DOSSIER PROJET

Léopold Lesaulnier

Réalisation projet de formation en groupe

Application EasyGifts

RNCP Concepteur Développeur d'Applications de niveau 6

Wild Code School - Bordeaux

Sommaire

Introduction

Mon intérêt pour le développement, et plus particulièrement pour le web, est né vers la fin de mes études en sculpture. À ce moment-là, je me suis interrogé sur l'intégration de nouveaux médiums et médias dans mon travail artistique. C'est ainsi que l'idée de m'intéresser aux interfaces web a émergé, stimulée par ma curiosité pour le traitement des données. Lorsque la pandémie de COVID est survenue, elle m'a offert du temps libre alors que tout s'arrêtait. J'en ai profité pour plonger dans le frontend, attiré par son aspect créatif, avant d'explorer plus en profondeur le JavaScript. Au fil de ma reconversion, j'ai découvert un plaisir croissant à manipuler les données, ce qui m'a naturellement conduit à m'intéresser aux aspects côté serveur.

Aujourd'hui, dans la continuité de ce parcours, et dans le cadre de mon alternance, je vous présente la conception et le développement d'un projet réalisé en groupe durant notre formation.

Il s'agit d'une application nommée EasyGifts, conçue pour répondre au besoin d'échanger des idées de cadeaux au sein d'un groupe familial ou amical. Le principe est simple : l'utilisateur crée un groupe de discussion dans lequel il invite les participants de son choix. Une fois intégré au groupe, chaque utilisateur a accès à toutes les discussions concernant les membres, permettant ainsi de partager et de discuter des idées de cadeaux pour chacun. Cependant, une discussion concernant un utilisateur spécifique reste active sans que celui-ci puisse y accéder, préservant ainsi l'effet de surprise.

Les technologies utilisées

Je vais vous décrire l'ensemble des technologies que nous avons utilisées pour la conception de l’application.

Backend :

Tout d'abord, nous avons choisi une base de données relationnelle PostgreSQL, couplée à TypeORM, qui permet d'interfacer les données avec notre serveur API GraphQL. TypeORM offre non seulement une interface efficace pour la gestion des données, mais aussi des fonctionnalités de sécurité robustes contre les injections SQL, garantissant une base de données protégée. De plus, TypeORM est régulièrement mis à jour, ce qui nous permet de bénéficier des dernières fonctionnalités et améliorations de sécurité.

Étant donné que nous avons opté pour développer l'ensemble de notre application en TypeScript, il était logique d'utiliser TypeGraphQL, qui tire directement parti des schémas de données extraits des types TypeScript. GraphQL, avec son exact fetching, permet de récupérer précisément les données nécessaires, évitant ainsi les requêtes excessives et optimisant les performances.

Enfin, pour lier le tout, nous avons décidé d'utiliser Apollo Server, qui facilite la gestion des requêtes GraphQL, offre des outils puissants pour le traitement des données, et s'intègre parfaitement avec notre stack technologique. Cela assure non seulement des performances optimales, mais également une modularité accrue dans notre application.

Frontend :

Nous avons opté pour l'utilisation de Next.js pour notre application, un choix qui s'est avéré judicieux en raison de son routage performant et efficace. Next.js est nativement orienté vers le server-side rendering (SSR), ce qui facilite la gestion des requêtes avec des surcouches middleware. Cependant, dans notre cas spécifique, nous avons majoritairement utilisé le client-side rendering, tout en gardant la possibilité d'intégrer des middlewares là où c'était nécessaire.

Pour renforcer l'intégration de notre backend GraphQL, nous avons choisi d'utiliser Apollo Server avec Apollo Client, complétés par Codegen. Cette combinaison nous permet de générer des hooks spécifiques à chaque requête GraphQL, rendant le développement plus fluide et optimisant la communication entre le frontend et le backend.

En ce qui concerne l'interface utilisateur, nous avons choisi d'utiliser la librairie de composants ShadCN. Cette librairie nous offre une collection de composants bien conçus et personnalisables, nous permettant de créer une interface utilisateur cohérente, moderne et responsive, tout en accélérant le développement de l'UI.

Compétences du référentiel mises en oeuvre par le projet

1 - Développer une application sécurisée

1.1 Installer et configurer son environnement de travail en fonction du projet.
En début de projet, nous nous sommes chargés en groupe de réaliser la configuration et l'installation de notre environnement de travail.

Avec la création d'un repo GitHub commun avec deux branches distinctes "main" et "dev".

La configuration de notre environnement de travail est partagée par la dockerisation de l'ensemble de notre projet, aussi bien de la base de données, du backend et du frontend.

1.2 Développer des interfaces utilisateur

Nous avons réalisé une interface de connexion et d'inscription au moment de la mise en place des bases du projet en groupe.

Ainsi que lors du développement de la fonctionnalité de création de groupe, au moment de la partie concernant le frontend j'ai réalisé différentes interfaces utilisateurs afin d'interagir avec l'api du backend pour créer, modifier ou obtenir des données.

1.3 Développer des composants métier

Toujours lors de la mise en place des bases du projet nous avons créé un middleware sur notre serveur Apollo afin de vérifier la présence d'un jeton d'authentification JWT dans les cookies accompagnant chaque requête entrante.

Puis lors du développement j'ai veillé à vérifier l'authentification pour exécuter certaines opérations dans dans le backend.

1.4 Contribuer à la gestion d'un projet informatique

Nous avons organisé la conception et le développement de ce projet à l'aide d'un logiciel de suivi de projet par tâche, avec la tenue d'un backlog sur Trello et une répartition des tâches sur des périodes de sprints d'une semaine.

2 - Concevoir et développer une application sécurisée organisée en couches

2.1 Analyser les besoins et maquetter une application

Afin de répondre au mieux au besoin du projet, nous avons réalisé des maquettes sur figma, tout d'abord un wireframe puis une maquette définissant la charte graphique et les inspirations.

2.2 Définir l'architecture logicielle d'une application

Dans le but de concevoir une application totale et efficace, nous avons étudié le cahier des charges du projet et avons défini des diagrammes de séquences pour les fonctionnalités principales.

2.3 Concevoir et mettre en place une base de données relationnelle

Toujours dans une démarche d'unicité nous avons initialisé notre base de données au moment de la création de la base commune de notre projet.
Établir les schémas de données.
Avec la connexion à la base en elle même et à la configuration de notre fonction de réinitialisation de données de test.

2.4 Développer des composants d'accès aux données SQL

Au moment de la mise en place de l'authentification nous avons créé un premier resolver concernant l'utilisateur.
Puis lors du développement de la fonctionnalité de création de groupe, je suis venu créer plusieurs resolvers pour gérer la création des groupes.

3 - Préparer le déploiement d'une application sécurisée

3.1 Préparer et exécuter les plans de test d'une application

Nous avons mis en place dès le début un environnement de test complet sur notre projet, pour plus simplement par la suite venir ajouter des éléments à tester.

3.2 Préparer et documenter le déploiement d'une application

Nous avons implémenté des règles sur notre repository GitHub afin de lancer les tests à chaque pull request sur notre branch "dev".

3.3 Contribuer à la mise en production dans une démarche DevOps

Nous avons défini des scripts de déploiement de nos environnements de développement conteneurisés, ainsi que configuré un ensemble de GitHub Actions pour lancer les tests afin d'approuver notre code.

Nous avons décidé de mettre en place les bases du projet ensemble afin d'établir des méthodes et bonnes pratiques communes afin de pouvoir développer ensemble de manière plus sereine par la suite.

Cahier des charges

Afin de concrétiser au mieux l'apprentissage de cette année d'alternance, nous avons réalisé, avec un groupe de cinq personnes, une application de discussion de groupe nommée EasyGifts. Cette application permet aux membres d'une famille ou d'un groupe d'amis de discuter des idées de cadeaux sans que chacun sache ce que les autres préparent pour lui. Chaque membre dispose d'un fil de discussion spécifique, permettant aux autres membres de partager des idées de cadeaux individuels ou collectifs.

Objectif du projet : l'objectif principal de ce projet est de permettre aux membres d'une famille ou d'un groupe d'amis de discuter d'idées de cadeaux sans que chacun puisse voir ce qui est prévu pour lui-même.

Principe Général

- **Création de Groupe** : Un utilisateur enregistré peut créer un groupe. Les utilisateurs non inscrits sur l'application reçoivent un mail d'invitation pour qu'ils rejoignent le groupe.
- **Accès aux fils de discussion** : Chaque membre peut accéder à tous les fils de discussion sauf à celui qui le concerne directement.

Fonctionnalités du MVP

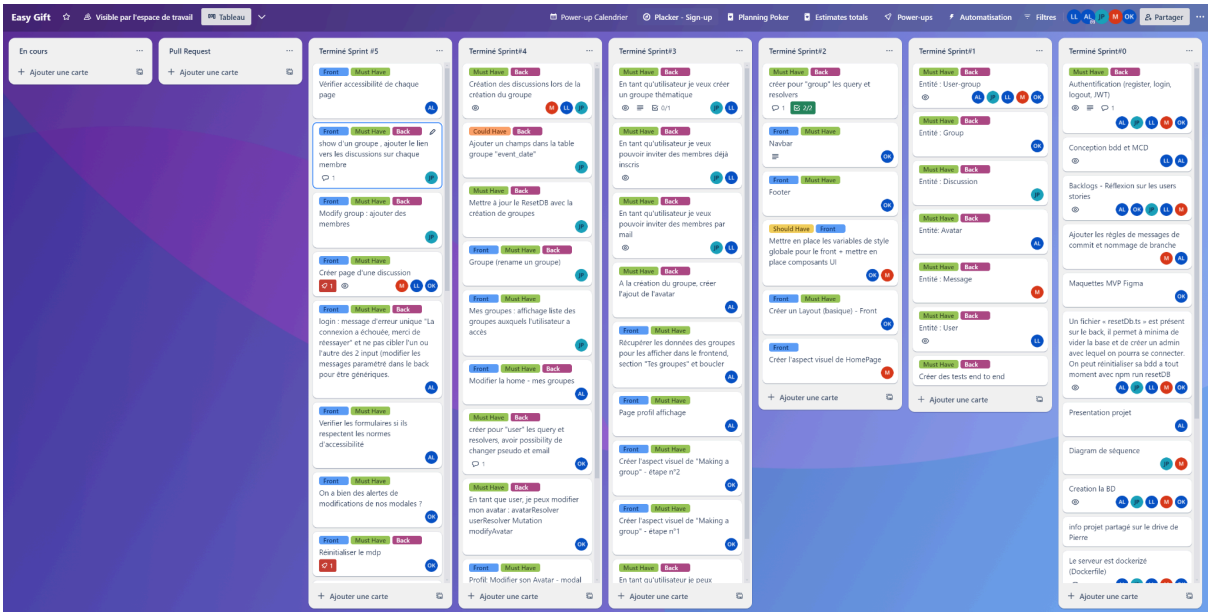
- **Création de groupe** : Les utilisateurs enregistrés peuvent créer un groupe.
- **Inscription et adhésion** : Les utilisateurs peuvent s'inscrire et rejoindre des groupes existants.
- **Accès aux fils de discussion** : Chaque membre a accès aux groupes dont il fait partie et peut rejoindre les discussions de ces groupes.
- **Publication et affichage des messages** : Les messages s'affichent du plus récent au plus ancien avec le nom de l'auteur et la date. Les nouveaux messages apparaissent en temps réel.

Évolutions Futures

- **Affichage des URL** : Les URLs dans les messages s'affichent avec un aperçu.
- **Cagnotte** : Ajout de cagnottes dans les fils de discussion.
- **Personnalisation des profils** : Chaque membre peut choisir un avatar.
- **Fonctionnalités Avancées de Discussion** : Possibilité d'épingler les messages pertinents.
- **Gestion de Groupe** : Le propriétaire du groupe peut exclure des membres, et chaque membre peut quitter un groupe.

Gestion projet

Nous étions une équipe de cinq développeurs, Morgane, Olga, Aurélie, Jérémie et moi-même, et nous avons travaillé en étroite collaboration pour mener à bien ce projet de développement d'application. Dès le début, nous avons opté pour une méthodologie Agile, en répartissant les différentes tâches liées à la conception de l'application grâce à l'outil de gestion de projet Trello. Cet outil nous a permis de suivre l'avancement du projet de manière visuelle et organisée, en créant un backlog constitué de multiples user stories. Ces user stories ont été minutieusement classées par ordre de priorité et d'importance par rapport à l'objectif principal du projet, le Minimum Viable Product (MVP).



Nous avons ensuite structuré le projet en sprints hebdomadaires, chaque sprint étant conçu pour se concentrer sur un ensemble spécifique de fonctionnalités à implémenter. Chaque début de semaine, un daily meeting était organisé pour planifier les tâches à accomplir. Ces réunions courtes mais précieuses nous ont permis de synchroniser nos efforts, de faire un point sur l'avancement des travaux, et de définir clairement les objectifs pour la semaine à venir. Pendant ces meetings, nous prenions également le temps de discuter des éventuelles difficultés rencontrées, ce qui nous permettait de réajuster notre plan de travail si nécessaire.

Pour le versionnage du code, nous avons choisi d'utiliser GitHub, un choix stratégique qui nous a permis de collaborer efficacement tout en maintenant une grande rigueur dans la gestion des versions. Nous avons mis en place deux branches principales : Dev et Main, chacune ayant un rôle bien défini dans notre workflow. Afin d'assurer la stabilité et la qualité du code, des règles de sécurité strictes ont été établies. Par exemple, le push direct sur la branche Main était strictement interdit, et le push sur la branche Dev nécessitait une validation préalable via une pull request. Chaque modification apportée au code devait donc être soumise sous forme de pull request, qui devait être examinée et approuvée par au moins un autre membre de l'équipe. Cette pratique a permis de garantir une revue de code systématique et d'éviter les régressions.

Par ailleurs, nous avons intégré des tests automatisés via GitHub Actions. Chaque pull request devait passer avec succès l'ensemble des tests configurés avant de pouvoir être fusionné avec la branche Dev. Cette approche a été cruciale pour assurer la robustesse et la fiabilité de l'application tout au long du développement.

Le processus de développement se déroulait de la manière suivante : chaque développeur choisissait une user story correspondant au sprint en cours, puis récupérait la dernière version de la branche Dev avant de créer une nouvelle branche locale. Cette branche était nommée en fonction de la date et de la tâche à réaliser, suivant une nomenclature prédéfinie en anglais. Une fois la tâche accomplie, le développeur soumettait une pull request vers la branche Dev et invitait ses collègues à procéder à une revue de code. Cette étape collaborative permettait de partager les avancées réalisées, d'obtenir des retours constructifs et d'améliorer le code si nécessaire. Après l'approbation de l'ensemble de l'équipe, la branche locale était fusionnée avec la branche Dev, garantissant ainsi une intégration continue et sans heurts des nouvelles fonctionnalités.

En conclusion, ce projet a été un excellent exemple de collaboration efficace et structurée au sein d'une équipe de développement. Grâce à une organisation rigoureuse et à l'utilisation d'outils adaptés comme Trello pour la gestion des tâches et GitHub pour le versionnage du code, nous avons pu avancer de manière cohérente et maîtrisée. Chaque membre de l'équipe a su contribuer à l'atteinte des objectifs fixés, en respectant les bonnes pratiques de développement et en veillant à la qualité du code via des revues régulières et des tests automatisés. Cette méthodologie nous a permis de livrer un MVP solide et fonctionnel, tout en renforçant la cohésion et les compétences de l'équipe. Ce projet illustre parfaitement l'importance de la communication, de la répartition claire des responsabilités et de l'engagement de chacun pour mener à bien un projet de développement logiciel. Cette expérience a non seulement permis d'aboutir à un produit de qualité, mais a également enrichi chacun de nous en termes de compétences techniques et de collaboration en équipe.

Maquette besoin projet

Concernant le design et l'expérience utilisateur (UX), nous avons pris soin de créer une interface à la fois intuitive et esthétique, en veillant à ce que l'accès des utilisateurs non authentifiés soit limité à la seule page d'accueil. Cette page d'accueil a été conçue pour offrir une vue d'ensemble des fonctionnalités proposées par l'application, tout en incitant les utilisateurs à s'inscrire ou à se connecter pour accéder aux fonctionnalités complètes. L'ensemble de l'application est structuré autour d'un layout cohérent, incluant une barre de navigation qui contient un titre, des liens de navigation dont la visibilité est conditionnée par l'état d'authentification de l'utilisateur, ainsi qu'un composant de connexion pour faciliter l'accès rapide aux fonctionnalités clés.

Pour assurer une identité visuelle forte et uniforme, nous avons sélectionné trois couleurs dominantes qui sont appliquées de manière cohérente sur l'ensemble de l'interface. Ces couleurs, choisies avec soin, créent une esthétique visuelle harmonieuse qui renforce l'image de l'application. Nous avons également intégré la librairie de composants ShadCN,

qui non seulement s'accorde parfaitement avec notre palette de couleurs, mais offre également des composants prêts à l'emploi, modernes et réactifs, améliorant ainsi la fluidité de l'expérience utilisateur.

Le wireframe que nous avons réalisé joue un rôle central dans la définition de la logique de base des différents écrans de l'application. Il s'agit d'un document visuel essentiel, qui nous a permis de structurer les différents éléments de l'interface en alignement avec les besoins identifiés lors de la phase de conception. Ce wireframe a été conçu en tenant compte des exigences du projet, en commençant par une page d'accueil stratégique qui limite l'accès aux utilisateurs non authentifiés, les encourageant ainsi à créer un compte pour bénéficier pleinement des fonctionnalités de l'application.

Le wireframe se compose d'un template unifié, reproduit sur l'ensemble des écrans de l'application, garantissant une navigation cohérente et une expérience utilisateur homogène. Les principaux écrans incluent :

Page d'accueil :

- Présentation des fonctionnalités principales de l'application.
- Accès restreint aux utilisateurs non authentifiés.
- Invitation à créer un compte ou à se connecter.

Écran de connexion et d'inscription :

- Formulaire de connexion pour les utilisateurs existants.
- Formulaire d'inscription pour les nouveaux utilisateurs.
- Page liée à la récupération du mot de passe.

Interface de création de groupe :

- Formulaire pour créer un nouveau groupe avec les détails nécessaires (nom, description, etc.).
- Option pour inviter des utilisateurs à rejoindre le groupe.
- Aperçu du groupe avant la validation de la création.

Page listant les groupes existants :

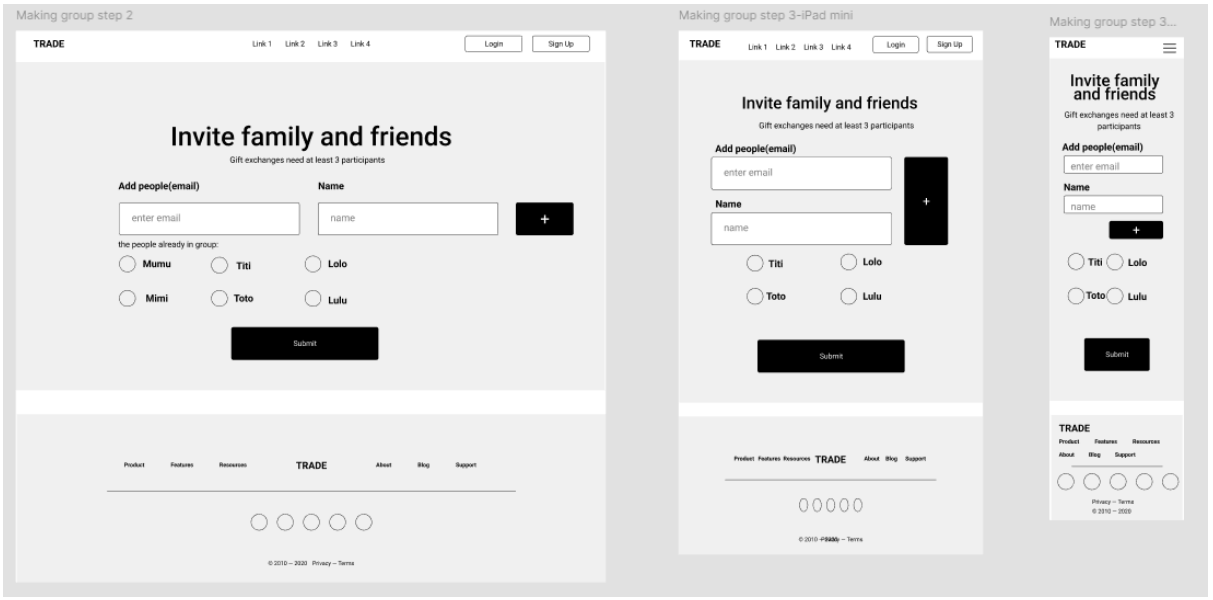
- Affichage des groupes disponibles avec des options de tri et de filtre.
- Accès rapide aux groupes auxquels l'utilisateur appartient.
- Option pour rejoindre un groupe public ou privé.

Écran de détails d'un groupe :

- Présentation des informations détaillées du groupe (nom, description, membres, etc.).
- Liste des membres avec leurs rôles respectifs.
- Options pour modifier les paramètres du groupe (pour les administrateurs).

Page de discussion :

- Interface de messagerie inspirée des applications de chat modernes.
- Historique des conversations avec possibilité de répondre aux messages.
- Intégration des fonctionnalités de partage de fichiers et de médias.



L'adaptation de l'interface à différents types d'écrans a également été une priorité. Nous avons veillé à ce que l'application soit responsive, offrant ainsi une expérience utilisateur optimale, qu'il s'agisse d'une utilisation sur mobile, tablette ou ordinateur.

En matière de design, des décisions clés ont été prises pour renforcer l'esthétique visuelle de l'application. En plus des trois couleurs contrastées qui forment la base de notre palette, nous avons choisi un nuancier subtil pour créer des variations harmonieuses de ces couleurs, enrichissant ainsi l'identité visuelle globale. Une attention particulière a également été portée à la sélection d'une police de caractères, soigneusement choisie pour s'accorder avec l'ensemble de l'esthétique de l'application. L'intégration des composants ShadCN, qui s'accordent avec notre palette de couleurs et notre typographie, a permis d'apporter une touche de modernité tout en garantissant une expérience utilisateur fluide et agréable.

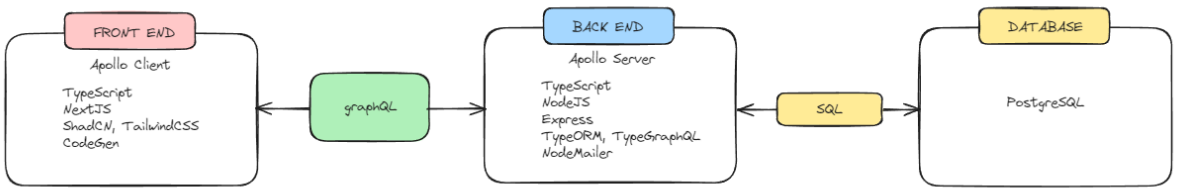
En conclusion, ce projet de design et d'UX a permis de créer une interface utilisateur à la fois fonctionnelle et esthétique, tout en assurant une expérience utilisateur cohérente sur l'ensemble des écrans. Grâce à une approche structurée et à des choix de design réfléchis, nous avons réussi à offrir une application visuellement engageante, facile à naviguer et accessible sur tous les appareils. Cette approche centrée sur l'utilisateur, combinée à une attention minutieuse aux détails visuels, a permis de garantir que l'application non seulement répond aux besoins fonctionnels des utilisateurs, mais leur offre également une expérience agréable et intuitive, contribuant ainsi à la réussite globale du projet.

Architecture de l’application

L'architecture de l'application est divisée en trois parties principales : le frontend, le backend, et la base de données.

- **Frontend :** Le frontend constitue l'interface utilisateur avec laquelle les utilisateurs interagissent directement. Il est responsable de la présentation et de la gestion des interactions, et communique avec le backend via GraphQL. Grâce à GraphQL, le frontend peut effectuer des requêtes et des mutations pour récupérer ou envoyer des données de manière efficace et structurée, réduisant ainsi la surcharge réseau et optimisant les performances de l'application.
- **Backend :** Le backend agit comme une couche intermédiaire entre le frontend et la base de données. Il reçoit les requêtes GraphQL provenant du frontend, exécute la logique métier nécessaire, et interagit avec la base de données pour récupérer, manipuler ou mettre à jour les données demandées. Le backend est essentiel pour orchestrer les opérations, gérer les autorisations, et assurer la cohérence des données à travers les différentes fonctionnalités de l'application.
- **Base de données :** La base de données est l'élément central où sont stockées toutes les informations nécessaires au bon fonctionnement de l'application, telles que les utilisateurs, les groupes, et autres entités. Les échanges de données entre le backend et la base de données se font en SQL, permettant de réaliser des opérations de lecture, écriture, mise à jour, et suppression des données de manière fiable et sécurisée.

Cette architecture modulaire assure une séparation claire des responsabilités, avec le frontend chargé de l'affichage et de l'interaction utilisateur, le backend orchestrant la logique applicative et la communication, et la base de données stockant les informations de manière sécurisée et organisée. Cette approche garantit une meilleure maintenabilité, évolutivité et sécurité de l'application.

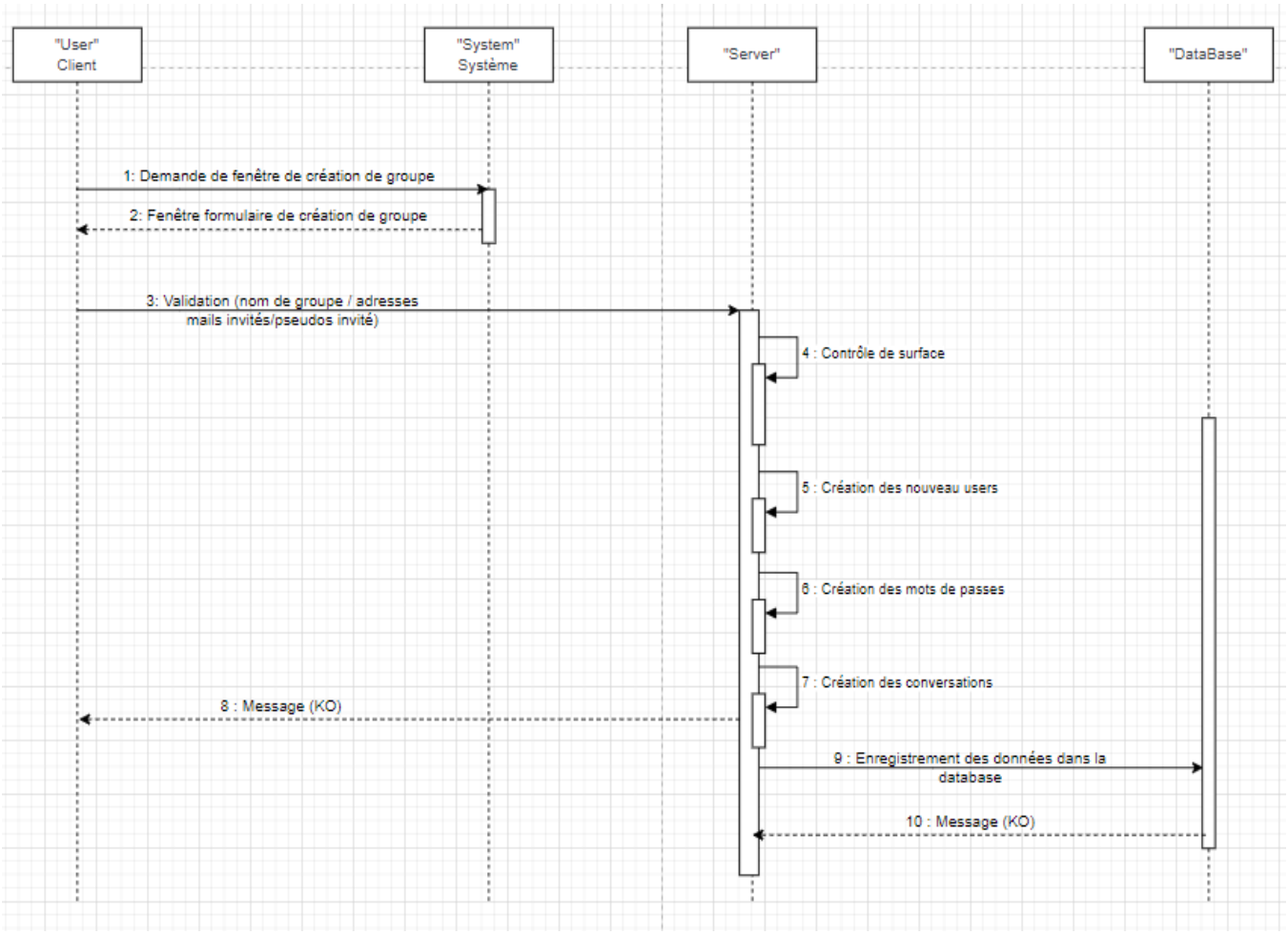


En outre, pour traiter la logique complexe de certaines fonctionnalités principales, nous avons travaillé sur la schématisation des processus. Avec l'aide de Jérémie, j'ai élaboré un diagramme représentant les différentes couches d'exécution pour la fonctionnalité de création d'un groupe. Ce diagramme retrace l'action d'un utilisateur sur son client, en illustrant l'exécution et le parcours de sa requête à travers les différentes couches de notre application. Pour chaque étape, nous avons conçu et intégré la logique de traitement et la réaction spécifique à chacune de ces couches, assurant ainsi une fluidité et une cohérence dans le traitement des requêtes.

Cette architecture en trois couches, combinée à une approche rigoureuse de la modélisation des processus, nous a permis de développer une application robuste, bien structurée, et capable de répondre aux besoins fonctionnels tout en offrant une expérience utilisateur fluide. L'organisation claire des responsabilités entre le frontend, le backend et la base de données, ainsi que la schématisation précise des fonctionnalités, ont été des éléments clés pour la réussite de ce projet, assurant à la fois efficacité et maintenabilité dans le développement et l'évolution de l'application.

Notamment, la fonctionnalité de création de groupe, dont je vais présenter le diagramme.

Diagramme de séquences de la création de groupe



Conception de la base de données

Pour concevoir une architecture de données optimale pour notre application, nous avons d'abord analysé les besoins spécifiques du projet. Celui-ci repose principalement sur la nécessité de relier diverses données entre elles. Par exemple, associer un utilisateur à un groupe, un groupe à une discussion, ou encore lier des messages à un utilisateur, et ce dernier à des discussions. Chaque groupe héberge des discussions concernant ses membres.

Afin de construire une base de données robuste et cohérente, c'est-à-dire en évitant la duplication des données et en facilitant leur réutilisation grâce aux relations, nous avons commencé par identifier les principaux acteurs de notre application, ainsi que les futures tables de notre base de données.

Ainsi, un utilisateur doit sélectionner un avatar pour se représenter.

Il peut appartenir à plusieurs groupes et peut également être à l'initiative de la création de ces groupes.

Chaque groupe peut accéder à plusieurs discussions.

Pour mettre en œuvre cette structure, nous avons élaboré différents modèles de schémas de données que je vais maintenant vous présenter.

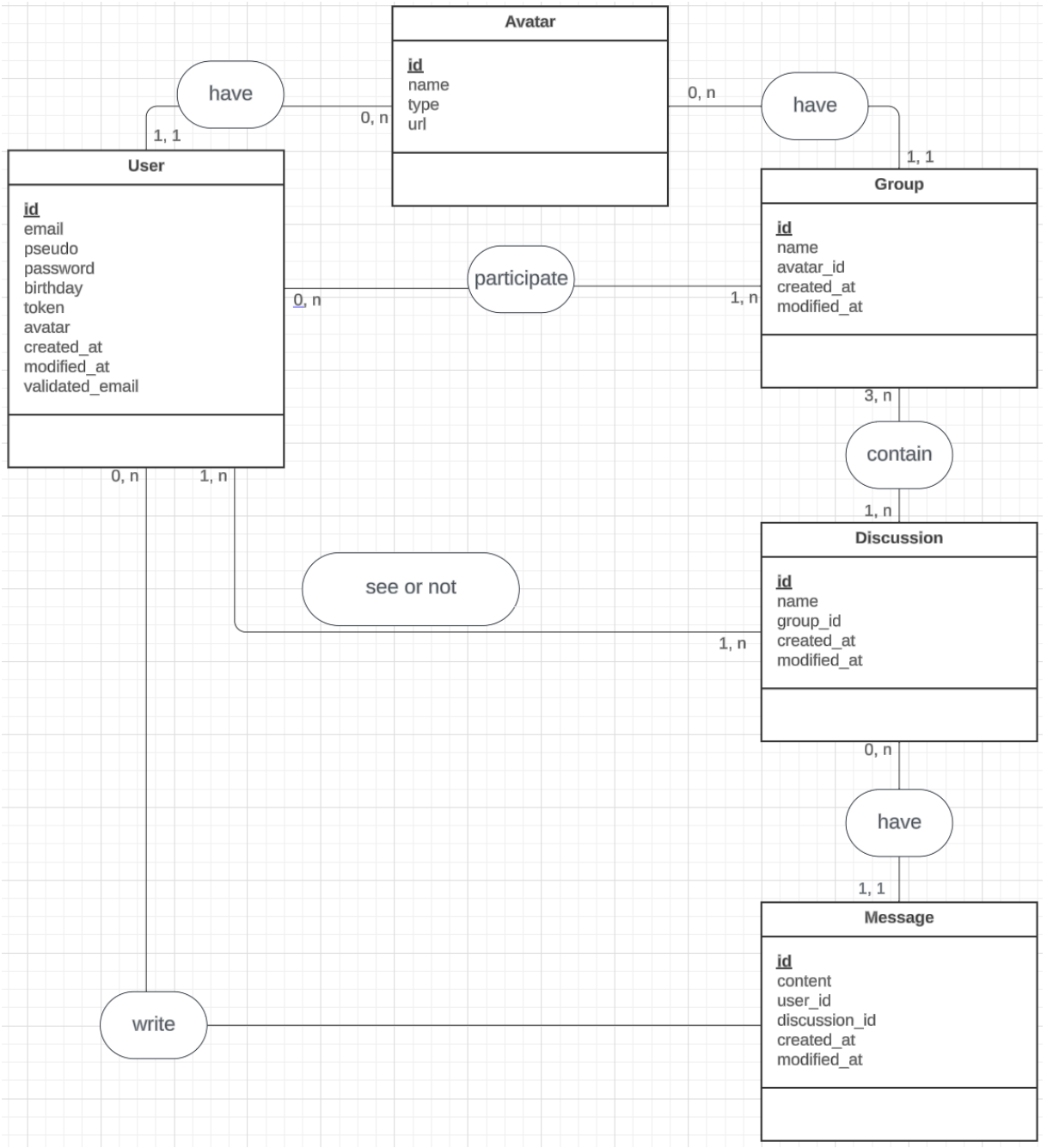
Le Modèle Conceptuel de Données (MCD) est une représentation abstraite des entités, de leurs attributs et des relations entre elles dans un système d'information, permettant de structurer et comprendre les données sans se soucier des détails techniques.Ce schéma vise à visualiser les différentes tables de données de notre application en les reliant par des verbes d’action, afin de créer des liens logiques entre elles.

Le Modèle Physique de Données (MPD) est la représentation détaillée de la manière dont les données seront effectivement stockées dans la base de données. Il précise la structure des tables, les types de données, les index, les contraintes, ainsi que les clés primaires et étrangères. Le MPD prend en compte les spécificités du système de gestion de bases de données (SGBD) choisi et optimise la performance du stockage et de l'accès aux données.

Le MCD :

Ce schéma vise à visualiser les différentes tables de données de notre application en les reliant par des verbes d'action, afin de créer des liens logiques entre elles.

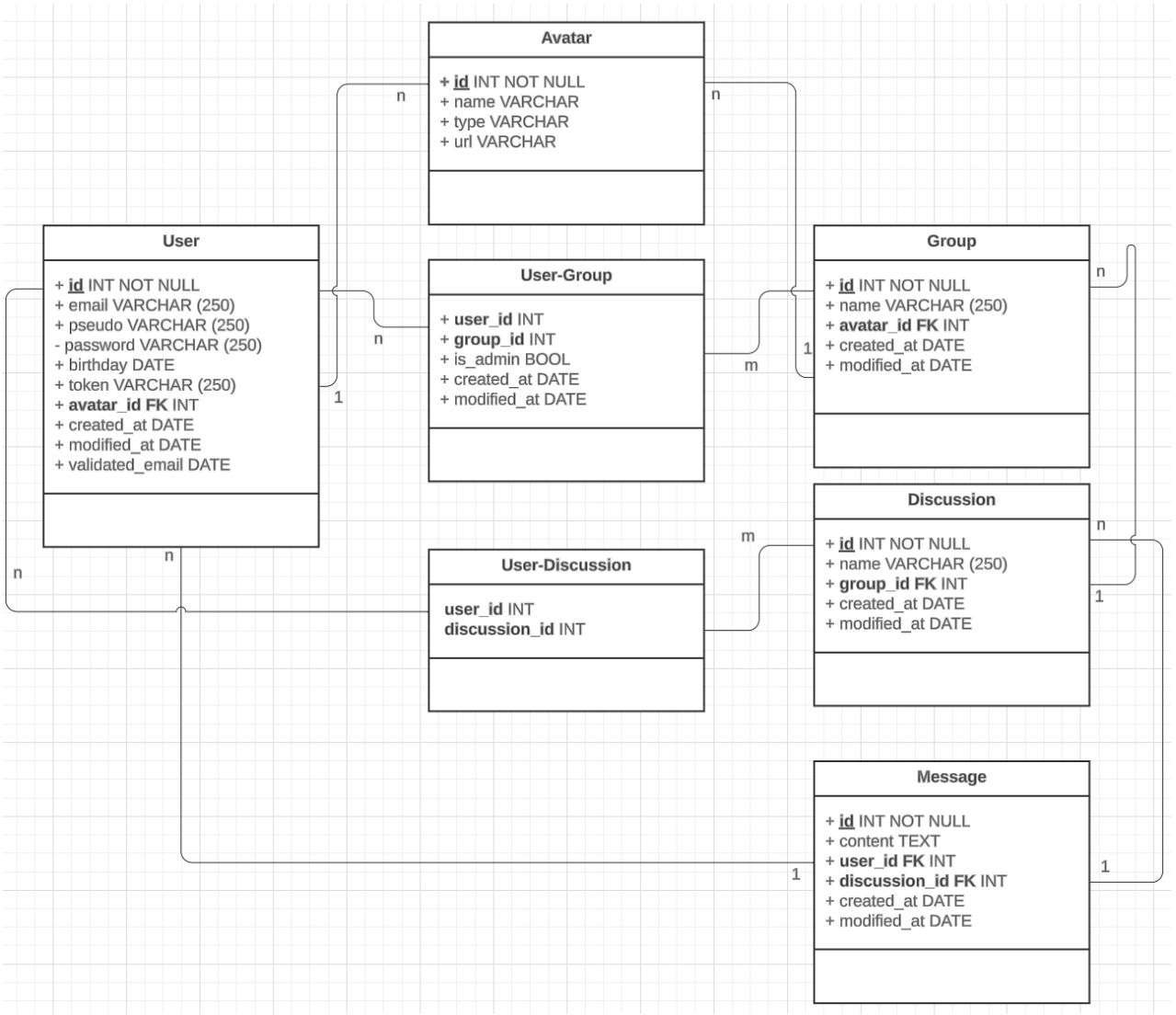
Schéma de données MCD



Le MPD :

J'ai retiré les verbes de liaison du MCD pour ne conserver que les plus grandes cardinalités. Les tables sont reliées entre elles par des liaisons one-to-many et many-to-one. Ce schéma met en évidence les relations one-to-many, many-to-one, et many-to-many, ainsi que les clés étrangères associées à chaque table. Les relations many-to-many sont gérées par une table de jointure qui contient les clés étrangères des deux parties concernées. Il s'agit du schéma de données le plus complet, où sont indiqués les types de données stockées ainsi que leur visibilité, précisant si les données seront publiques (+) et accessibles via une requête, ou privées (-) et protégées contre un accès direct.

Schéma de données MPD



Mise en place de la base de données

Pour mettre en place la base de données, j'ai commencé par installer TypeORM, un ORM (Object-Relational Mapping) pour Node.js et TypeScript qui facilite l'interaction avec les bases de données en mappant les tables à des classes, permettant ainsi de gérer les données sans avoir à écrire de SQL directement. De plus, en utilisant son système de requêtes, TypeORM offre une sécurité accrue en empêchant les injections SQL grâce à l'échappement automatique des données insérées en base.

Dans un premier temps, j'ai créé une instance de **`DataSource`** pour configurer et établir une connexion avec ma base de données Postgres.

instance db db.ts

```ts
const db = new DataSource({
    type: 'postgres',
    host: process.env.DB_HOST || 'db',
    port: parseInt(process.env.DB_PORT || '0') || 5432,
    username: process.env.DB_USER || 'postgres',
    password: process.env.DB_PASS || 'postgres',
    database: process.env.DB_NAME || 'postgres',
    synchronize: true,
    logging: true,
})
```

Je reviendrai plus tard sur la création des premières entités lorsque j'aborderai la partie sur l'authentification.

Je poursuis en créant une fonction de nettoyage de la base de données, dont le but est de supprimer toutes les données. Cette fonction est utile lors du redémarrage du projet pour garantir qu'aucune donnée résiduelle ne reste en base.

clear db db.ts

```ts
export async function clearDB() {
    const runner = db.createQueryRunner()
    await runner.query("SET session_replication_role = 'replica'")

    await Promise.all(
        db.entityMetadatas.map(async entity =>
            runner.query(
                `ALTER TABLE "${entity.tableName}" DISABLE TRIGGER ALL`
            )
        )
    )
    await Promise.all(
        db.entityMetadatas.map(async entity =>
            runner.query(`DROP TABLE IF EXISTS "${entity.tableName}" CASCADE`)
        )
    )
    await runner.query("SET session_replication_role = 'origin'")
    await db.synchronize()
}

export default db
```

Je peux maintenant écrire la fonction qui réinitialise la base de données en y ajoutant un jeu de données contrôlées et fiables, permettant ainsi de redémarrer l'application à tout moment avec un ensemble de données connues et valides.

Fonction de réinitialisation de la base de données

backend > src > TS resetDb.ts > ...

```ts
import db from './db'
import { User } from './entities/user'
import { Avatar } from './entities/avatar'
import { clearDB } from '../src/db'
import initAvatars from './initAvatars'
import { fakerFR as faker } from '@faker-js/faker'
import { groupFactory } from './utils/groupFactory'

export async function main() {
    await db.initialize()
    await clearDB()

    await initAvatars()

    const avatarIds = {
        Pierre: 1,
        Aurelie: 15,
        Olga: 3,
        Morgane: 4,
        Leopold: 5,
        Jeremie: 7,
    }

    for (const [pseudo, avatarId] of Object.entries(avatarIds)) {
        // Récupérez l'avatar correspondant à l'ID spécifié

        const avatar = await Avatar.findOne({ where: { id: avatarId } })
        // Vérifiez si l'avatar correspondant à l'ID existe
        if (!avatar) {
            console.error(`Avatar with ID ${avatarId} not found`)
            continue // Passez à l'utilisateur suivant si l'avatar n'est pas trouvé
        }

        // Créez l'utilisateur en lui associant l'avatar récupéré
        const user = User.create({
            pseudo: pseudo,
            email: `${pseudo.toLowerCase()}@gmail.com`,
            password: 'test@1234',
            avatar: avatar,
        })

        await user.save()
    }
    //on récupère la liste des users fraichement crées
    const users = await User.find()
    for (const user of users) {
        //on détermine le nombre de groupe à creer pour chaque users
        const numberOfGroups = faker.number.int({ min: 5, max: 15 })

        Array.from({ length: numberOfGroups }).forEach(() => {
            //pour chaque groupe on détermine le nombre de users qui le composeront
            const numberOfGroupUser = faker.number.int({ min: 2, max: 15 })
            try {
                groupFactory(user, numberOfGroupUser)
            } catch {
                console.error(`Error creating group for user ${user.pseudo}`)
            }
        })
    }
}

main()
```

Dans l'ordre d'exécution:

- j'initialise mon instance de base de données
- je supprime son contenu
- je crée l'ensemble des avatar
- je crée un utilisateur avec un pseudo, un email, un mot de passe et un avatar et je le sauvegarde en base, j'effectue cette opération autant de fois que j'ai d'utilisateur à créer
- puis je viens créer des groupes pour chacun de mes utilisateurs

Configurer la base commune du projet

Dans le but de développer dans un environnement uniforme et cohérent, nous avons pris soin de configurer ensemble les bases du projet dès le départ. Cette approche collaborative a été essentielle pour assurer une intégration harmonieuse entre les différentes parties du système, tout en garantissant que chaque membre de l'équipe puisse travailler dans des conditions similaires, réduisant ainsi les risques d'incompatibilités ou de conflits. Cette coordination initiale a également favorisé une communication fluide et une compréhension commune des objectifs du projet, créant ainsi un socle solide pour la suite du développement.

Nous avons commencé par la mise en place de la base de données, un élément fondamental qui a été soigneusement planifié. Nous avons choisi les technologies les plus adaptées en fonction des besoins spécifiques du projet, en tenant compte de la performance, de la scalabilité et de la sécurité. La structure de la base de données a été conçue pour être robuste et flexible, permettant une gestion efficace des données et une réponse rapide aux requêtes. Cette étape cruciale a servi de fondation solide pour toutes les autres couches de l'application, assurant ainsi une cohérence et une fiabilité tout au long du processus de développement.

Une fois la base de données en place, nous avons poursuivi avec le développement du backend. Cette phase a impliqué la définition des API, des services, et des logiques métier qui gèrent les interactions entre le frontend et la base de données. Le backend a été conçu pour être modulaire et évolutif, permettant des ajouts et des modifications futures sans compromettre la stabilité de l'application. Chaque API a été documentée avec soin, facilitant ainsi son utilisation par le frontend et garantissant une communication fluide et sécurisée entre les différentes couches de l'application.

Ensuite, nous avons abordé la création du frontend, en sélectionnant un framework moderne et performant qui nous permettrait de construire une interface utilisateur réactive, intuitive, et conviviale. Le choix du framework a été guidé par des critères de performance, de maintenabilité, et de compatibilité avec le backend. Chaque composant de l'interface a été soigneusement conçu pour interagir efficacement avec le backend, assurant ainsi une expérience utilisateur fluide et cohérente. Nous avons également pris en compte l'accessibilité et la réactivité de l'interface, pour offrir une expérience optimale sur tous types de dispositifs, qu'il s'agisse de mobiles, tablettes, ou ordinateurs de bureau.

Parallèlement à ces développements, nous avons mis en place un environnement de développement conteneurisé avec Docker. Ce choix stratégique s'est avéré être un atout majeur pour le projet, en nous permettant de standardiser les environnements de développement, de test, et de production. En encapsulant l'ensemble des dépendances et des configurations nécessaires au bon fonctionnement de l'application, Docker a non seulement simplifié la gestion des environnements, mais a également grandement amélioré la portabilité du code. Grâce à Docker, nous avons pu éliminer les problèmes courants liés aux différences de configuration entre les environnements, souvent résumés par le fameux "ça marche chez moi". De plus, cette conteneurisation a facilité le déploiement de l'application sur différentes infrastructures, assurant ainsi une plus grande fiabilité et une continuité sans faille du processus de développement à la production.

Cette configuration globale a été déterminante pour créer un environnement de travail stable, évolutif, et propice à un développement collaboratif efficace. En prenant le temps de définir et de mettre en place ces fondations dès le départ, nous avons non seulement minimisé les risques d'erreurs, mais nous avons également optimisé la productivité de l'équipe. Chaque membre a pu travailler dans un environnement cohérent, ce qui a facilité la collaboration et permis de respecter les délais du projet tout en maintenant un haut niveau de qualité. Cette approche méthodique et prévoyante a permis de garantir la réussite du projet, en créant une base solide sur laquelle nous pourrons continuer à construire et à faire évoluer l'application à l'avenir.

Le backend

Pour faire fonctionner le backend, il est essentiel de commencer par créer le schéma GraphQL en utilisant TypeGraphQL. Ce schéma sert de base pour structurer les requêtes et mutations qui interagiront avec les données de notre application. En utilisant TypeGraphQL, je peux définir les types de données et les relations entre eux de manière déclarative. Ces types de données sont ensuite récupérés par les résolveurs, qui agissent comme des intermédiaires entre le schéma GraphQL et les entités gérées par TypeORM.

L'intégration de TypeGraphQL avec TypeORM présente un avantage majeur : elle permet de centraliser la définition des types dans un seul endroit, ce qui garantit une cohérence totale entre les entités de la base de données et les types utilisés dans l'API GraphQL. Cela signifie que les typages TypeScript sont générés automatiquement à partir des entités, éliminant ainsi la nécessité de les définir plusieurs fois et réduisant le risque d'erreurs. Cette approche permet également de tirer parti de la puissance de TypeScript pour obtenir une validation des types en temps réel, améliorant ainsi la fiabilité et la maintenabilité du code. En résumé, ce processus simplifie grandement la gestion des données dans l'application et assure une intégration fluide entre les différentes couches du backend.

Schéma type-graphql schema.ts

```
11  export default buildSchema({
12      resolvers: [
13          UsersResolver,
14          GroupsResolver,
15          AvatarsResolver,
16          UsersToGroupsResolver,
17          DiscussionResolver,
18          MessageResolver,
19      ],
20      authChecker: customAuthChecker,
21  })
```

Ensuite, j'initialise mon serveur Apollo dans le fichier **`index.ts`** en utilisant le schéma que j'ai créé précédemment. Une fois cette étape terminée, les données, via l'instance de la base de données et le schéma généré, seront accessibles depuis Apollo Studio. Cette interface permet de tester visuellement les requêtes de manière instantanée via une API. Je reviendrai sur ce point plus en détail lors du développement de la fonctionnalité de création de groupe.

Initialisation serveur Apollo index.ts

```
41
42  const port = 4001
43
44  schema.then(async schema => {
45      await db.initialize()
46      const serverCleanup = useServer({ schema }, wsServer)
47      const server = new ApolloServer<MyContext>({
48          schema,
49          csrfPrevention: true,
50          cache: 'bounded',
51          plugins: [
52              ApolloServerPluginDrainHttpServer({ httpServer }),
53              ApolloServerPluginLandingPageLocalDefault({ embed: true }),
54              {
55                  async serverWillStart() {
56                      return {
57                          async drainServer() {
58                              await serverCleanup.dispose()
59                          },
60                      }
61                  },
62              },
63          ],
64      })
65
66      await server.start()
67
68      const { url } = await startStandaloneServer(server, { listen: { port } })
69
70      console.log(`🚀 Server ready at ${url}`)
71
72  })
```

Mon API Apollo est désormais accessible à l'adresse '**http://localhost:4001**'.

Le frontend

Il est maintenant temps de configurer une interface client pour interagir avec notre API fournie par le backend. Pour cela, j'ai commencé par initialiser un projet Next.js, qui offre un environnement React optimisé pour le rendu côté serveur. Next.js se distingue par sa simplicité dans la gestion du routage, grâce à un système basé sur les fichiers, où chaque nouvelle page ajoutée dans le dossier **`pages`** crée automatiquement une nouvelle route.

Cela permet une organisation claire et une navigation intuitive à travers l'application, tout en profitant des performances accrues offertes par le rendu côté serveur.

Afin de rendre notre API accessible depuis cette interface client, il est crucial d'installer Apollo Client. Apollo Client fonctionne comme un provider qui, une fois intégré, permet à l'ensemble de l'application de se connecter facilement à l'API et de gérer le contexte global. En encapsulant toute l'application avec ce provider, chaque composant peut accéder aux requêtes GraphQL et aux données associées de manière centralisée. Cette configuration non seulement simplifie la gestion des données, mais aussi la gestion des états et des erreurs au sein de l'application, améliorant ainsi l'efficacité et la maintenabilité du code.

En complément, nous avons choisi d'utiliser Codegen, un outil extrêmement puissant qui génère automatiquement du code à partir de requêtes GraphQL. Avec Codegen, à partir d'une simple requête, un hook personnalisé est automatiquement créé. Ce hook prend en charge non seulement l'exécution de la requête, mais aussi la gestion des données, de l'état, des erreurs, et bien plus encore. Cette automatisation réduit considérablement la charge de travail lors du développement, tout en garantissant que le code reste propre, cohérent et facile à maintenir. Je reviendrai sur l'utilisation spécifique de ces hooks, notamment lors de l'implémentation de la fonctionnalité de création d'un groupe.

À la racine de notre application Next.js, dans le fichier **`_app.tsx`**, j'ai encapsulé l'application avec le provider Apollo. Cette étape est essentielle pour établir et maintenir une connexion robuste entre le client et l'API, permettant à chaque composant de l'application d'effectuer des requêtes GraphQL et de gérer l'état de manière cohérente. Cette encapsulation centralisée favorise une communication fluide entre le frontend et le backend, tout en optimisant les performances globales de l'application et en facilitant la gestion des erreurs et des données.

L'utilisation de Docker dans notre projet a également été un choix stratégique. Docker permet de créer un environnement de travail reproductible avec une configuration commune, ce qui est particulièrement avantageux pour les projets en équipe. Il garantit que le code s'exécute de manière identique sur chaque machine de développement, éliminant ainsi les problèmes liés aux différences de versions de dépendances ou de systèmes d'exploitation. Cette homogénéité est cruciale pour maintenir la cohérence du développement et éviter les surprises lors des phases de déploiement.

Pour simuler le lancement de l'ensemble de l'application, y compris le backend et le frontend, j'ai défini un script d'installation et de démarrage. Ce script joue un rôle crucial en s'assurant que tous les composants de l'application sont déployés et fonctionnent de

manière cohérente à travers les différents environnements. En automatisant ce processus, nous facilitons non seulement le développement quotidien, mais aussi le déploiement, en garantissant que chaque composant de l'application est opérationnel et interconnecté de manière optimale.

Dockerfile frontend

Je vais détailler le script Docker pour la partie frontend.

frontend > Dockerfile

```
1   FROM node:20.9.0-alpine3.17
2
3   WORKDIR /app
4   COPY package*.json ./
5   RUN npm i
6   COPY tsconfig.json tsconfig.json
7   COPY tailwind.config.ts tailwind.config.ts
8   COPY postcss.config.js postcss.config.js
9   COPY next.config.mjs next.config.mjs
10  COPY components.json components.json
11  COPY public public
12  COPY src src
13
14  CMD npm run dev
```

Je viens définir l'environnement d'exécution
Je crée un dossier app
Je copie l'ensemble du package json à la racine de app

Je copie les fichiers qui ont nécessité à être dupliqué dans dans l'environnement de docker pour le hot reload

Puis je lance la commande de démarrage

Dockerfile dossier frontend

Je réalise la même opération pour mon dossier backend, puis je configure le script d'orchestration de mon application. Ce script a pour but de lancer, dans un ordre précis, les différentes images Docker afin de créer un conteneur complet. Il permet également de spécifier d'éventuelles variables d'environnement, de créer des volumes (c'est-à-dire des ponts entre l'environnement virtuel Docker et la machine hôte pour transférer et préserver les données), et surtout, d'exposer les ports afin de rendre les services accessibles.

Schéma type-graphql schema.ts

docker-compose.yml

```yaml
services:
  db:
    image: postgres:15
    environment:
      - POSTGRES_PASSWORD=postgres
    ports:
      - 5432:5432

  backend:
    build: ./backend
    ports:
      - 4001:4001
    volumes:
      - ./backend/src:/app/src
      - ./backend/src/db.ts:/app/src/db.ts
    env_file:
      - ./global.env
      - ./backend/.env
    environment:
      - DB_HOST=db
    healthcheck:
      test:
        [
          'CMD-SHELL',
          "curl -f http://backend:4001/graphql?query=%7B__typename%7D
          -H 'Apollo-Require-Preflight: true' || exit 1",
        ]
      interval: 10s
      timeout: 30s
      retries: 5

  frontend:
    build: ./frontend
    ports:
      - 3000:3000
    volumes:
      - ./frontend/tailwind.config.ts:/app/tailwind.config.ts
      - ./frontend/src:/app/src
    env_file:
      - ./frontend/.env
      - ./global.env
```

J'utilise l'image postgres 15

J'expose le port 5432

Je construis l'image du backend

J'expose le port 4001

Je monte les fichiers sur lesquels je veux du temps réel

Je charge les variables d'environnement

Je définis l'host de connexion à la base de données

Je vérifie que l'API est disponible

Je construis l'image frontend

J'expose le port 3000
Je monte les fichiers pour le temps réel

Je charge les variables d'environnement

En utilisant Docker, je suis tenu d'employer des images d'environnement prédéfinies et standardisées, comme ici avec Postgres. Cela garantit que toute personne exécutant le

script **`"dev": "docker compose up --build"`**, disponible dans le fichier **`package.json`** à la racine, lancera la base de données dans des conditions exactement identiques.

Authentification

Maintenant que le backend et le frontend sont installés et configurés, nous arrivons à une étape cruciale du développement : la conception de la base commune pour l'authentification. Cette phase est essentielle non seulement pour garantir une expérience utilisateur fluide, mais aussi pour assurer la sécurité de l'application, un aspect fondamental dans tout projet logiciel.

Importance de l'authentification

L'authentification est le processus qui permet de vérifier l'identité d'un utilisateur avant de lui accorder l'accès aux fonctionnalités de l'application. Elle joue un rôle clé dans la protection des données sensibles et dans le contrôle des accès, en s'assurant que seuls les utilisateurs autorisés peuvent interagir avec certaines parties de l'application.

Processus d'authentification

Pour mettre en place une authentification efficace, il est important de structurer le processus de manière logique et sécurisée :

Inscription :

- **Collecte des informations utilisateur :** Lors de l'inscription, l'utilisateur fournit des informations de base comme son nom, son adresse email et un mot de passe. Ces données sont ensuite stockées dans la base de données de manière sécurisée, avec des mots de passe hashés pour éviter tout risque en cas de fuite de données.
- **Validation des données :** Le processus d'inscription peut inclure une validation des données, comme la vérification de la validité de l'adresse email via un lien de confirmation envoyé à l'utilisateur. Cette étape assure que l'utilisateur est bien propriétaire de l'email fourni.

Connexion :

- **Vérification des identifiants :** Lors de la connexion, l'utilisateur entre ses identifiants (email et mot de passe). Le backend compare les informations fournies avec celles stockées dans la base de données pour vérifier l'identité de l'utilisateur. Si les identifiants sont corrects, une session est initiée.
- **Gestion des erreurs :** Si les identifiants ne correspondent pas, l'application renvoie un message d'erreur spécifique, évitant les messages génériques qui pourraient donner des indices aux attaquants potentiels.

Maintien de la session :

Cookies : Une fois connecté, l'utilisateur peut rester connecté grâce à l'utilisation de cookie qui contient le token d'authentification JWT. Ce mécanisme permet de maintenir une session utilisateur active tout en assurant une sécurité optimale. Par exemple, un token JWT peut être signé et envoyé au frontend, où il est stocké localement pour autoriser les requêtes ultérieures sans nécessiter une nouvelle connexion à chaque fois.

Expiration des sessions : Pour renforcer la sécurité, il est important de définir une durée d'expiration pour les sessions ou les tokens. Une session peut expirer après un certain temps d'inactivité, obligeant l'utilisateur à se reconnecter, ce qui réduit les risques d'accès non autorisé.

Pour cela, je vais avoir besoin de plusieurs éléments :

- Une entité utilisateur qui correspond à la table **`user`** dans notre MPD.
- Un resolver pour effectuer des actions sur les données.
- Une interface utilisateur pour interagir avec l'application.
- Enfin, il sera nécessaire de mettre en place une logique d'authentification sécurisée, idéalement avec la création d'une session pour l'utilisateur afin de lui éviter de devoir se reconnecter à chaque visite.

Entité User

Je procède donc à la création de l'entité **`user`**, en y incluant les champs définis lors de la conception du MPD, ainsi que les types de données correspondants. Cette entité représente la table de base, contenant les éléments essentiels pour la création d'un utilisateur. Elle reflète l'état initial nécessaire pour mettre en place l'authentification. Par la suite, je reviendrai sur cette entité pour y ajouter les relations avec les groupes, afin de compléter la structure de l'application.

J'utilise différents décorateurs de TypeGraphQL : le décorateur **`@Field`** sert à rendre un champ accessible dans le schéma GraphQL, tandis que **`@ObjectType`** permet de transformer les types d'une entité en types TypeScript. Les décorateurs TypeORM **`@BeforeInsert`** et **`@BeforeUpdate`** servent à exécuter une opération avant l'enregistrement en base de données. Le décorateur **`@Column`** définit une colonne dans la base de données et peut accepter des paramètres de contrainte SQL, comme la longueur ou l'unicité. Enfin, les décorateurs de relation comme **`@ManyToOne`** spécifient l'entité associée, la clé de liaison, ainsi que les conditions de suppression.

entité utilisateur
user.ts

backend > src > entities > TS user.ts > ...

```ts
import { Field, InputType, Int, ObjectType } from 'type-graphql'
import { Avatar } from './avatar'
import {
    BaseEntity,
    BeforeInsert,
    BeforeUpdate,
    Column,
    CreateDateColumn,
    Entity,
    ManyToOne,
    PrimaryGeneratedColumn,
    UpdateDateColumn,
} from 'typeorm'
import * as argon2 from 'argon2'
import { ObjectId } from '../utils'
import { IsDateString, IsEmail, IsNotEmpty, Length } from 'class-validator'

@Entity()
@ObjectType()
export class User extends BaseEntity {
    @BeforeInsert()
    @BeforeUpdate()
    protected async hashPassword() {
        if (!this.password.startsWith('$argon2')) {
            this.password = await argon2.hash(this.password)
        }
    }
    @Field(() => Int)
    @PrimaryGeneratedColumn()
    id: number

    @Field()
    @Column({ unique: true })
    email: string

    @Column({ length: 50 })
    @Field()
    pseudo: string

    @Column()
    password: string

    @ManyToOne(() => Avatar, avatar => avatar.users, {
        cascade: true,
        onDelete: 'CASCADE',
    })
    @Field(() => Avatar, { nullable: true })
    avatar: Avatar | null

    @Field()
    @CreateDateColumn()
    created_at: string

    @Field()
    @UpdateDateColumn()
    modified_at: string
}

```

Il me faut maintenant ajouter l'entité créée à mon instance db.ts dans le champ “entities” autrement au moment de la compilation TypeORM me signalera une erreur.

Authentification JWT

Pour l'authentification, je vais procéder étape par étape en mettant en place un système sécurisé basé sur JSON Web Tokens (JWT). Ce système d'authentification repose sur la création et la vérification de tokens, qui permettent de maintenir une session utilisateur sécurisée sans nécessiter le stockage d'informations sensibles côté serveur.

Création d'un utilisateur

La première étape consiste à créer un utilisateur à partir des données fournies par le frontend, telles que l'email, le pseudo, et le mot de passe. La fonction **`Register`**, qui gère la création de l'utilisateur, prend ces données et crée un nouvel enregistrement dans la base de données.

- **Hashage du mot de passe :** Avant de stocker le mot de passe dans la base de données, le décorateur **`@BeforeInsert`** est utilisé pour s'assurer que le mot de passe est haché à l'aide de la librairie **`argon2`**. Cela signifie que le mot de passe n'est jamais stocké en clair, mais sous forme de hash, une chaîne de caractères indéchiffrable. Le hashage ajoute une couche de sécurité importante, car même en cas de fuite de données, les mots de passe réels restent protégés.
- **Unicité des utilisateurs :** Avant de finaliser l'inscription, il est essentiel de vérifier que l'email ou le pseudo de l'utilisateur n'existent pas déjà dans la base de données, évitant ainsi les doublons et assurant que chaque utilisateur à des identifiants uniques.

Une fois l'utilisateur créé, il est prêt à se connecter, et c'est là que les JWT entrent en jeu.

Implémentation de JWT pour la connexion

Lors de la phase de connexion, après avoir validé les identifiants de l'utilisateur (en comparant le hash du mot de passe fourni avec celui stocké dans la base de données), un JWT est généré pour cet utilisateur. Ce token contient des informations essentielles, telles que l'ID de l'utilisateur, et est signé avec une clé secrète pour garantir son authenticité.

- **Création du JWT :** Le JWT est créé en utilisant une clé secrète propre à l'application. Cette clé est utilisée pour signer le token, garantissant ainsi que le serveur pourra vérifier que le token n'a pas été altéré. Le token est ensuite renvoyé au frontend dans un cookie en modification stricte httpOnly qui signifie qu'il ne pourra être modifié par le client.
- **Durée de vie du token :** Le JWT inclut une durée de vie (**`exp`**), après laquelle il devient invalide. Cela permet de limiter la durée pendant laquelle un token volé peut être utilisé, renforçant ainsi la sécurité.

Middleware pour la vérification des JWT

Avant de permettre l'accès aux routes protégées de l'application, il est nécessaire de vérifier que la requête est accompagnée d'un JWT valide. C'est ici qu'intervient le middleware.

- **Vérification du token :** Le middleware agit comme un filtre pour toutes les requêtes entrantes. Il vérifie d'abord la présence d'un JWT dans les en-têtes de la requête ou dans les cookies. Si le token est présent, le middleware le décode et valide sa signature en utilisant la même clé secrète que celle utilisée lors de la création du token.
- **Gestion des erreurs :** Si le token est absent ou invalide (par exemple, s'il a été altéré ou a expiré), le middleware renvoie une réponse d'erreur, empêchant ainsi l'accès aux routes protégées. Cela garantit que seules les requêtes authentifiées et valides peuvent accéder aux fonctionnalités sécurisées de l'application.

```
app.use(
    '/',
    cors<cors.CorsRequest>({
        origin: [
            'http://localhost:3000',
            'https://studio.apollographql.com',
            'https://staging.0923-bleu-3.wns.wilders.dev/',
        ],
        credentials: true,
    }),
    express.json(),
    expressMiddleware(server, {
        context: async ({ req, res }) => {
            let user: User | null = null

            const cookies = new Cookies(req, res)
            const token = cookies.get('token')

            if (token) {
                try {
                    const verify = await jwtVerify<Payload>(
                        token,
                        new TextEncoder().encode(process.env.SECRET_KEY)
                    )
                    user = await findUserByEmail(verify.payload.email)
                } catch (error) {
                    console.error('Error during JWT verification, ', error)
                }
            }
            return { req, res, user }
        },
    })
)
```

Middleware serveur apollo index.ts

Il est aussi utile de préciser l'ajout de cors qui est ici pour bloquer toute requête dont l'origine serait étrangère à la liste fournie

Pour mettre en place ce système d'authentification sécurisé, j'ai besoin d'un serveur Express, qui est essentiel pour intégrer le middleware à notre serveur Apollo. Express me permet d'utiliser **`expressMiddleware`**, une fonction clé pour intercepter les requêtes entrantes et vérifier leur légitimité avant de les passer à l'application principale.

Implémentation du Middleware avec Express

Le middleware joue un rôle central dans le contrôle d'accès aux différentes parties de l'application. À chaque arrivée de requête, le middleware exécute plusieurs tâches cruciales :

- **Recherche du token :** Dès qu'une requête atteint le serveur, le middleware commence par vérifier la présence d'un token dans les cookies de la requête. Ce token est une version cryptée et signée des informations d'identification de l'utilisateur, que j'aborderai plus en détail lors de la discussion sur sa création. La recherche de ce token est la première étape pour déterminer si la requête provient d'un utilisateur authentifié.
- **Vérification de la validité du token :** Si un token est trouvé, le middleware le décode en utilisant la même clé secrète qui a servi à le créer. Cette clé est cruciale car elle garantit l'intégrité du token : si le token a été altéré ou falsifié, la vérification échouera, et la requête sera rejetée. Cette étape garantit que seules les requêtes provenant d'utilisateurs authentifiés peuvent accéder aux ressources protégées.
- **Extraction et validation de l'utilisateur :** Si le token est valide, j'extrais l'email ou l'ID utilisateur encodé dans le token. Avec cette information, je recherche l'utilisateur correspondant dans la base de données. Si un utilisateur correspondant est trouvé, ses données sont ajoutées au contexte de la requête, le rendant ainsi disponible pour toute opération ultérieure dans le cycle de vie de la requête. Cela signifie que les données de l'utilisateur, telles que ses permissions ou préférences, sont directement accessibles par les résolveurs GraphQL ou autres couches de l'application, facilitant ainsi la gestion des sessions et des autorisations.

Avantages de cette approche

L'intégration du middleware avec Express et Apollo présente plusieurs avantages significatifs pour la sécurité et la personnalisation de l'application :

- **Sécurité renforcée :** En vérifiant systématiquement la validité des tokens à chaque requête, le middleware joue un rôle crucial dans la protection de l'application contre les accès non autorisés. Cette vérification garantit que seules les requêtes authentifiées et légitimes peuvent accéder aux fonctionnalités critiques, réduisant ainsi les risques d'exploitation par des utilisateurs malveillants.
- **Contexte utilisateur dynamique :** L'ajout de l'utilisateur authentifié au contexte de la requête permet une personnalisation des réponses en fonction de l'utilisateur courant. Cette approche rend l'application plus réactive et centrée sur l'utilisateur, en permettant par exemple de filtrer les données ou d'afficher des contenus spécifiques en fonction des permissions et des préférences de l'utilisateur.

Lorsque les données de l'utilisateur sont transmises dans le contexte de la requête, cela permet également de valider le **`authChecker`**, un mécanisme qui contrôle l'accès aux routes protégées par le décorateur **`@Authorized()`**. Ce décorateur est utilisé pour restreindre l'accès à certaines fonctionnalités de l'application, s'assurant que seuls les

utilisateurs authentifiés et autorisés peuvent les utiliser. Si aucun token valide n'est trouvé, la requête continue son traitement sans utilisateur associé, limitant ainsi l'accès aux ressources protégées.

Processus d'authentification

login userResolver.ts

backend > src > resolvers > TS usersResolver.ts > UsersResolver > login

```
60   class UsersResolver {
109      @Query(() => ResponseMessage)
110      async login(@Arg('infos') infos: InputLogin, @Ctx() ctx: MyContext) {
111          const user = await findUserByEmail(infos.email)
112
113          if (!user) {
114              throw new GraphQLError(`Veuillez vérifier vos informations`)
115          }
116
117          const isPasswordValid = await argon2.verify(
118              user.password,
119              infos.password
120          )
121
122          const responseMessage = new ResponseMessage()
123          if (isPasswordValid) {
124              const token = await new SignJWT({ email: user.email })
125                  .setProtectedHeader({ alg: 'HS256', typ: 'jwt' })
126                  .setExpirationTime('2h')
127                  .sign(new TextEncoder().encode(`${process.env.SECRET_KEY}`))
128
129              const cookies = new Cookies(ctx.req, ctx.res)
130              cookies.set('token', token, { httpOnly: true })
131
132              responseMessage.message = 'Bienvenue!'
133              responseMessage.success = true
134          } else {
135              responseMessage.message = 'Vérifiez vos informations'
136              responseMessage.success = false
137          }
138
139          return responseMessage
140      }
```

Maintenant que nous avons vu comment une requête est traitée et comment son en-tête est analysée, passons à l'authentification proprement dite. Ce processus commence par la récupération des informations envoyées par le frontend : un email et un mot de passe.

- **Recherche de l'utilisateur :** La première étape consiste à rechercher l'utilisateur correspondant à l'email fourni. Si aucun utilisateur n'est trouvé, l'application renvoie un message d'erreur vague, une mesure de sécurité visant à éviter de divulguer des

informations sensibles qui pourraient être exploitées dans des tentatives d'attaque malveillantes, comme les attaques par force brute.

- **Vérification du mot de passe :** Ensuite, le mot de passe fourni est comparé avec celui stocké en base de données. Pour ce faire, j'utilise la même librairie de hachage, **`argon2`**, qui a été utilisée lors de l'inscription pour hacher le mot de passe. Le mot de passe de la requête est haché et comparé au hash stocké dans la base de données. Si les deux correspondent, l'utilisateur est authentifié avec succès.

- **Création du token d'authentification :** Si l'authentification réussit, je crée un jeton d'authentification (JWT) à l'aide de la librairie JOSE. Ce jeton chiffre l'email de l'utilisateur en utilisant un algorithme de cryptage, lui attribue une période de validité, et le signe avec une clé secrète pour en garantir l'authenticité. Cette même clé secrète est utilisée par le middleware pour vérifier la validité du token lors des requêtes futures.

- **Stockage du token dans un cookie sécurisé :** Une fois le jeton créé, il est stocké dans un cookie nommé **`token`**. Ce cookie est configuré avec l'attribut **`HttpOnly`**, ce qui empêche toute manipulation du cookie par des scripts côté client, ajoutant ainsi une couche supplémentaire de sécurité contre les attaques de type XSS (Cross-Site Scripting). L'utilisateur est alors considéré comme connecté à l'application pour une durée de 2 heures, à moins qu'il ne se déconnecte de lui-même avant la fin de cette période.

- **Gestion des erreurs d'authentification :** Si le mot de passe ne correspond pas à celui stocké en base, l'application renvoie le même message d'erreur générique que celui utilisé lorsqu'aucun utilisateur correspondant n'est trouvé. Cette approche préserve la sécurité en ne donnant aucune indication sur la raison exacte de l'échec, compliquant ainsi les tentatives d'accès non autorisé.

Réalisation de la fonctionnalité de création de groupe

À présent que nous avons configuré et installé une base commune pour notre projet, je peux me concentrer sur le développement de la fonctionnalité de création de groupe au sein de l’application. Cette fonctionnalité nécessitera une gestion efficace des données, une logique métier robuste dans le backend, ainsi qu'une interface utilisateur intuitive pour permettre aux utilisateurs d'interagir avec cette fonctionnalité côté frontend.

1 Base de données

Pour commencer, je dois mettre à jour l'entité **`User`** afin d'ajouter les éléments nécessaires à l'intégration des groupes dans notre application. Il est crucial de respecter les relations entre les tables de données définies dans le Modèle Physique de Données (MPD) pour garantir l'intégrité et la cohérence des données.

- **Relation utilisateur-groupe :** La première étape consiste à établir la relation entre l'utilisateur et le nouveau groupe. Pour cela, je vais créer une table de jointure personnalisée, appelée **`UserToGroup`**, qui contiendra les clés étrangères de l'utilisateur (**`User`**) et du groupe (**`Groupe`**). Cette table de jointure joue un rôle central dans la gestion des accès aux groupes, car elle permet de déterminer quels utilisateurs appartiennent à quels groupes.

- **Indication du rôle d'administrateur :** En plus des clés étrangères, la table **`UserToGroup`** inclura un champ supplémentaire pour indiquer si l'utilisateur est l'administrateur du groupe, c'est-à-dire la personne à l'origine de sa création. Ce champ est important car il détermine les privilèges spécifiques de l'utilisateur au sein du groupe, comme la capacité à inviter d'autres utilisateurs ou à modifier les paramètres du groupe.

- **Implémentation de la relation OneToMany :** Pour lier l'utilisateur à la table de jointure **`UserToGroup`**, j'implémente une relation **`OneToMany`**. Cela signifie qu'un utilisateur peut être lié à plusieurs enregistrements dans **`UserToGroup`**, chaque enregistrement représentant un groupe auquel il appartient. Cette relation est définie dans l'entité **`User`**, et elle est reflétée dans la table **`UserToGroup`** par la présence de la clé primaire de l'utilisateur. Cette configuration permet au schéma de générer un tableau **`UserToGroup`**, qui représentera tous les groupes auxquels l'utilisateur a accès, facilitant ainsi la gestion des appartenances et des rôles au sein de l'application.

```
@Field(() => [UserToGroup])
@OneToMany(() => UserToGroup, userToGroup => userToGroup.user)
public userToGroups: UserToGroup[]
```

Relation OneToMany entité User user.ts

Je passe maintenant à la création de l'entité Group, qui comprend une clé primaire, un nom, une date de création, une date de modification, ainsi qu'une relation ManyToOne vers l'entité Avatar. De manière similaire, j'ajoute une relation OneToMany vers la table UserToGroup pour finaliser la liaison entre le groupe et les utilisateurs.

Entite UserToGroup userToGroup.ts

backend > src > entities > TS userToGroup.ts > ...

```ts
@ObjectType()
@Entity()
export class UserToGroup extends BaseEntity {
    @Field(() => Int)
    @PrimaryGeneratedColumn()
    public id: number

    @Field()
    @Column()
    public user_id: number

    @Field()
    @Column()
    public group_id: number

    @Field()
    @Column()
    public is_admin: boolean

    @Field()
    @CreateDateColumn()
    public created_at: string

    @Field()
    @UpdateDateColumn()
    public modified_at: string

    @Field(() => User)
    @ManyToOne(() => User, user => user.userToGroups)
    @JoinColumn({ name: 'user_id' })
    public user: User

    @ManyToOne(() => Group, group => group.userToGroups)
    @JoinColumn({ name: 'group_id' })
    public group: Group
}
```

Pour préparer la prochaine étape du développement de la fonctionnalité, je mets en place l'entité **`Discussion`**. Cette entité inclut une clé primaire, une date de création, une date de modification, ainsi qu'une relation **`ManyToOne`** vers l'entité **`Group`**, ce qui permet de lier chaque discussion à un groupe spécifique. De plus, une relation **`ManyToMany`** est établie avec l'entité **`User`**, représentant les utilisateurs participants à la discussion.

Il existe également une autre relation **`ManyToOne`** avec l'entité **`User`**, cette fois-ci pour un utilisateur particulier, appelé "utilisateur sujet", qui n'aura pas accès à la discussion. Cet utilisateur est inclus dans l'entité pour permettre un accès direct à ses données, comme son pseudo, afin de nommer la discussion de manière pertinente. Enfin, j'ajoute une relation **`OneToMany`** vers l'entité **`Message`**, qui représentera les messages échangés dans la discussion (les détails concernant les messages seront abordés ultérieurement).

```
@ManyToOne(() => Group, g => g.discussions, {
    cascade: true,
    onDelete: 'CASCADE',
})
@Field(() => Group)
group?: Group
```

On expose le groupe, et on ajoute une condition onDelete CASCADE qui signifie qu'à la suppression du groupe lié, cette discussion le sera aussi

```
@ManyToMany(() => User)
@JoinTable()
@Field(() => [User])
users?: User[]
```

Ici il s'agit de la relation exposant les utilisateurs participant, on utilise JoinTable afin de pouvoir exposer et rendre accessible dans schéma cette table, ainsi on y aura accès dans la requête.

Je n'oublie pas d'ajouter les relations relatives aux discussions dans les entités User et Group.

2 La logique métier

Maintenant que les données sont prêtes, je peux me concentrer sur le développement de la logique de création de groupe. Je vais expliquer brièvement la démarche que je vais suivre. Plutôt que de présenter l'intégralité du resolver en une seule fois, je vais vous montrer différents segments de code en détaillant leur fonction respective.

Création du Resolver

Pour commencer, je crée le resolver dédié aux groupes dans le dossier backend, nommé **`groupsResolver.ts`**. Ce resolver contiendra la logique nécessaire à la gestion des groupes. Ensuite, je mets en place une mutation appelée **`addNewGroup`**. Cette mutation est choisie plutôt qu'une requête (query) car l'action modifie la base de données en y ajoutant de nouvelles informations. La mutation est protégée par le décorateur **`@Authorized`**, qui empêche son exécution si l'utilisateur à l'origine de la requête n'est pas authentifié.

Paramètres et Sécurité

Cette fonction de mutation prend en paramètre le contexte de la requête, qui contient potentiellement les informations de l'utilisateur actuel, ainsi que les arguments fournis par l'utilisateur, typés sous la forme de **`NewGroupInput`** pour garantir que les données saisies dans la requête GraphQL sont correctes.

Vérification du Contexte

La première étape consiste à vérifier le contenu du contexte. Si aucune donnée utilisateur n'est présente, cela signifie que l'utilisateur n'est pas connecté, et une erreur est alors renvoyée pour indiquer cette absence d'authentification.

Traitement des Données

Ensuite, je récupère les données fournies par l'utilisateur : le nom du groupe, les emails des participants, et la date de l'événement. Un avatar est sélectionné de manière aléatoire, le nouveau groupe est créé, et l'avatar est associé à ce groupe.

Création de la Relation `UserToGroup`

Je crée ensuite la première relation **`UserToGroup`** en associant l'utilisateur courant au groupe nouvellement créé, tout en définissant le champ **`is_admin`** sur **`true`**, ce qui confère à cet utilisateur le statut d'administrateur du groupe.

Gestion des Participants

Je procède ensuite à l'itération sur la liste des emails des participants :

- **Utilisateur actuel :** Si l'email correspond à celui de l'utilisateur courant, aucune action supplémentaire n'est nécessaire.
- **Utilisateur existant :** Si l'email est associé à un utilisateur déjà inscrit sur la plateforme, je crée une relation **`UserToGroup`** pour cet utilisateur et je lui envoie une notification par email via notre service de messagerie. Cette notification l'informe de son ajout au groupe et lui fournit un lien d'accès direct.
- **Nouvel utilisateur :** Si l'email ne correspond à aucun utilisateur inscrit, je crée un compte pour cette personne en générant un mot de passe aléatoire. J'associe également un token à ce compte, qui sera utilisé pour la connexion ultérieure de l'utilisateur. Une relation est ensuite créée entre ce nouvel utilisateur et le groupe.

Envoi de l'Invitation

C'est ici que le token joue un rôle clé : je l'envoie par email au nouvel utilisateur, l'invitant à créer un compte sur la plateforme pour participer aux discussions. Le token est intégré dans l'URL fournie, permettant à l'utilisateur de définir un mot de passe personnalisé lors de sa première connexion.

```
@Mutation(() => UserWithoutPassword)
async registrationWithToken(
    @Arg('data', { validate: true }) data: InputRegistrationWithToken
) {
    const user = await User.findOne({ where: { token: data.token } })
    if (!user) {
        throw new GraphQLError('Aucun utilisateur trouvé avec ce token')
    }

    Object.assign(user, {
        ...data,
        token: null,
    })

    await user.save()
    return user
}
```

Mutation utilisateur pré-enregistré usersResolver.ts

Il s'agit de la mutation dans le **`userResolver`** qui permet de valider le changement de mot de passe du nouvel utilisateur ajouté lors de la création du groupe. On peut voir que l'utilisateur est récupéré à partir du token fourni dans la requête.

Attention, l'étape précédente d'itération sur les emails implique du code asynchrone. Je récupère une liste d'utilisateurs participants, mais je dois m'assurer que chaque étape asynchrone de l'itération se déroule correctement avant de continuer.

Une fois cela fait, j'ajoute l'utilisateur courant à la liste des participants et je crée les discussions pour le groupe. Une discussion est créée par groupe, en veillant à identifier l'utilisateur sujet de la discussion afin de pouvoir distinguer les participants de la discussion du sujet lui-même.

Je peux maintenant essayer ma mutation sur apollo studio.

3 L'interface côté client

Maintenant que la logique de création de groupe est en place côté API, je peux me concentrer sur le développement de l'interface client pour permettre aux utilisateurs de saisir les données nécessaires à la création d'un groupe. Cette étape est cruciale pour offrir une expérience utilisateur fluide et intuitive.

Intégration de la Requête GraphQL

Je commence par récupérer la requête de mutation que j'ai créée précédemment dans Apollo Client et l'ajoute dans mon dossier GraphQL, dans un fichier dédié nommé **`addNewGroup.gql`.** Cette organisation permet à Codegen de détecter automatiquement la requête et de générer le code nécessaire. Ensuite, je lance le script de génération avec Codegen, ce qui me permet d'obtenir un hook de mutation personnalisé pour la création de groupe.

requête gql “brute”, création d’un groupe addNewGroup.gql

frontend > src > graphql > addNewGroup.gql

```
mutation AddNewGroup($data: NewGroupInput!) {
    addNewGroup(data: $data) {
        id
        name
        event_date
        avatar {
            id
            name
        }
    }
}
```

Utilisation du Hook Personnalisé

Le hook personnalisé généré par Codegen, **`useAddNewGroup`**, fournit une fonction **`addNewGroup`** pour exécuter la mutation, ainsi que des états associés, tels que l'état d'erreur. Cela simplifie considérablement l'intégration de la mutation dans l'interface utilisateur.

Hook de création du groupe autogénéré par Codegen creating-groups.tsx

frontend > src > pages > creating-groups.tsx > CreatingGroups

```
export default function CreatingGroups() {
    const [addNewGroup, { error }] = useAddNewGroupMutation({
        onCompleted: () => {
            toast.success('Groupe créé avec succès!')
            router.push('/groupes')
        },
        onError: error => {
            toast.error(
                `Erreur lors de la création du groupe: ${error.message}`
            )
        },
    })
```

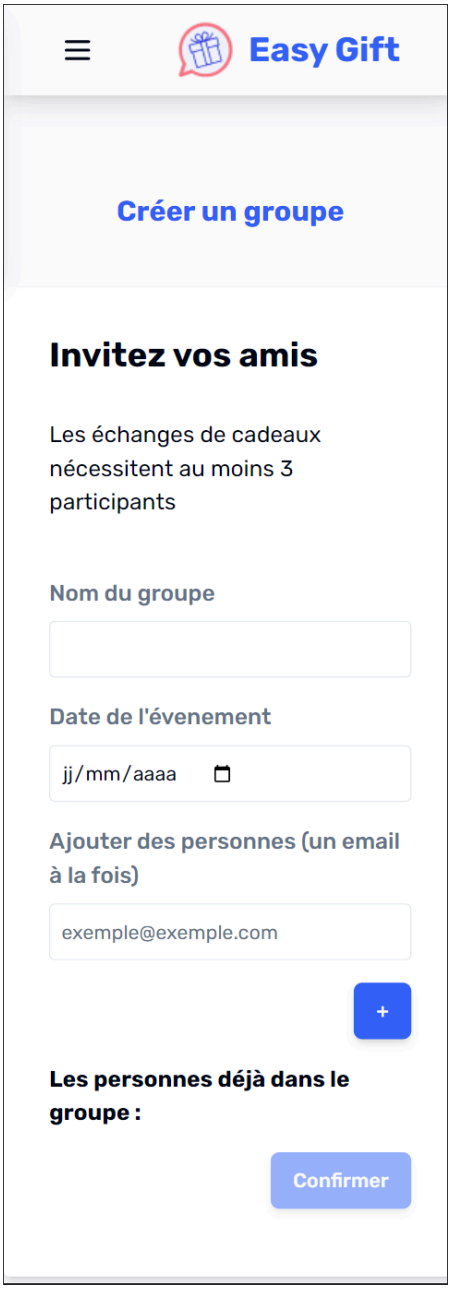
Gestion des Scénarios de Mutation

Lors de l'utilisation de ce hook dans l'application, je définis la logique à exécuter pour deux scénarios spécifiques :

- **Succès de la mutation :** En cas de succès de la mutation, j'affiche une notification de confirmation sous forme de toast pour informer l'utilisateur que le groupe a été créé avec succès. Ensuite, je redirige l'utilisateur vers la page des groupes, où il pourra voir le groupe nouvellement créé et accéder à ses détails.
- **Échec de la mutation :** Si la mutation échoue, par exemple en raison d'un problème de validation ou d'un problème de réseau, une notification d'erreur est affichée. Cela permet à l'utilisateur de comprendre ce qui s'est passé et de prendre les mesures nécessaires pour résoudre le problème.

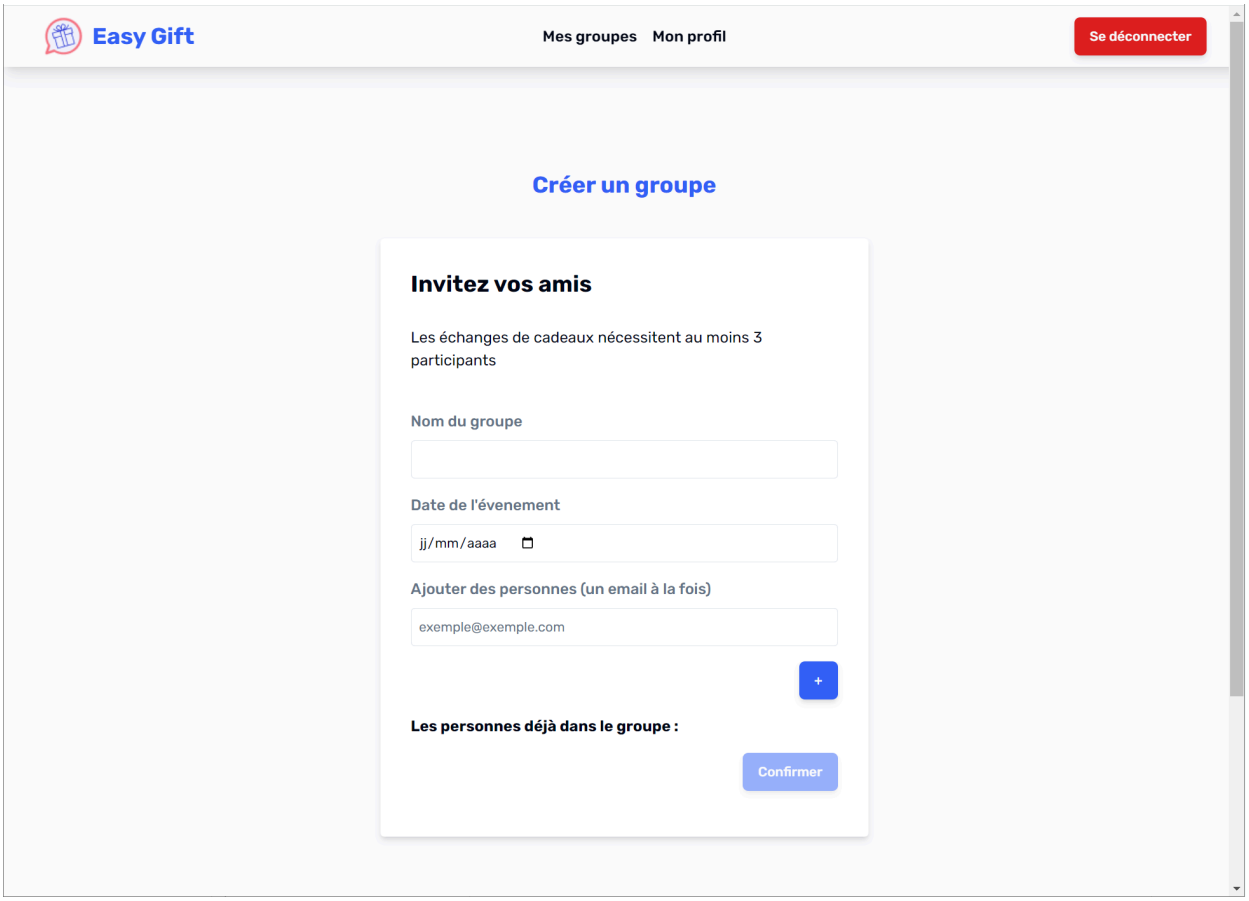
Développement de l'Interface Client

Avec le hook principal désormais en place, je peux intégrer l'écran de création de groupe dans l'application. J'adopte une approche *mobile first*, c'est-à-dire que je commence par concevoir et intégrer la version mobile de l'interface. Cette approche garantit que l'application est optimisée pour les petits écrans dès le départ, offrant ainsi une expérience utilisateur cohérente sur tous les appareils.



Adaptation pour les Écrans Plus Grands

Une fois la version mobile de l'écran de création de groupe intégrée, j'adapte l'interface aux écrans plus grands, tels que les tablettes et les ordinateurs de bureau. Cette adaptation inclut l'ajustement des éléments de l'interface pour tirer parti de l'espace supplémentaire disponible, tout en maintenant une mise en page claire et accessible.



Mon utilisateur peut désormais saisir des données dans le formulaire, et je peux récupérer ces informations via la fonction **`handleSubmit`** pour ensuite exécuter la fonction **`addNewGroup`**.

Si tout se déroule correctement, l'utilisateur est redirigé vers la page des groupes où il peut voir le groupe qu'il vient de créer.

Les tests

La mise en place de l'environnement de test est essentielle pour établir une méthodologie de CI/CD efficace. L'objectif est de réaliser différents types de tests, notamment des tests d'intégration et des tests end-to-end.

Les tests d'intégration visent à vérifier que les interactions entre différentes méthodes et fonctions du code se déroulent sans causer de perturbations ou d'erreurs. Ils sont souvent utilisés pour valider la bonne création de données en base de données. Dans notre cas, j'ai réalisé un test d'intégration sur le resolver utilisateur en utilisant la librairie Jest. En résumé, le test consiste à simuler la création d'un ou plusieurs utilisateurs avec un jeu de données spécifique et à vérifier que ces données sont correctement enregistrées en base à la suite de l'opération.

Les tests end-to-end (E2E) sont ceux qui se rapprochent le plus des situations réelles. Ils traversent toutes les couches de l'application pour tester un flux logique, comme la modification d'un jeu de données. Dans ce cas précis, j'ai testé le processus de connexion d'un utilisateur.

Le test d'intégration

Pour effectuer le test d'intégration, je me place dans le répertoire de mon backend, où j'installe et configure l'environnement de test à l'aide de la librairie Jest. Cette configuration me permet de décrire des scénarios, d'exécuter des logiques, puis de comparer les résultats obtenus avec des données attendues.

La configuration de l'environnement de test se fait principalement dans le fichier **`jest.setup.ts`**, où je procède de la manière suivante :

- Je simule un serveur GraphQL pour avoir accès à mon schéma.
- Avant toute chose, j'initialise ma base de données et j'extrais mon schéma de données.
- Je rédige une fonction pour nettoyer la base de données, afin d'éviter toute interférence pendant les tests. Cette fonction est exécutée avant chaque test.
- Enfin, après l'exécution de tous les tests, je supprime la base de données de test.

Dans cet exemple, je décris une action qui doit renvoyer une liste d'utilisateurs. Plus précisément, je vais spécifier une série d'actions simulant l'inscription d'utilisateurs sur l'application, avec l'ajout de leurs données en base de données.

En utilisant l'entité User, je crée les utilisateurs, puis je compare les résultats obtenus avec les résultats attendus.

backend > __tests__ > TS usersResolver.test.ts > ...

```ts
import { User } from '../src/entities/user'
import getUsers from './operations/getUsers'
import { execute } from '../jest.setup'
import register from './operations/register'

describe('user resolver', () => {
    it('should return the users', async () => {
        await User.create({
            pseudo: 'John',
            email: 'john@gmail.com',
            password: 'test@1234',
        }).save()
        await User.create({
            pseudo: 'Jane',
            email: 'jane@gmail.com',
            password: 'test@1234',
        }).save()
        const res = await execute(getUsers)
        expect(res).toMatchInlineSnapshot(`
            {
              "data": {
                "users": [
                  {
                    "email": "john@gmail.com",
                    "id": 1,
                    "pseudo": "John",
                  },
                  {
                    "email": "jane@gmail.com",
                    "id": 2,
                    "pseudo": "Jane",
                  },
                ],
              },
            }
        `)
    })

    it('can create a user', async () => {
        const res = await execute(register, {
            data: {
                pseudo: 'Léopold',
                email: 'leopold@gmail.com',
                password: 'test@1234',
            },
        })
        expect(res).toMatchInlineSnapshot(`
            {
              "data": {
                "register": {
                  "email": "leopold@gmail.com",
                  "pseudo": "Léopold",
                },
              },
            }
        `)
    })
})
```

Le test end to end

Il s'agit d'un test global qui évalue l'application dans son ensemble, de bout en bout. Pour ce faire, j'initialise l'environnement de test dans un dossier indépendant situé à la racine, appelé **`e2e-tests`**. Cette configuration nécessite la réplication de ma base de données existante afin de lancer une simulation. Ensuite, avec la librairie Playwright, je vais émuler un navigateur Chrome pour accéder à mon client frontend et l'utiliser comme le ferait un utilisateur.

Pour lancer correctement l'environnement de test end-to-end, je dois orchestrer le démarrage d'images de test à l'aide d'un fichier YAML, **`docker-compose.e2e-tests.yml`**, situé à la racine du projet. Ainsi, je peux exécuter les tests rédigés dans le fichier **`login.spec.ts`**, qui ont pour objectif de tester l'inscription et la connexion d'un utilisateur de bout en bout. Pour ce faire :

1. Je nomme le test et spécifie les données à renseigner.
2. Je crée un utilisateur.
3. Je me rends sur la page de connexion.
4. Je remplis le formulaire en récupérant les éléments du DOM à l'aide d'ID de test.
5. Je soumets le formulaire.
6. Je vérifie que la page sur laquelle je suis redirigé contient un bouton de déconnexion, ce qui indique que la connexion a réussi.

e2e-tests > tests > TS login.spec.ts > ...

```ts
import { test, expect } from "@playwright/test";
import { User } from "../../backend/src/entities/user";
import { clearDB } from "../../backend/src/db";
import db from "../../backend/src/db";

test.beforeAll(async () => {
  await db.initialize();
});

test.beforeEach(async () => {
  await clearDB();
});

test("has a valid user name", async ({ page }) => {
  const pseudo = "Olga";
  const emailToCheck = "olga@gmail.com";
  const passwordToCheck = "test@1234";

  await User.create({
    pseudo: pseudo,
    email: emailToCheck,
    password: passwordToCheck,
  }).save();

  await page.goto("http://localhost:3000/auth/login");
  await page.getByTestId("login-email").fill(emailToCheck);
  await page.getByTestId("login-password").fill(passwordToCheck);
  await page.getByTestId("login-button").click();
  await page.waitForTimeout(5000);
  const disconnectButton = page.getByRole("button", {
    name: "Se déconnecter",
  });
  expect(disconnectButton).toBeVisible();
});
```

Ainsi, j'ai testé toutes les couches de mon application, en passant par la base de données, l'API du backend, et l'interface utilisateur du frontend.

Le déploiement

L'objectif de la CI/CD est de créer un cycle d'intégration et de déploiement continu, permettant de déployer et d'intégrer son code de manière autonome. Il est crucial de s'assurer que le code soumis fonctionne correctement, et c'est là qu'interviennent les tests. Pour cela, nous utilisons GitHub Actions, qui permet d'exécuter des tests sur les serveurs de GitHub. Cela nous permet de vérifier que le code récemment poussé est valide. Si les tests sont réussis, le code de la pull request pourra être mergé.

Pour assurer un développement d'équipe serein et de haute qualité, nous avons mis en place ESLint et Prettier afin de contrôler et standardiser le code que nous produisons.

ESLint est un outil qui nous aide à maintenir une qualité de code cohérente au sein de l'équipe. Il analyse notre code pour détecter les erreurs potentielles, les incohérences, et les mauvaises pratiques. En utilisant ESLint, nous pouvons repérer rapidement les problèmes avant même de tester notre application, ce qui nous permet de corriger les erreurs plus tôt dans le processus de développement. En plus, il nous aide à suivre des règles de codage communes, ce qui rend notre code plus lisible et plus facile à maintenir par tous les membres de l'équipe.

Quant à lui, Prettier est un outil qui nous permet de formater automatiquement notre code de manière cohérente et uniforme, quel que soit le développeur qui travaille dessus. Il s'assure que l'apparence du code est propre et ordonnée, en respectant des règles de mise en forme définies, comme l'indentation, l'espacement, et la gestion des lignes. Grâce à Prettier, nous n'avons pas à nous soucier des détails de mise en forme, ce qui nous permet de nous concentrer sur la logique du code. De plus, cela permet d'éviter les modifications inutiles dans les commits, en supprimant les différences de formatage superflues, ce qui rend l'historique des versions plus clair et plus facile à suivre.

L’approche DevOps

Nous avons pris en charge l'installation de notre serveur VPS en sécurisant son accès. Pour cela, nous avons modifié le port par défaut et renforcé la sécurité en changeant le mot de passe. Ensuite, nous avons mis en place une liste de bannissement automatique pour les adresses IP qui tentent de se connecter et échouent trop souvent, afin de prévenir les attaques par force brute.

En plus de ces mesures de sécurité, nous avons installé une librairie de routage au sein de notre serveur. Étant donné qu'il n'y a qu'un seul point d'entrée, nous utilisons Nginx et Caddy

pour rediriger les requêtes vers différents ports internes, ce qui nous permet de gérer et de pointer vers plusieurs services déployés sur le même serveur.

Ensuite, nous avons abordé le déploiement de notre application sur ce serveur. Pour ce faire, nous avons mis en place un webhook connecté à GitHub, qui attend d'être activé lors d'actions spécifiques comme un push sur les branches **`main`** ou **`staging`**. Une fois activé, ce webhook déclenche un script de déploiement sur notre serveur, qui récupère les images de nos conteneurs depuis Docker Hub et lance l'application, la distribuant sur les ports définis lors de l'installation du serveur.

Ainsi, nous sommes désormais capables de déposer du code sur notre branche de production GitHub, en nous assurant que le code est sain et fonctionnel. Tout cela automatise le déploiement, rendant notre application disponible sur le web.