création d'interfaces uniformes et accessibles, elle **réduit considérablement le temps nécessaire** pour développer des éléments d'interface de qualité.

Dans notre projet, MUI est utilisé pour structurer et styliser plusieurs parties de l'application, comme le formulaire de connexion dans le fichier **Login.tsx**. Par exemple, les composants **TextField** et **Button** de MUI sont employés pour créer un formulaire d'authentification élégant, tandis que le composant **Box** permet d'encadrer le tout avec des marges et un style cohérent.

```
return (
  <Box sx={{ maxWidth: 500, mx: 'auto', mt: 4, p: 2, border: '1px solid #ccc', borderRadius: 2, backgroundColor:'white' }}>
    <Typography variant="h4" style={{color:'black', textAlign:'center', marginBottom:'2rem'}}>Se connecter à votre compte</Typography>
    <form onSubmit={handleSubmit}>
      <TextField
        type="email"
        name="email"
        label="Email"
        value={formState.email}
        onChange={handleChange}
        required
        fullWidth
        margin="normal"
      />
      <TextField
        type="password"
        name="password"
        label="Password"
        value={formState.password}
        onChange={handleChange}
        required
        fullWidth
        margin="normal"
      />
      <Button type="submit" variant="contained" color="primary" disabled={loading} fullWidth style={{marginBottom:'2rem'}}>
        {loading ? 'Submitting...' : 'Login'}
      </Button>
      {error && <Typography style={{color:'red', textAlign:'center'}}>Error: {error.message}</Typography>}
    </form>
x to add a breakpoint  aphy onClick={showRegister} style={{color:'black', textAlign:'center', cursor: 'pointer'}}>Vous ne possédez pas de compte ? Inscrivez-vous</Typography>
    {data && data.login.success && <Typography style={{color:'green', textAlign:'center'}}>Success! {data.login.message}</Typography>}
    {data && !data.login.success && <Typography style={{color:'red', textAlign:'center'}}>Failed! {data.login.message}</Typography>}
  </Box>
);
```

Le composant **Box** permet de définir un cadre visuellement agréable et centré pour le formulaire, et le **Typography** est utilisé pour styliser le titre. Les composants MUI assurent la cohérence de l'apparence tout en gérant automatiquement les interactions utilisateur comme les états de chargement via **Button**.

Dans **MainNav.tsx**, des composants comme **Autocomplete**, **Button**, et **Modal** sont utilisés pour rendre la navigation intuitive et interactive, en créant par exemple un champ de recherche pour les matériels et une modale pour le login.

```jsx
{/*Barre de recherche des materiels*/}
<div className={styles.navContent}>
  <Autocomplete
    disablePortal
    id="combo-box-demo"
    options={materials}
    onChange={handleOptionSelect}
    sx={{ width: 300 }}
    className={styles.input}
    renderInput={(params) => <TextField {...params} label="Rechercher un matériel" />}
  />
  {/*Titre*/}
  <div className={styles.title}>
    <Typography>Louez le matériel adapté à votre sport de montagne !</Typography>
```

```jsx
<div className={styles.icons}>
  <Button><ShoppingCartIcon/></Button>
  <Button onClick={openLoginModal}><AccountCircleIcon/></Button>
  <Modal
    open={isModalOpen}
    onClose={() => setIsModalOpen(false)}
    aria-labelledby="modal-modal-title"
    aria-describedby="modal-modal-description"
    className={styles.modal}
    style={{display:'flex',alignItems:'center',justifyContent:'center', border: '1px solid black'}}
  >
    {showRegisterModal ? (
      <Register showRegister={openLoginModal} />
    ) : (
      <Login showRegister={openRegisterModal} closeModals={closeAllModals}/>
    )}
  </Modal>
</div>
```

Grâce à la flexibilité de MUI, **l'intégration** avec les **hooks** Apollo permet de lier l'interface graphique aux données de l'application de manière fluide.

Back-office

6.6-Test

Les tests jouent un rôle crucial dans la maintenance et l'amélioration d'une application au fil du temps, en garantissant que chaque nouvelle fonctionnalité ou modification n'introduit pas de régressions ou de bugs. Ils se déclinent en plusieurs types, chacun ayant un objectif spécifique :

Tests unitaires : Ils concernent une unité de code, souvent une seule fonction ou méthode. Leur but est de valider que chaque partie isolée de l'application fonctionne comme prévu. Les tests unitaires sont rapides à exécuter et ne

nécessitent pas d'environnement complexe. Ils constituent la majorité des tests implémentés par les développeurs.

- **Tests d'intégration** : Ils vérifient l'interaction entre plusieurs composants ou fonctions, permettant de tester leur intégration dans un sous-ensemble de l'application. Par exemple, un test d'intégration peut s'assurer qu'un composant et son enfant interagissent correctement, offrant un compromis entre tests unitaires et tests end-to-end.
- **Tests end-to-end (E2E)** : Ces tests couvrent le flux complet d'une application, en simulant les actions de l'utilisateur. Ils testent l'application dans son ensemble, depuis l'interface jusqu'à la base de données. Ils sont plus lents et moins précis que les tests unitaires, car ils couvrent une plus grande surface fonctionnelle, mais ils sont utiles pour s'assurer que l'application fonctionne dans des scénarios réels.

Dans notre projet Oros, nous avons mis en place des tests pour vérifier la récupération des matériaux via notre API GraphQL. Nous avons utilisé **Jest**, qui est très utilisé dans le développement de logiciels pour effectuer des **tests unitaires**, des **tests d'intégration** et des **tests snapshot**. Plus précisément, nous avons utilisé **ts-jest** qui est un **préprocesseur TypeScript,** soit une **extension de Jest** spécifiquement conçue pour faciliter les tests de projets TypeScript.

Afin d'écrire un test il faut respecter certaines règles concernant le nommage du fichier. Jest cherche automatiquement les fichiers se terminant par test.(ts|tsx) - configuré dans jest.config.ts et plus particulièrement par le preset ts-jest.

Le test suivant simule la réponse du serveur à une requête **listMaterials**, en utilisant un **mock store** pour **émuler** la base de données. Cela nous permet de tester la requête sans dépendre d'une véritable base de données ou du back-end.


__tests__ > TS material.test.ts > materialsData > category
//backend/__tests/books-store.test.ts
import assert from 'assert';
import {
IMockStore,
addMocksToSchema,
createMockStore,
} from '@graphql-tools/mock';
import { ApolloServer } from '@apollo/server';
import { buildSchemaSync } from 'type-graphql';
import { printSchema } from 'graphql';
import { makeExecutableSchema } from '@graphql-tools/schema';
import MaterialResolver from '../src/resolvers/material.resolver';
import Material from '../src/entities/Material.entity';
const materialsData: Material[] = [
{
id: '1',
name: 'My Material 1',
category: {
id: 1,
name: 'Category 1',
materials: [],
slug: 'category-1',
},
description: 'Description 1',
image: 'image1.jpg',
initial_stock: 10,
price: 10.5,
slug: 'my-material-1',
},
{
id: '2',
name: 'My Material 2',
category: {
id: 2,
name: 'Category 2',
materials: [],
slug: 'category-2',
},
description: 'Description 2',
image: 'image2.jpg',
initial_stock: 20,
price: 20.5,
slug: 'my-material-2',
},
];

Dans ce test, nous définissons une requête **LIST_MATERIALS** pour interroger notre API et récupérer une liste de matériaux. Nous simulons cette liste de matériaux avec des données mockées stockées dans un **mock store**. Ce store est créé avec **createMockStore** via la librairie **@graphql-tools/mock**, qui permet de **simuler** une base de données en mémoire.

```
export const LIST_MATERIALS = `#graphql
query ListMaterials {
  listMaterials {
    id
  }
}
`;
type ResponseData = {
  books: Material[];
};

let server: ApolloServer;

const baseSchema = buildSchemaSync({
  resolvers: [MaterialResolver],
  authChecker: () => true,
});

const schemaString = printSchema(baseSchema);
const schema = makeExecutableSchema({ typeDefs: schemaString });

const resolvers = (store: IMockStore) => ({
  //resolvers est une fonction qui reçoit le store en argument!
  Query: {
    listMaterials() {
      return store.get('Query', 'ROOT', 'listMaterials');
    },
  },
});

beforeAll(async () => {
  const store = createMockStore({ schema });
  server = new ApolloServer({
    schema: addMocksToSchema({
      schema: baseSchema,
      store,
      resolvers,
    }),
  });

  store.set('Query', 'ROOT', 'listMaterials', materialsData);
});
```

Le **mock store** stocke des objets de données, comme les matériaux, que les résolveurs GraphQL peuvent interroger. Dans ce cas, **listMaterials** est configuré pour récupérer les matériaux à partir du store, au lieu d'une base de données réelle. Cela nous permet de simuler la logique de récupération de données sans dépendre d'un environnement complet de back-end. Le serveur Apollo est configuré pour utiliser un **schéma** GraphQL et des **résolveurs mockés** grâce à **@graphql-tools/mock**, qui renvoient les données du store.

Ce test vérifie ensuite que la réponse renvoyée par l'API contient bien les identifiants des matériaux attendus, prouvant que la requête listMaterials fonctionne correctement. Ce type de test d'intégration garantit que les résolveurs et le schéma GraphQL interagissent comme prévu avec la logique du store.

```
describe('Test sur les matériels', () => {
  it('Récupération des matériels depuis le store', async () => {
    const response = await server.executeOperation<ResponseData>({
      query: LIST_MATERIALS,
    });

    assert(response.body.kind === 'single');
    expect(response.body.singleResult.data).toEqual({
      listMaterials: [{ id: '1' }, { id: '2' }],
    });
  });
});
```

6.7-Intégration continue (CI) et déploiement continu (CD)

Pour simplifier le processus de déploiement et minimiser les manipulations manuelles, nous avons mis en place un **processus d'intégration et de déploiement continu** (CI/CD) en pré-production et en production. Ce processus repose sur une combinaison d'outils tels que **GitHub Actions**, les **webhooks**, **Docker**, **DockerHub**, et **Caddy** en tant que reverse proxy.

- GitHub Actions
  Lancé en 2019, GitHub Actions est une solution CI/CD intégrée aux repositories GitHub. Elle permet de déclencher des workflows basés sur des événements comme un push sur une branche. Dans mon projet, j'ai configuré deux workflows YAML pour :

    - Lancer automatiquement les tests
    - Créer et stocker des images Docker

- **Docker & DockerHub :** Les workflows GitHub Actions génèrent des **images Docker**, qui sont ensuite stockées sur **DockerHub**, un répertoire en ligne pour les conteneurs Docker. DockerHub permet de gérer facilement les images Docker en offrant des fonctionnalités pour **créer**, **tester**, **stocker** et **déployer** des conteneurs.

- **Caddy :** Nous utilisons **Caddy** comme **reverse proxy** pour gérer les requêtes HTTP vers notre domaine. Un reverse proxy est un serveur logiciel qui redirige toutes les requêtes entrantes vers d'autres serveurs, puis retourne la réponse au client. Ce système permet de répartir la charge des serveurs et d'optimiser la sécurité.
  En plus de gérer la redirection des requêtes, Caddy simplifie la gestion du **protocole HTTPS** (Hypertext Transfer Protocol Secure). HTTPS protège l'intégrité et la confidentialité des données échangées entre le client et le serveur grâce au protocole **Transport Layer Security (TLS)**, qui remplace l'ancien SSL. Ce protocole garantit trois niveaux clés de protection :
    - **Chiffrement** des données pour empêcher leur interception
    - **Intégrité des données** pour s'assurer qu'elles ne sont pas modifiées
    - **Authentification** pour garantir l'identité des parties communicantes

- **Webhooks :** Enfin, les **webhooks** sont configurés pour déclencher des scripts sur le serveur lorsqu'un événement spécifique survient, comme la mise à jour d'une branche. Cela permet d'automatiser des actions comme le déploiement ou la mise à jour d'une application sans intervention manuelle.

Nous avons donc intégré des workflows d'intégration continue (CI) pour automatiser les tests et le déploiement Docker de nos applications front-end et back-end. Ces workflows sont déclenchés à chaque push, assurant ainsi que le code est toujours testé et que les images Docker sont construites et poussées si les tests réussissent. En suivant la documentation officielle pour configurer les workflows, le fichier yaml doit se situer à la racine du projet dans les repositories « .github/workflows ».

Workflow Back-End

Le workflow back-end est divisé en deux étapes principales :

1. **Tests Jest** : Lors d'un push, les tests back-end sont exécutés via le script npm run test-ci. Ces tests permettent de vérifier le bon fonctionnement des différentes parties de l'application, garantissant que les changements apportés au code n'introduisent pas de régressions.
2. **Build et Push Docker** : Si les tests passent avec succès sur la branche dev, une image Docker de l'application back-end est construite et poussée sur DockerHub. Les secrets GitHub sont utilisés pour sécuriser les identifiants DockerHub.

.github > workflows > back-tests.yml

```yaml
name: jest-and-docker-ci

on: push

jobs:
  test-back:
    runs-on: ubuntu-latest
#    container:
#      image: node:lts
    steps:
      - name: Check out code
        uses: actions/checkout@v2
      - name: run tests
        run: npm i && npm run test-ci
  docker:
    needs: test-back
    if: github.ref == 'refs/heads/dev'
    runs-on: ubuntu-latest
    steps:
      - name: Set up QEMU
        uses: docker/setup-qemu-action@v2
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2
      - name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build and push
        uses: docker/build-push-action@v4
        with:
          push: true
          context: '{{defaultContext}}'
          tags: ${{ secrets.DOCKERHUB_USERNAME }}/wns-jaune-oros-back:latest
```

Workflow Front-End

Le workflow front-end suit une structure similaire :

1. **Tests Jest** : Les tests sont exécutés pour le front-end, bien que les tests spécifiques soient encore en cours de mise en place. Pour l'instant, le processus installe les dépendances avec npm i sans lancer de tests front, mais cela sera amélioré à mesure que les tests frontaux seront finalisés.
2. **Build et Push Docker** : Après l'installation des dépendances, l'image Docker du front-end est construite et poussée sur DockerHub, utilisant également les secrets pour la sécurité des identifiants.

.github > workflows > front-tests.yml

```yaml
name: jest-and-docker-ci

on: push

jobs:
  test-front:
    runs-on: ubuntu-latest
    #    container:
    #      image: node:lts
    steps:
      - name: Check out code
        uses: actions/checkout@v2
      - name: run tests
        # run: npm i && npm run test-ci // supprimer npmi quand on a les tests
        run: npm i
  docker:
    # needs: test-front
    if: github.ref == 'refs/heads/dev'
    runs-on: ubuntu-latest
    steps:
      - name: Set up QEMU
        uses: docker/setup-qemu-action@v2
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2
      - name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build and push
        uses: docker/build-push-action@v4
        with:
          push: true
          context: "{{defaultContext}}"
          tags: ${{ secrets.DOCKERHUB_USERNAME }}/wns-jaune-oros-front:latest
```

Ensuite, nous avons configuré l'environnement serveur nécessaire pour le déploiement continu, (ici on suppose que l'installation préalable du VPS, de Docker et de Docker Compose ont déjà été effectuées.)

- **Organisation des dossiers**
  Nous avons créé deux dossiers **staging** et **production** dans le répertoire principal de notre projet **Oros** sur le VPS. Chaque dossier contient plusieurs fichiers de configuration essentiels :
    - **Docker-compose-prod.yml** : Ce fichier indique à Docker comment construire et démarrer les services du **backend**, du **frontend**, et de la **base de données**. La version des services y est spécifiée pour assurer des déploiements stables et reproductibles. Ce fichier est conçu pour la production (**prod**), à la différence du fichier **docker-compose.staging.yml**, utilisé uniquement en développement.
    - **Nginx.conf** : Ce fichier gère la redirection des requêtes **/graphql** vers notre application **Node.js** et sert également les fichiers statiques en envoyant les informations à **Caddy**.
    - **fetch-and-deploy-${{ENV}}.sh** : Ce script bash, appelé par la configuration de notre **Webhook**, relance les services via Docker Compose avec les dernières versions des images Docker. Il spécifie également les ports utilisés par Nginx (8000 pour la production et 8001 pour le staging), ce qui permet à Caddy de rediriger les requêtes correctement.
    - **.env** : Ce fichier contient les variables d'environnement nécessaires au projet, comme les secrets utilisés par les services. Il est lu par **docker-compose-prod.yml** pour transmettre ces informations aux conteneurs Docker.

- **Configuration de Caddy et des webhooks**
  Nous avons également configuré les fichiers de configuration pour **Caddy** et les **webhooks** :

    - **Caddyfile** : Ce fichier permet de rediriger les requêtes entrantes via la directive reverse_proxy vers les ports adéquats. Les requêtes vers staging.1123-jaune-1.wns.wilders.dev sont redirigées vers le port 8001 (staging), celles vers 1123-jaune-1.wns.wilders.dev vers le port 8000 (production), et les requêtes vers ops.1123-jaune-1.wns.wilders.dev sont envoyées vers le port 9000 (service Webhook).

- **Webhook.conf** : Ce fichier de configuration en JSON décrit les actions à exécuter lorsqu'une notification est reçue. Nous avons mis en place deux webhooks :
  - Le premier est déclenché par **DockerHub** lorsqu'une nouvelle image de staging est disponible.
  - Le second est déclenché par un push sur la branche **main** de GitHub. Les deux webhooks exécutent un script bash pour mettre à jour soit l'environnement de staging, soit celui de production.

- **Fichiers Docker Compose et Nginx**
  Voici un aperçu des fichiers utilisés pour la configuration de nos services et de notre proxy :
  - **services : backend, frontend, db, nginx** : Le fichier **docker-compose-prod.yml** configure les services backend (Node.js), frontend (Next), base de données (Postgres), et Nginx pour gérer le reverse proxy. Chacun de ces services est supervisé par Docker, avec des checks de santé (`healthcheck`) et des dépendances bien définies entre eux.

```
services:
  backend:
    image: orosgroup/wns-jaune-oros-back
    expose:
      - 4000
    env_file:
      - ./db.env
    healthcheck:
      test:
        - CMD-SHELL
        - "curl -f http://backend:4000/graphql?query=%7B__typename%7D -H 'Apollo-Require-Preflight: true' || exit 1"
      interval: 10s
      timeout: 30s
      retries: 5
    depends_on:
      db:
        condition: service_healthy
    command: npm run start

  frontend:
    image: orosgroup/wns-jaune-oros-front
    expose:
      - 3000
    depends_on:
      backend:
        condition: service_healthy
    env_file:
      - ./.env
```

```yaml
  db:
    image: postgres
    restart: always
    env_file:
      - ./db.env
    expose:
      - 5432
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -d oros -U utilisateur"]
      interval: 10s
      timeout: 5s
      retries: 5
    volumes:
      - orosdatabase:/var/lib/postgresql/data


  nginx:
    image: nginx:1.21.3
    depends_on:
      - backend
      - frontend
    restart: always
    ports:
      - ${GATEWAY_PORT:-8000}:80
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
      - ./logs:/var/log/nginx


  adminer:
    image: adminer
    restart: always
    ports:
      - 8087:8080

volumes:
  orosdatabase:
```

- **nginx.conf** : Ce fichier redirige les requêtes entrantes. Par exemple, les requêtes vers `/graphql` sont envoyées au backend, et toutes les autres requêtes sont redirigées vers le frontend.

```
events {}

http {
  include mime.types;

  server {
    listen 80;

    location /graphql {
      proxy_pass http://backend:4000;
    }

    location / {
      proxy_pass http://frontend:3000;
    }
  }
}
```

- **fetch-and-deploy.sh** : Ce script bash automatise le redéploiement en arrêtant les services existants, en téléchargeant les dernières versions des images Docker, puis en redémarrant les services sur le bon port.

```
#!/bin/sh
# fetch-and-deploy.sh
docker compose -f docker-compose-prod.yml down && \
    docker compose -f docker-compose-prod.yml pull && \
    GATEWAY_PORT=8000 docker compose -f docker-compose-prod.yml up -d;
```

- **Caddyfile et webhook.conf** : Ces fichiers gèrent respectivement les redirections vers les ports appropriés et l'automatisation du redéploiement via les webhooks.

Caddyfile

```
1123-jaune-1.wns.wilders.dev {
        # Redirect request to the production running on port 8000
        reverse_proxy localhost:8000

        # log
        log {
            output file /var/log/caddy/production.log
        }
}

staging.1123-jaune-1.wns.wilders.dev {
        # Redirect request to the staging running on port 8001
        reverse_proxy localhost:8001

        # log
        log {
            output file /var/log/caddy/staging.log
        }

}

ops.1123-jaune-1.wns.wilders.dev {
        # Redirect request to the webhook service running on port 9000
        reverse_proxy localhost:9000

        # log
        log {
            output file /var/log/caddy/ops.log
        }
}
```

webhook.conf

```
[
  {
    "id": "update-staging",
    "execute-command": "/home/wns_student/apps/oros/staging/fetch-and-deploy.sh",
    "command-working-directory": "/home/wns_student/apps/oros/staging/"
  },
  {
    "id": "update-prod",
    "execute-command": "/home/wns_student/apps/oros/prod/fetch-and-deploy.sh",
    "command-working-directory": "/home/wns_student/apps/oros/prod/"
  }
]
```

7. Sécurité et veille

7.1- Sécurité

La **sécurité** est un aspect primordial pour une application web. Nous avons mis en place plusieurs mesures :

- Un middleware permettant de hacher les mots de passe à l'aide **d'Argon2**, renforçant la protection contre les attaques par force brute et compromissions de bases de données
- Utilisation des **JSON Web Tokens (JWT)** pour l'authentification, assurant la gestion des sessions et la déconnexion automatique en cas de token expiré.
- Les **JWT sont stockés dans des cookies sécurisés** (avec l'option **HttpOnly** activée), empêchant tout accès direct depuis le JavaScript côté client, réduisant ainsi les risques d'attaques **XSS**.
- nous avons défini des options **d'origine** des requêtes, les "**CORS**" (Cross Origin Resource Sharing), qui permettent de définir quelles origines sont autorisées à faire des requêtes sur notre serveur (l'url de notre front-end par exemple) (cf. p33)
- Un **authChecker personnalisé** avec le décorateur **@Authorized**, permettant de protéger l'accès aux résolveurs en fonction des rôles des utilisateurs (ex. : **@Authorized("ADMIN")** pour restreindre certaines actions aux administrateurs).
- **Protection des données transmises** via le protocole **HTTPS** pour garantir le chiffrement des échanges entre le client et le serveur, protégeant ainsi l'intégrité et la confidentialité des informations.
- Utilisation de **variables d'environnement** pour stocker les informations sensibles, telles que les secrets JWT et les identifiants de base de données, empêchant leur divulgation dans le code source.
- Configuration d'un **reverse proxy** avec **Caddy**, assurant la redirection des requêtes tout en gérant les certificats HTTPS.

Afin de compléter la réflexion sur les **vulnérabilités de sécurité** et les mesures possibles, j'ai effectué une recherche sur les attaques **CSRF** (Cross-Site Request Forgery) et plus particulièrement sur la mise en place de **jetons CSRF** pour protéger

l'application. J'ai effectué une recherche avec les mots clés "**csrf token**" et ai obtenu plusieurs résultats de différents sites (Stackoverflow, Clever-age, Bright Security...). J'ai constaté que notre application pourrait être la cible d'attaques CSRF si un utilisateur se connecte et visite un **site malveillant** lorsque sa session est active. L'attaquant pourrait alors par exemple supprimer ou modifier un matériel via une requête qui serait acceptée car il utiliserait le **token** de notre utilisateur.

Pour s'en prémunir, nous pourrions ajouter un **paramètre unique et imprévisible** appelé "jeton CSRF" à chaque action de l'utilisateur. Ce jeton est généré par le serveur et peut être **stocké en session** avec d'autres informations d'identification. Il est inséré dans les formulaires ou ajouté aux liens vers des actions privées. Lorsqu'un formulaire est soumis, le jeton est renvoyé au serveur pour vérification. Cela empêche les "attaques forgées" et garantit que les actions ne sont exécutées que par les **utilisateurs légitimes**. Le jeton change pour chaque utilisateur.

Il est également important de noter que, dans le cadre d'une application GraphQL, les attaques CSRF restent pertinentes. Étant donné que les requêtes sont souvent envoyées via des appels **POST**, les mécanismes de protection comme les jetons CSRF doivent être appliqués. Si l'application utilise des **cookies** pour l'authentification, elle est vulnérable aux **CSRF**, d'où l'importance de mettre en œuvre ces jetons.

En complément, j'ai découvert que les **cookies** et le **localStorage** présentent chacun des vulnérabilités différentes. Par exemple, le **localStorage**, bien qu'insensible aux attaques **CSRF**, est vulnérable aux attaques **XSS** (Cross-Site Scripting) car il est accessible via JavaScript. Il est donc important de bien gérer la suppression des données puisque le **localStorage** n'a pas de date d'expiration automatique. À l'inverse, les **cookies** peuvent être protégés des attaques **XSS** en activant les attributs **HttpOnly** et **Secure**, garantissant ainsi qu'ils ne sont pas accessibles via JavaScript et qu'ils ne sont transmis qu'en HTTPS. Cependant, ils restent vulnérables aux attaques **CSRF**, sauf si l'attribut **SameSite** est défini sur **Strict** ou **Lax**, limitant ainsi leur utilisation à un contexte propriétaire.

7.2- Recherche à partir d'une source anglophone

Lors de la mise en place des **JWT**, j'avais besoin de **comprendre** leur fonctionnement. J'ai donc réalisé une recherche avec les mots-clés "**JsonWebToken structure**". Parmis

les résultats, des sites tels que **auth0**, **jwt.io**, **ionos**, **supertokens.com...** C'est ce dernier que j'ai choisi de détailler, pour sa **clarté** et sa **concision** :

"JSON Web Token is an open industry standard used to share information between two entities, usually a client (like your app's frontend) and a server (your app's backend).

They contain JSON objects which have the information that needs to be shared. Each JWT is also signed using cryptography (hashing) to ensure that the JSON contents (also known as JWT claims) cannot be altered by the client or a malicious party.

For example, when you sign in with Google, Google issues a JWT which contains the following claims / JSON payload:

A JWT contains three parts:

- ***Header**: Consists of two parts:*
  - *The signing algorithm that's being used.*
  - *The type of token, which, in this case, is mostly "JWT".*
- ***Payload**: The payload contains the claims or the JSON object.*
- ***Signature**: A string that is generated via a cryptographic algorithm that can be used to verify the integrity of the JSON payload."*

Traduction :

JSON Web Token (jeton web JSON) est une norme libre utilisée pour partager l'information entre deux entités, habituellement un client (comme la partie client de votre application) et un serveur (celui de votre application).

Ils sont constitués d'objets JSON qui contiennent l'information qui nécessite d'être partagée. Chaque JWT est également signé en utilisant une cryptographie (hachage) pour s'assurer que le contenu JSON (également nommé attributs JWT) ne puisse pas être altéré par le client ou un parti malveillant.

Par exemple, lorsque vous vous connectez sur Google, Google produit un JWT qui contient le payload (littéralement "charge utile") suivant :

- **En-tête**: consiste en deux parties :

    - la signature algorithmique utilisée
    - le type de jeton, qui est, dans ce cas, la plupart du temps “JWT”
- **Payload**: le payload contient les attributs ou l’objet JSON
- **Signature**: une chaîne de caractères qui est générée via un algorithme cryptographique qui peut être utilisé pour vérifier l'intégrité du payload JSON.

Cette recherche m’a permis de comprendre les JWT, que j’ai mis en place dans notre application.