Création d'une API avec le langage de requête GraphQL

Bénédicta MANGBA & François TOYI

19 mars 2025

Table des matières

1 Introduction à GraphQL

1.1 Qu'est-ce que GraphQL ?

GraphQL est un langage de requêtes pour les APIs développé par Facebook en 2015. Contrairement aux APIs REST traditionnelles, qui fonctionnent par plusieurs points de terminaison (endpoints) pour des ressources spécifiques, GraphQL utilise un seul endpoint et permet aux clients de définir précisément les données qu'ils souhaitent recevoir. Cela rend GraphQL particulièrement flexible et optimisé pour réduire le sur- ou sous-chargement de données, ce qui est très utile dans les applications modernes où le front-end peut nécessiter un contrôle plus fin sur les données échangées.

1.2 Concepts de base de GraphQL

- Schéma et types : Le cœur de GraphQL repose sur un schéma fort typé. Un schéma définit les types d'objets que l'API expose, leurs champs et leurs relations.
- Requêtes (queries) : Les requêtes permettent aux clients de demander des données spécifiques en fonction de leurs besoins.
- Mutations : Les mutations permettent d'envoyer des modifications au serveur, comme la création, la mise à jour, ou la suppression d'enregistrements.
- Résolveurs : Un résolveur est une fonction qui gère la logique derrière chaque champ de la requête.
- Souscriptions : Les souscriptions permettent de recevoir des mises à jour en temps réel, utiles dans les applications en temps réel.

1.3 Les types en GraphQL

- Types scalaires : Ce sont les types de base utilisés pour représenter des valeurs simples. Les principaux types scalaires sont :
  - `Int` : Nombre entier.
  - `Float` : Nombre décimal.
  - `String` : Chaîne de texte.
  - `Boolean` : Valeur booléenne (`true` ou `false`).
  - `ID` : Identifiant unique.
- Types objets : Ils définissent des structures de données regroupant plusieurs champs.
- Types racines : GraphQL repose sur trois types principaux permettant l'interaction avec l'API :
  - `Query` : Permet de récupérer des données.
  - `Mutation` : Permet de modifier des données (ajout, suppression, mise à jour).
  - `Subscription` : Permet de recevoir des mises à jour en temps réel.
- Types avancés :

— `[Type]` : Représente une liste d'éléments d'un type donné.
— `Type!` : Indique qu'une valeur est obligatoire (non nulle).
— `enum` : Définit un ensemble de valeurs prédéfinies.
— `interface` : Permet de définir des structures communes à plusieurs types.

Les types GraphQL avec des simpleexemples

1. Types scalaires ✱

- `Int` : Nombre entier — Ex : `42`
- `Float` : Nombre décimal — Ex : `3.14`
- `String` : Texte — Ex : `"Bonjour ! ☺"`
- `Boolean` : Vrai/Faux — Ex : `true`
- `ID` : Identifiant unique — Ex : `"abc123"`

2. Types objets (Groupes de données)

```
type Livre {
  titre: String!
  pages: Int!
  disponible: Boolean!
}
```

3. Types racines (Points d'entrée)

```
# Lire des donnees
type Query {
  livres: [Livre!]!
}

# Modifier des donnees
type Mutation {
  ajouterLivre(titre: String!): Livre!
}

# Recevoir en temps reel
type Subscription {
  nouveauLivre: Livre!
}
```

4. Types avancés

- `[Type]` : Liste — Ex : `[String]`
- `Type!` : Obligatoire — Ex : `Int!`
- `enum` : Choix fixés — Ex : `enum Couleur  ROUGE VERT BLEU`
- `interface` : Structure commune — Ex : `interface Produit`
  `prix: Float!`

Exemple complet

```
type Query {
  produits: [Produit!]!
}

type Produit {
  id: ID!
  nom: String!
  prix: Float!
  enStock: Boolean!
}
```

1.4 Avantages de GraphQL

— Précision des requêtes : Les clients peuvent demander uniquement les champs dont ils ont besoin.
— Un endpoint unique : Simplifie la gestion des APIs, puisque toutes les opérations passent par un seul point de terminaison.
— Performance optimisée : Moins de surcharges de données et plus de flexibilité dans le traitement des informations.
— Documentation intégrée : GraphQL est auto-documenté, ce qui facilite la découverte des possibilités de l'API.

1.5 Limites de GraphQL

— Complexité : Il peut être complexe à configurer et à apprendre, notamment pour les grands schémas.
— Cache difficile à gérer : Contrairement aux APIs REST où le caching est plus intuitif, le cache dans GraphQL demande souvent une gestion sur mesure.

2 Prérequis pour utiliser GraphQL sur Ubuntu

2.1 Installation des outils nécessaires

Avant de commencer à utiliser GraphQL, vous devez installer certains outils sur votre machine Ubuntu. Voici les étapes à suivre :

1. Node.js et npm : GraphQL est souvent utilisé avec Node.js. Pour installer Node.js et npm (Node Package Manager), exécutez les commandes suivantes :

```
    sudo apt update
    sudo apt install nodejs npm

```

2. Éditeur de texte : Vous aurez besoin d'un éditeur de texte pour écrire votre code. Vous pouvez utiliser Visual Studio Code (VS Code) qui est très populaire parmi les développeurs. Pour installer VS Code :

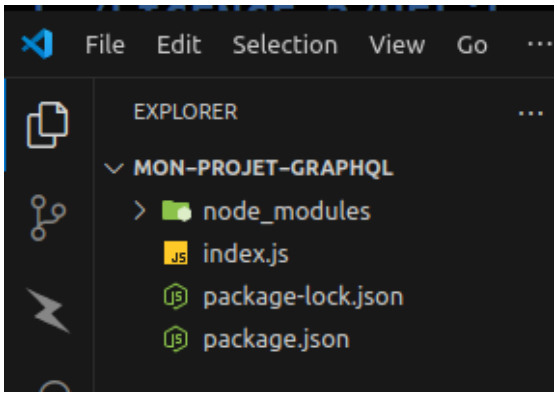
```
    sudo snap install --classic code

```

2.2 Passons à la pratique ! 🚀

Maintenant que tout est installé, nous allons créer un projet GraphQL simple. Suivez les étapes ci-dessous pour afficher "Hello World" avec GraphQL. Création d'un projet GraphQL

3 Configuration de l'environnement de travail

Maintenant que tout est installé, nous allons créer un projet GraphQL simple. Suivez les étapes ci-dessous pour afficher "Hello World" avec GraphQL. Voici à quoi ressemblerait notre structure :

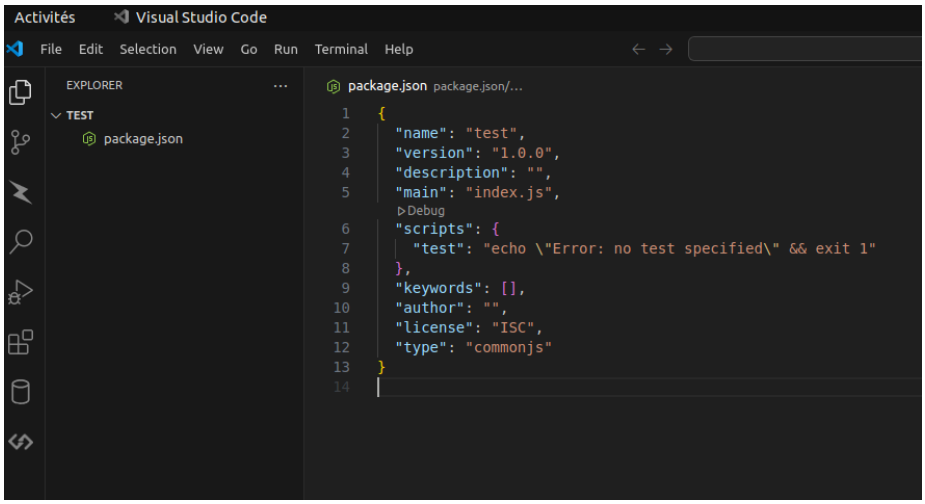


4 Initialisation du projet

4.1 Commandes de base et explication

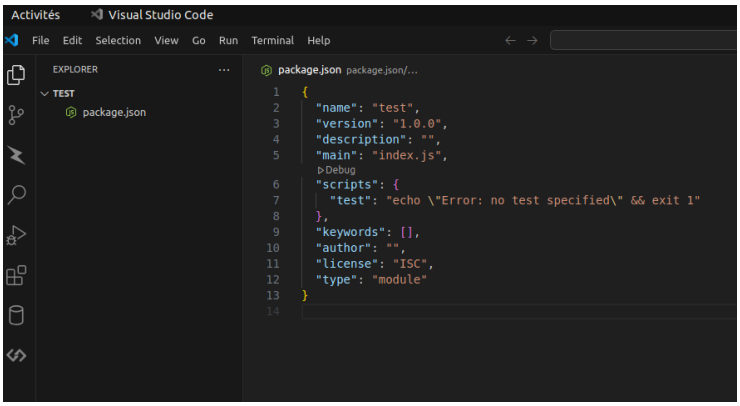
```
mkdir test                  // Creation du dossier du projet
cd test                     // Acces au dossier du projet
npm init -y                 // Initialisation dun projet Node.js
```

Après cette commande, vous aurez un fichier `package.json` qui ressemblera à ceci :



5 Modification du fichier `package.json`

Maintenant, vous devez changer le type de module de `commonjs` en `module`, car c'est ce que nous allons utiliser dans notre projet. Modifiez la valeur de `type` de `commonjs` en `module`.



6 Installation des dépendances

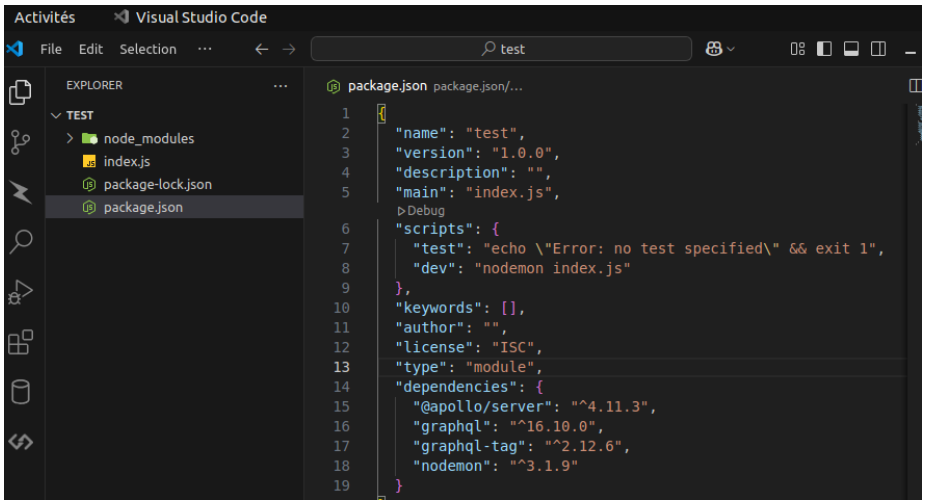
6.1 Commandes de base et explication

Acceder à votre terminal et executez les commandes suivantes

```
npm install @apollo/server graphql  // Installation des
    dependances
npm install graphql-tag  // Installation de la bibliotheque
    graphql-tag
npm install nodemon // pour permettre le redemarrage a chaud
     du serveur
```

Dans le fichier `package.json`, au niveau de la section script, mettez virgule(,) et ajoutez la ligne suivante :

```
"dev": "nodemon index.js"
```



7 Ajout du code dans `index.js`

```
import { ApolloServer } from '@apollo/server';
import { gql } from 'graphql-tag';
import { startStandaloneServer } from "@apollo/server/standalone";

// Definition du schema GraphQL
const typeDefs = gql`
  type Query {
    hello: String
  }
`;

// Resolveur pour le schema
const resolvers = {
  Query: {
    hello: () => 'Hello, world!',
  },
};

// Creer une instance d'ApolloServer
const server = new ApolloServer({
  typeDefs,
  resolvers,
});

// Lancer le serveur en utilisant startStandaloneServer
startStandaloneServer(server, {
  listen: { port: 4000 }
}).then(({ url }) => {
  console.log(`Server ready at ${url}`);
});
```

Voici une explication détaillée du code index.js :

Explication du code Apollo Server

Structure du code en 5 parties :

1. **Importations des dépendances** :
   - `ApolloServer` : Framework GraphQL
   - `gql` : Parseur de schéma GraphQL
   - `startStandaloneServer` : Serveur HTTP intégré
2. **Définition du schéma GraphQL** (typeDefs) :

```
    type Query {
        hello: String  #Expose une requete 'hello'
    }

```

3. **Résolveurs (resolvers)** :
   - Implémente la logique de la requête
   - Renvoie toujours "Hello, world !"
4. **Création du serveur** :
   - Combine schéma et résolveurs
   - Configuration minimale
5. **Lancement du serveur** :
   - Port 4000 par défaut
   - Message de confirmation au démarrage

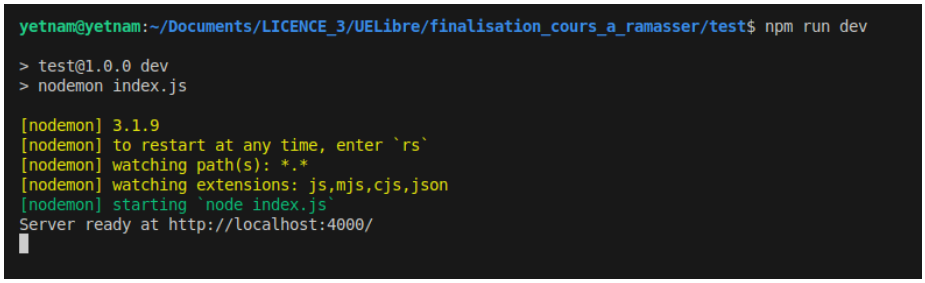
Fonctionnement global :
Le serveur répondra à une requête GraphQL `{ hello }` avec la chaîne "Hello, world !".

8 Exécution du serveur

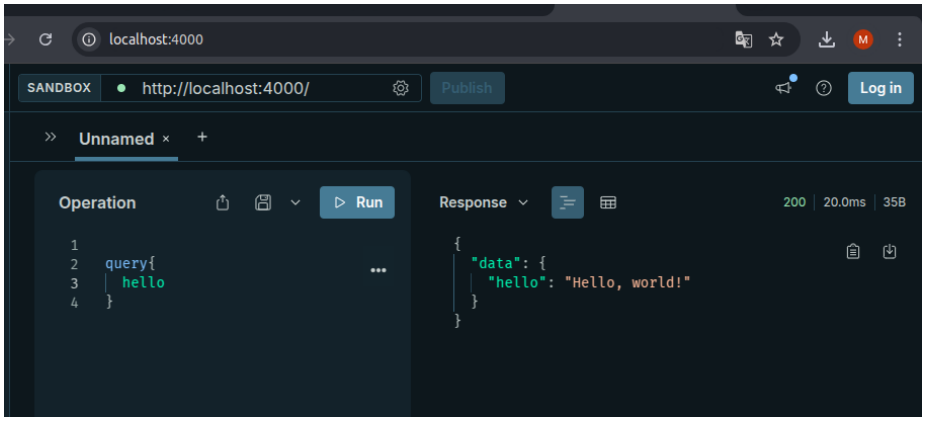
Dans votre terminal, exécutez la commande suivante pour démarrer le serveur :

```
npm run dev
```

Vous devriez voir le message suivant dans votre terminal :



Si vous obtenez ce résultat, cela signifie que vous avez bien effectué les tests. Ensuite, appuyez sur `Ctrl` puis cliquez sur le endpoint pour afficher votre requête. Vous n'avez qu'à taper dans la partie opération exactement ce qui est affiché sur l'image suivante pour obtenir le résultat :



9 Test pratique

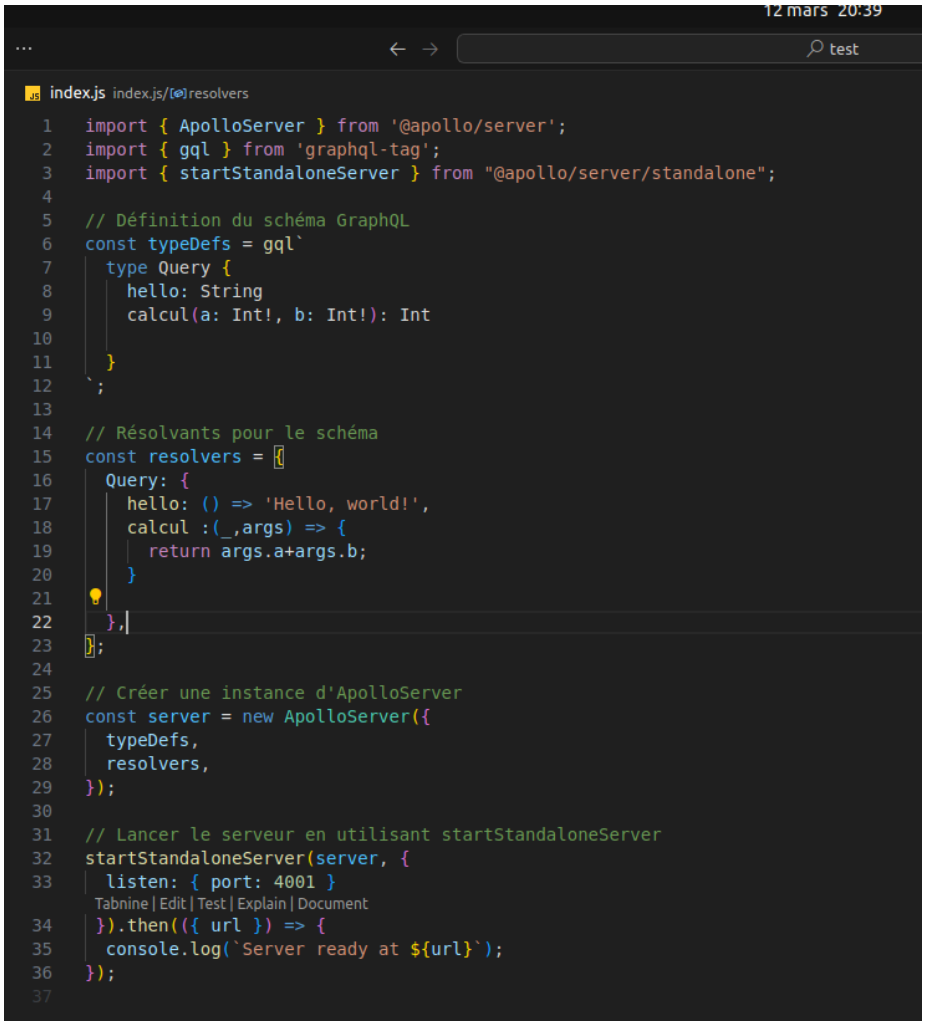
9.1 Ajout de nouveaux types et requêtes 🪄🧮

Défi mathématique !

"Et si on équipait notre API de super circuits intégrés ?
Transformons ce serveur en calculatrice GraphQL ! 🚀

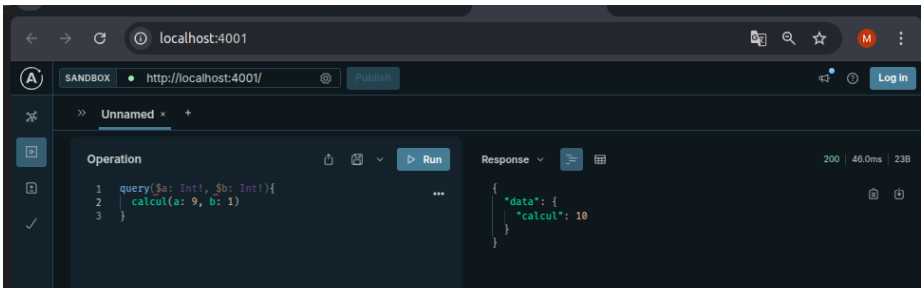
Mission :

- </> Créer une requête `calcul` pour additionner deux nombres
- Déclarer les types GraphQL appropriés
- Résolveur effectuant `a + b`



Capture du code modifié

- Type Query avec `calcul(a: Int!, b: Int!): Int!`
- Résolveur qui manipule `a` et `b`
- Test avec `{ calcul(a: 9, b: 1) }`



👍 *Résultat attendu : 10(pas 57!)* ☺

💬 **Astuce :**

"Si votre code retourne 57...
C'est un concaténateur masqué ! 🔍

Reprennons notre serieux et comprenons le code suivant, cela nous est très important :

</> Explication du résolveur GraphQL

Structure du code

1. Schéma GraphQL :

```
type Query {
  "Additionne deux entiers (a + b)"
  calcul(a: Int!, b: Int): Int
}
```

- `Int!` : Paramètre obligatoire
- `Int` : Peut retourner `null`

2. Résolveur JavaScript :

```
const resolvers = {
// 1. Requette
  Query: {
    calcul: (_, args) => {

      // 2. Calcul
      const result = args.a + args.b;

      // 3. Retour
      return result;
    }
  }
};
```

Explications clés

_	Placeholder pour l'objet parent
args	`{ a: 9, b: 1 }` (arguments)
Retour	Doit matcher le type GraphQL

Flux d'exécution

1. Requête client : `query { calcul(a:9, b:1) }`
2. Vérification des types par GraphQL
3. Exécution du résolveur si valide
4. Retour : `{ "data": { "calcul": 10 } }`

Alors maintenant, nous allons tester les types objets.

Pour commencer, les types objets sont des types personnalisés.

Voici ce que nous allons ajouter dans le fichier `index` :

Nous allons définir une tâche sur laquelle nous allons effectuer un petit CRUD. Voici les étapes à suivre :

1. Définition du type `Tache`

Pour définir le type, voici ce que nous allons ajouter dans le `typedef` :

```
type Tache {
  id: ID!
  titre: String!
  terminer: Boolean!
}
```

2. Définition de la requête

Ensuite, nous allons définir la requête pour obtenir la liste des tâches :

```
taches: [Tache]
```

3. Définition du resolveur

Enfin, nous allons définir le resolveur qui va gérer la requête des tâches :

```
taches: () => tasks
```

4. Exemple de données

Voici un exemple de données dans une liste pour effectuer les tests :

```
let tasks = [
  { id: 1, titre: "Apprendre GraphQL", terminer: false },
  { id: 2, titre: "Faire les courses", terminer: true }
];
```

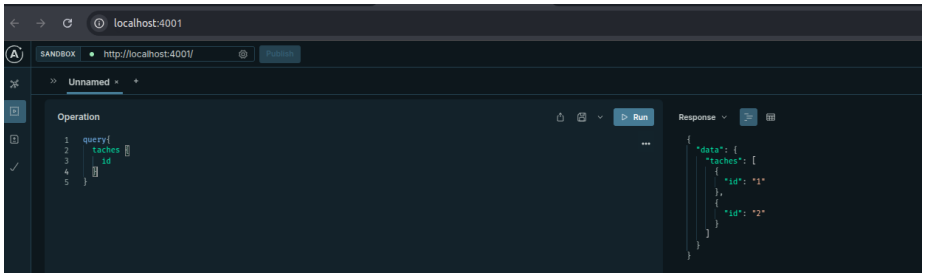
Ces étapes vous permettront de définir et d'utiliser des types objets dans GraphQL pour effectuer un CRUD basique.

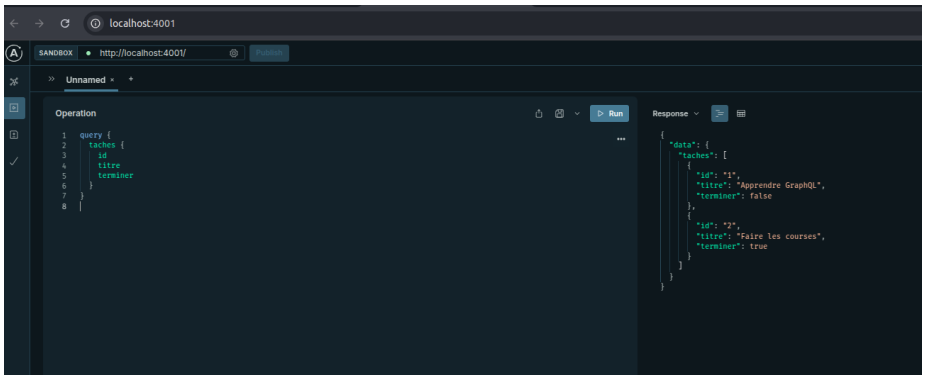
Votre fichier **index.js** devrait ressembler à ceci

```
import { ApolloServer } from '@apollo/server';
import { gql } from 'graphql-tag';
import { startStandaloneServer } from "@apollo/server/standalone";

// D finition du sch ma GraphQL
const typeDefs = gql`
  type Tache {
    id: ID!
    titre: String!
    terminer: Boolean!
  }

  type Query {
    hello: String
    calcul(a: Int!, b: Int!): Int
    taches: [Tache]
```

```
  }
`;

let tasks = [
  { id: 1, titre: "Apprendre GraphQL", terminer: false },
  { id: 2, titre: "Faire les courses", terminer: true }
];

// Resolvants pour le sch ma
const resolvers = {
  Query: {
    hello: () => 'Hello, world!',
    calcul: (_, args) => args.a + args.b,
    taches: () => tasks
  }
};

// Creer une instance d'ApolloServer
const server = new ApolloServer({
  typeDefs,
  resolvers,
});

// Lancer le serveur en utilisant startStandaloneServer
startStandaloneServer(server, {
  listen: { port: 4001 }
}).then(({ url }) => {
  console.log(`Server ready at ${url}`);
});
```

Maintenant pour tester retourner sur le playGround et faite :





Ce qui rend GraphQL si puissant ?
(On vous l'avait déjà expliqué, mais un rappel est toujours utile)

Avec GraphQL, **personnalisez à la volée** les données affichées !
Plus de surcharge, place à la précision
Vous décidez des champs, rien que ceux dont vous avez besoin
Exactement ce que vous voulez, rien de plus

(Une flexibilité que REST ne peut égaler !)

Maintenant , faisons la mise à jour , la creation et la supression, nous pourrons maintenant manipuler notre type **Mutation**

Pour se faire, nous allons utiliser le type **Tache** déjà défini , puis d'abord declarer la mutation apres cela nou allons de definir le corps de la mutation dans le resolvers . Voici ce qu'il faut faire :

```

type Tache {
  id: ID!
  titre: String!
  terminer: Boolean!
}
```

Declarer la mutation

```

type Mutation{
  ajouterTache(titre: String!, terminer: Boolean!): Tache

}
```

Apres definir le corps de la mutation dans le resolveur

```
 Mutation:{
    ajouterTache: (_, {titre, terminer}) => {
    const nouvelleTache = {
      id: tache.length + 1,
      titre,
      terminer
    };
    tache.push(nouvelleTache);
    return nouvelleTache;
    }
  }
```

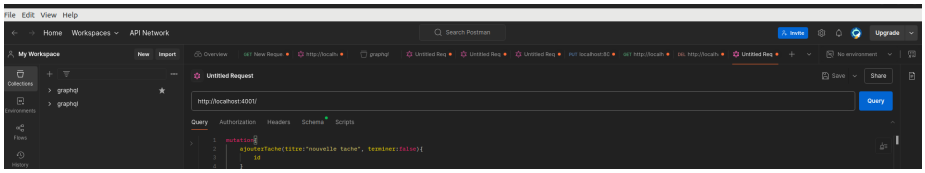
Si vous faite bien ce qui a été dit ; Voici la manière dont votre fichier index devrait etre

```
import { ApolloServer } from '@apollo/server';
import { gql } from 'graphql-tag';
import { startStandaloneServer } from "@apollo/server/
    standalone";

// D finition du sch ma GraphQL
const typeDefs = gql`
  type Tache {
    id: ID!
    titre: String!
    terminer: Boolean!
  }
  type Query {
    hello: String
    calcul(a: Int!, b: Int!): Int
    taches: [Tache]
  }
  type Mutation{
    ajouterTache(titre: String!, terminer: Boolean!): Tache
  }
`;
let taches = [
  { id: 1, titre: "Apprendre GraphQL", terminer: false },
  { id: 2, titre: "Faire les courses", terminer: true }
];
// Resolvants pour le schema
const resolvers = {
  Query: {
    hello: () => 'Hello, world!',
    calcul: (_, args) => args.a + args.b,
  taches :()=> taches
  },
  Mutation:{
    ajouterTache: (_, {titre, terminer}) => {
```

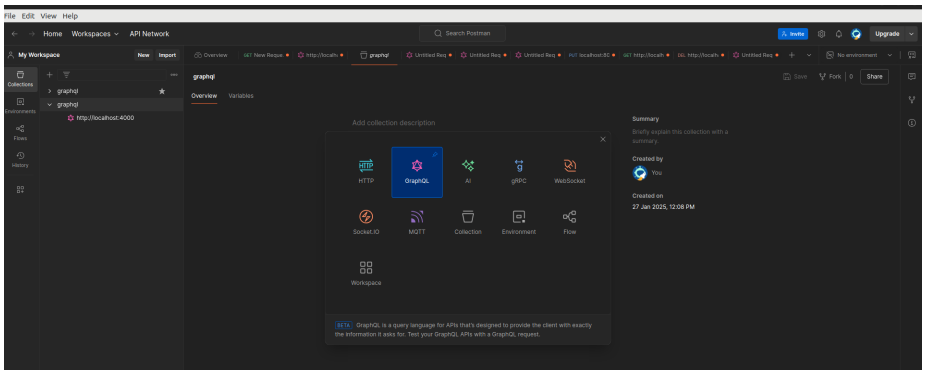
```
    const nouvelleTache = {
      id: taches.length + 1,
      titre,
      terminer
    };
    taches.push(nouvelleTache);
    return nouvelleTache;
    }
  }
};
// Creer une instance d'ApolloServer
const server = new ApolloServer({
  typeDefs,
  resolvers,
});
// Lancer le serveur en utilisant startStandaloneServer
startStandaloneServer(server, {
  listen: { port: 4001 }
}).then(({ url }) => {
  console.log(`Server ready at ${url}`);
});
```

Maintenant que nous avons terminé la définition de notre mutation, pourquoi ne pas tester tout ça ? Il est important de noter que GraphQL ne s'exécute pas uniquement dans le Playground, mais aussi dans des clients HTTP comme Postman. Voici comment vous pouvez tester la création d'une tâche dans Postman.

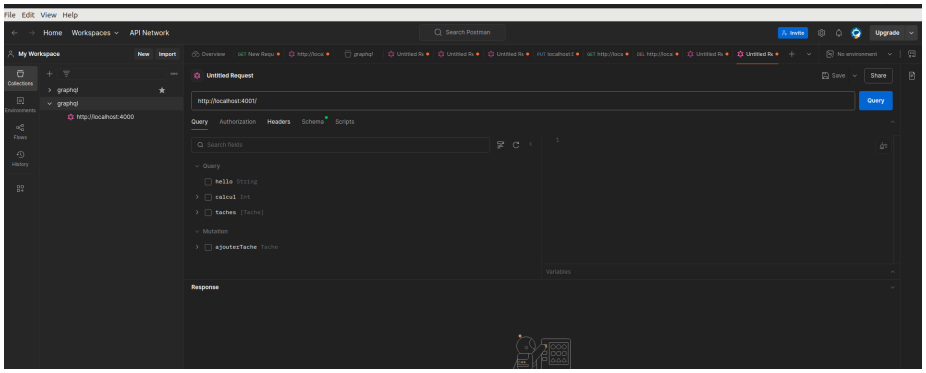
D'abord, téléchargez Postman en suivant ce lien : Télécharger Postman. Une fois l'installation terminée, si vous ne l'avez pas encore configuré, voici comment procéder :



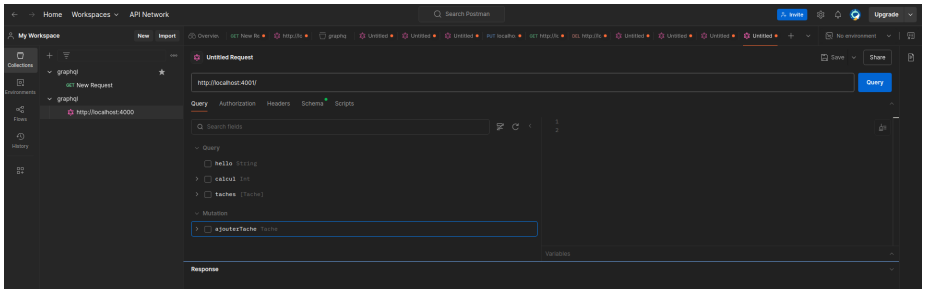
Ensuite, en haut à gauche, là où il est écrit New et Import, cliquez sur New pour ouvrir un nouvel onglet. Sélectionnez ensuite GraphQL. Voici l'illustration :



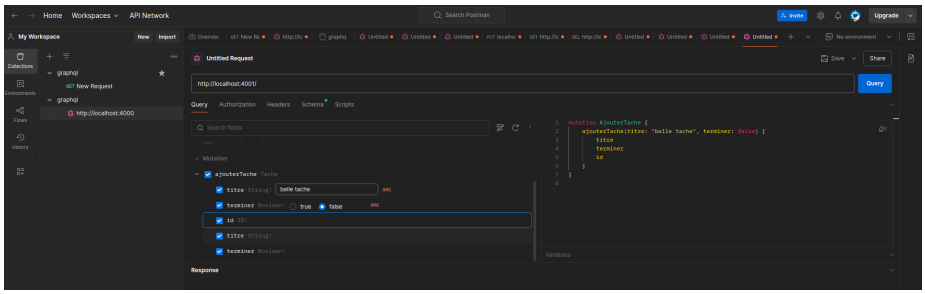
Une fois que vous avez cliqué dessus, entrez l'URL de votre endpoint unique en haut, comme montré ci-dessous. N'oubliez pas de remplacer l'exemple par le vôtre.



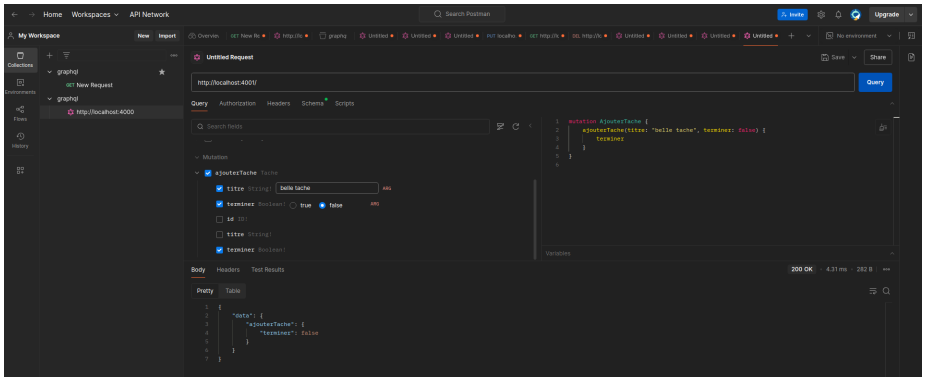
Maintenant que tout est prêt, voici comment tester la création d'une tâche dans Postman. Une fois que vous accédez à Postman, vous verrez la liste de vos requêtes et mutations, comme montré ci-dessous :



Cliquez sur le nom de votre mutation. Vous aurez alors des champs pour saisir les données nécessaires. Voici l'avantage d'utiliser Postman par rapport au Playground.



Par défaut, tous les champs sont sélectionnés, ce qui signifie que tous seront retournés après l'envoi des données. Ne vous inquiétez pas, lors de la définition de la mutation en GraphQL, nous avons précisé le type de retour. Vous pouvez donc choisir les champs que vous souhaitez recevoir. En cliquant sur Query, voici le résultat que vous obtiendrez :



À vous de manipulez pour plus de compréhension.

10 Fin de la section de test

Félicitations ! 🏆Vous avez réussi à créer votre première API GraphQL et à effectuer des tests pratiques. Dans la prochaine section, nous explorerons des concepts plus avancés, tels que les souscriptions, l'intégration avec une base de données, et bien d'autres sujets passionnants ☺.

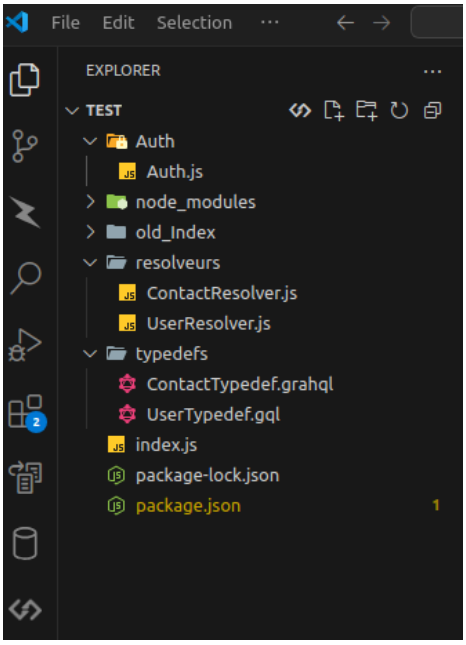
11 Projet réaliste : Gestion des contacts avec Prisma et Flutter

11.1 Concept

Maintenant que vous avez une compréhension de base de GraphQL, nous allons passer à un projet plus réaliste. Nous allons créer une API GraphQL pour gérer une liste de contacts, en utilisant Prisma comme ORM pour interagir avec une base de données, JWT pour la gestion de l'authentification. Plus tard, nous consommerons cette API.

11.2 Restructuration du projet

Voici à quoi doit ressembler la structure du projet :
Reproduisez minutieusement l'architecture suivante pour structurer proprement notre projet.



Dossiers
Les dossiers créés seront :

- **resolvers**
- **typedefs**
- **auth**
- **old_index**

Creer les fichiers se trouvant sur la capture respectivement dans chaque dossier.

Sauvegarde de l'ancien index
Le dossier **old_index** contiendra une copie de notre fichier `index.js`. Ce dossier servira à conserver les tests effectués avant la refonte du projet.

Nettoyage du fichier principal
Le fichier **index.js**, situé à la racine du projet, doit être vidé pour repartir sur une base propre et bien structurée.

Maintenant tout est en ordre nous allons installer **prisma**, une ORM qui

va permettre l'interaction avec la base de donees .mais nous repelloms que vous pouvez utiliser autre ORM de votre choix telques **Postgraphile Hasura**

Passons donc à la configuration de prisma.

12 Configuration de Prisma en ligne de commande

— **Installation de Prisma et de son client**

```
    npm install prisma --save-dev
    npm install @prisma/client

```

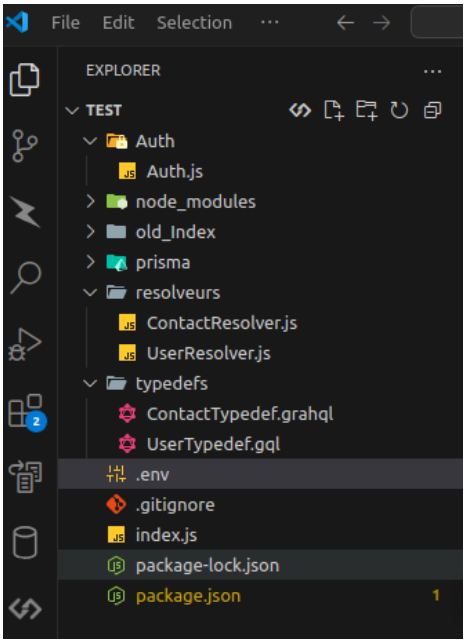
— **Initialisation de Prisma**

```
    npx prisma init

```

Si vous exécutez la commande **npx prisma init**, un dossier **prisma** apparaîtra, contenant notamment un fichier **.env**. Ce fichier de configuration vous permettra de définir les paramètres de votre base de données, ainsi que différentes variables d'environnement.

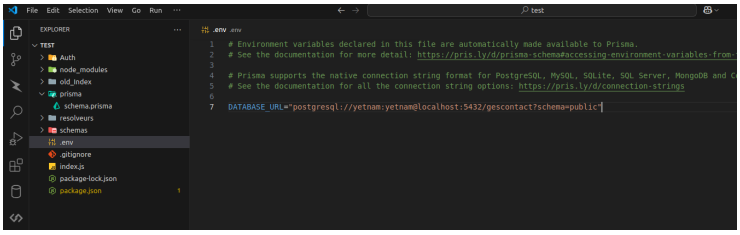
En accédant au répertoire **prisma** , vous y trouverez un fichier**schema.prisma** . C'est dans ce fichier que nous allons définir le schéma de la base de données.

Voici à quoi devrait ressembler notre aborescence à présent.

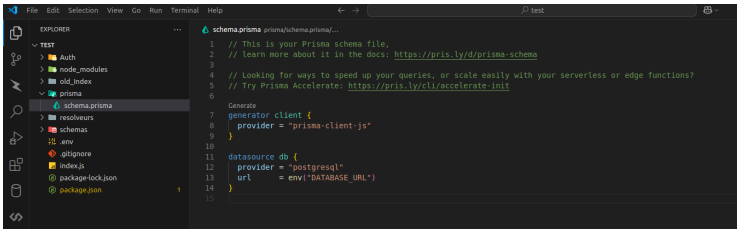


Nous allons accéder au fichier **.env** afin d'effectuer les configurations nécessaires. À la base, voici à quoi ressemble le contenu du fichier **.env**

Vous devez remplacer ces données par les informations correctes, à savoir le nom d'utilisateur, le mot de passe et le nom de la base de données. Je rappelle que dans notre cas, nous utilisons `PostgreSQL`, mais vous pouvez opter pour **SQLite**, **MySQL**, **SQL Server**, **MongoDB**, **CockroachDB**, etc., selon votre choix. Voici comment nous avons effectué notre configuration.



Accédons au fichier **schema.prisma** et définissons notre schéma, à savoir **User** et **Contact**. Mais avant cela, voici l'architecture de base de Prisma.



Voici la manière dont nous avons défini la structure de notre table.

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

// Looking for ways to speed up your queries, or scale easily with
    your serverless or edge functions?
// Try Prisma Accelerate: https://pris.ly/cli/accelerate-init

generator client {
  provider = "prisma-client-js"
}

```

```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id        Int       @id @default(autoincrement())
  name      String
  email     String    @unique
  password  String
  contacts  Contact[]
  createdAt DateTime  @default(now())
  updatedAt DateTime  @updatedAt
}

model Contact {
  id         Int       @id @default(autoincrement())
  firstName  String
  lastName   String
  phone      String
  email      String    @unique
  address    String?
  userId     Int
  user       User      @relation(fields: [userId], references: [id
    ])
  createdAt  DateTime  @default(now())
  updatedAt  DateTime  @updatedAt
}
```

Ensuite, exécutez la commande suivante pour créer la base de données et générer le client Prisma :

Création + application de migration

```
    npx prisma migrate dev --name init

```

Génération de Prisma Client

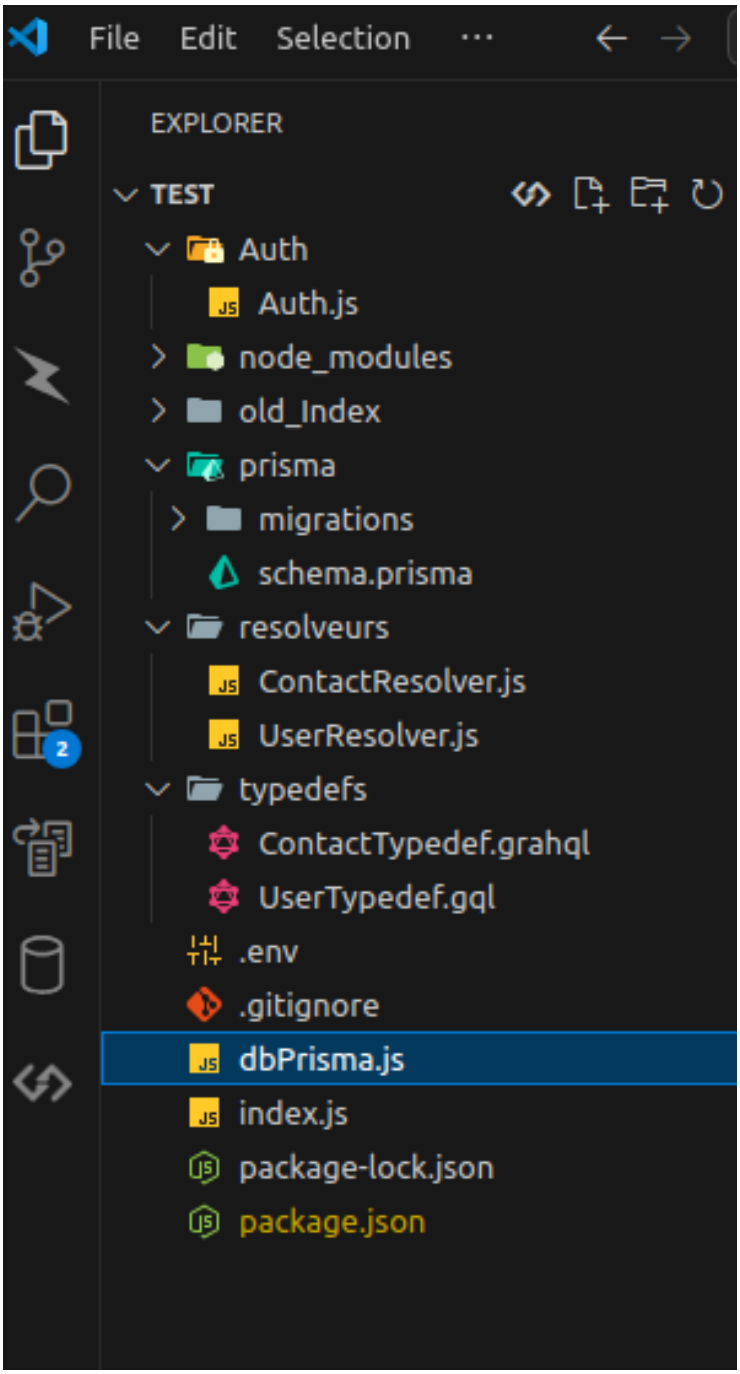
```
    npx prisma generate

```

Apres ce la tapez cette commande pour explorer la base de donnees et ajouter les donner directement dans l'interface qu'offre prisma.

Exploration de la base de données

```
    npx prisma studio

```

Ou vous pouvez egalemnt créer un fichier **dbPrisma.js** a la racine et met ce script à l'interieur pour nous generer quelques seeders pours nos differents tests.




```
1 import { PrismaClient } from '@prisma/client';
2
```

```
const prisma = new PrismaClient();
async function main() {
  const user = await prisma.user.create({
    data: {
      name: 'John Doe',
      email: 'johndoe@example.com',
      password: 'securepassword123',
      contacts: {
        create: [
          {
            firstName: 'Alice',
            lastName: 'Smith',
            phone: '123-456-7890',
            email: 'alice.smith@example.com',
            address: '123 Main St, Springfield',
          },
          {
            firstName: 'Bob',
            lastName: 'Johnson',
            phone: '987-654-3210',
            email: 'bob.johnson@example.com',
            address: '456 Elm St, Springfield',
          }
        ]
      }
    }
  });

  console.log('Utilisateur cree :', user);
}

main()
  .catch(e => {
    throw e;
  })
  .finally(async () => {
    await prisma.$disconnect();
  });
```

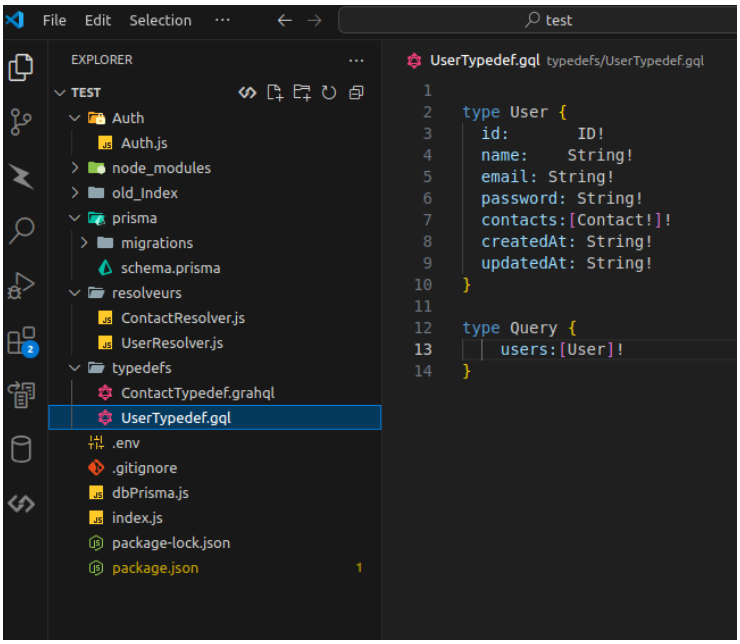
Apres ce la tapez cette commande pour executer les données de test

Insertion des données

```
node dbPrisma.js

```

Passons maintenant aux choses sérieuses. Comme cela a été mentionné à plusieurs reprises, en GraphQL, il est primordial de commencer par déclarer les types avant de se lancer dans la logique de l'API. Cette étape fondamentale permet de définir clairement la structure des données et de garantir que toutes les requêtes et mutations respectent un schéma précis. Une fois les types définis, nous pourrons ensuite aborder les fonctionnalités de l'API et la logique des résolveurs qui manipulent ces types.
Definissons donc notre schema.
Allez donc dans le fichier **schema/UserTypedef.graphql**

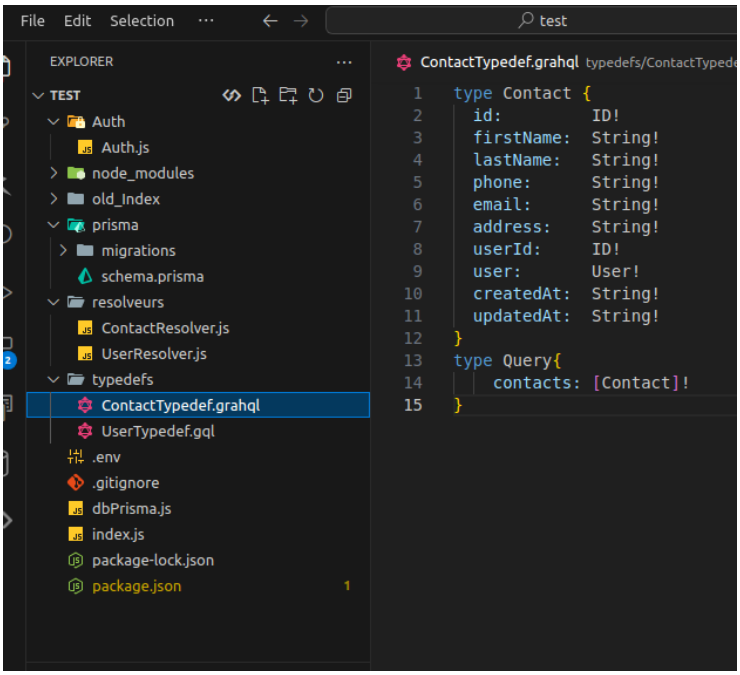
```
type User {
  id:        ID!
  name:     String!
  email: String!
  password: String!
  contacts:[Contact!]!
  createdAt: String!
  updatedAt: String!
}

type Query {
    users:[User]!
}  updatedAt: String!
}

```



et dans le ficher **ContactTypedef.gql**

```
type Contact {
  id:          ID!
  firstName:   String!
  lastName:    String!
  phone:       String!
  email:       String!
  address:     String!
  userId:      ID!
  user:        User!
  createdAt:   String!
  updatedAt:   String!
}
```

```
type Query{
    contacts: [Contact]!
}
}

```



Maintenant que nous fini avec les **typesdef** , allons dans le resolvers. Il faut vraiment noter que graphQl est vraiment exigeant sur les types sans c'est type , il nous serait impossible de demarrer notre serveur , donc il faut declarer les types meme si nous n'allons pas encore dfinir le corprs dans le resolvers.
Accedons à notre fichier, **resolvers/UserResolver.js**, ceci fait nous allons juste declarer le resolveur qui affiche la liste de tous les tutilisateurs. Voici le contenu du fichier **UserResolver.js**

```
import { PrismaClient } from "@prisma/client";
const prisma = new PrismaClient();
const userResolvers={
    Query: {
    users: async () =>{},
    },
};
export default userResolvers;

```

Voici pour contact egalement.

```
import { PrismaClient } from "@prisma/client";
const prisma = new PrismaClient();
```

```
const userResolvers={
    Query: {
    users: async () =>{},
    },
};
export default userResolvers;

```

Nous espérons que vous vous souvenez de la petite histoire de **gql** et **graphql**, un langage de requête révolutionnaire pour les API! ☺
Maintenant que nous avons fini avec nos schemas.

```
import { ApolloServer } from "@apollo/server"
import { startStandaloneServer } from "@apollo/server/standalone"
import gql from "graphql-tag"
import fs from 'fs'
import userResolvers from './resolveurs/UserResolver.js'
import contactResolvers from './resolveurs/ContactResolver.js'

const  typeDefs = gql(`
  ${fs.readFileSync('typedefs/UserTypedef.gql','utf8')}
  ${fs.readFileSync('typedefs/ContactTypedef.graphql','utf8')}
  `)
  const resolvers = {
    Query:{
      ...userResolvers.Query,
      ...contactResolvers.Query
    },
  }
  const server = new ApolloServer(
    { typeDefs, resolvers }
  )
startStandaloneServer(server,{
  listen:4000, path:'/graphql'}).then(({url})=>{
    console.log(`serveur demare sur ${url}`);

  })


```

</> Explication du contenu de **index.js**

Structure du code

1. Importation :

```
1 import { ApolloServer } from "@apollo/server"
2 import { startStandaloneServer } from "@apollo/server/standalone"
3 import gql from "graphql-tag"
4 import fs from 'fs'
5 import userResolvers from './resolveurs/UserResolver.js'
6 import contactResolvers from './resolveurs/ContactResolver.js'
```

Ici nous faisons les importations ,

ligne 1 Cette ligne importe le ApolloServer à partir du package @apollo/server. ApolloServer est une bibliothèque qui permet de configurer et de lancer un serveur GraphQL. Il est utilisé pour définir le schéma, les résolveurs, et exécuter les requêtes des clients.

ligne 2 Cette ligne importe la fonction startStandaloneServer à partir du module @apollo/server/standalone. .

ligne 3 Cette ligne importe gql de la bibliothèque graphql-tag. graphql-tag est un utilitaire qui permet d'annoter les requêtes GraphQL en JavaScript.

ligne 4
Cette ligne importe le module fs de Node.js. fs est un module natif de Node.js qui permet de manipuler le système de fichiers (par exemple, lire, écrire, supprimer des fichiers). Cela peut être utilisé pour charger des fichiers comme des schémas GraphQL définis dans un fichier .graphql ou .gql.

ligne 5
Cette ligne importe les resolvers pour le type User depuis un fichier externe UserResolver.js. Les résolveurs sont des fonctions qui spécifient la logique d'exécution pour les différentes opérations de GraphQL (comme la récupération de données).

ligne 6 Cette ligne fais la même chose que la ligne 4 mais cette fois-ci c'est pour la table **Contact**

</> Explication du contenu de **index.js**

2. Lecture des typedef :

```
const  typeDefs = gql(‘
  ${fs.readFileSync(’typedefs/UserTypedef.gql’,’utf8’)}
  ${fs.readFileSync(’typedefs/ContactTypedef.graphql’,’utf8’)}
  ‘)
```

Ici nous permettons la lecture des type definis afin de le rendre accessible dans le fichier **index.js**, puisque qu'il le faut pour le demarrage du serveur

3. Lecture des typedef :

```
  const resolvers = {
    Query:{
      ...userResolvers.Query,
      ...contactResolvers.Query
    },
  }
```

...userResolvers.Query et ...contactResolvers.Query : Ces deux lignes utilisent l'opérateur spread pour prendre tous les résolveurs (fonctions de lecture) définis dans userResolvers.Query et contactResolvers.Query, et les "fusionner" dans un seul objet.
Ce code est une façon de combiner plusieurs ensembles de résolveurs pour les rendre accessibles dans un seul objet Query.

4. Instanciation de ApolloServer avec les argument typeDef et relovers :

```
  const server = new ApolloServer(
    { typeDefs, resolvers }
  )
```

5. Instanciation de ApolloServer avec les argument typeDef et relovers :

```
  const server = new ApolloServer(
    { typeDefs, resolvers }
  )
```

Ce code démarre un serveur Apollo qui écoute sur le port 4000 et expose l'API GraphQL à l'URL /graphql. Une fois le serveur démarré, il affiche l'URL dans la console pour indiquer où l'API est accessible.

Enfin si vous avez suivit correctement toutes ces etapes et que vous retapez la commande **npm run dev** le serveur va demarrer correctement.

À présent, nous allons afficher la liste des utilisateurs et des contacts depuis la base de données. Si vous avez bien suivi, vous avez probablement remarqué

que nous avons défini, au niveau du typedef, la requête qui permet d'afficher l'ensemble des utilisateurs et des contacts. Il ne nous reste plus qu'à définir la logique dans le résolveur afin d'afficher les listes correspondantes de manière dynamique.

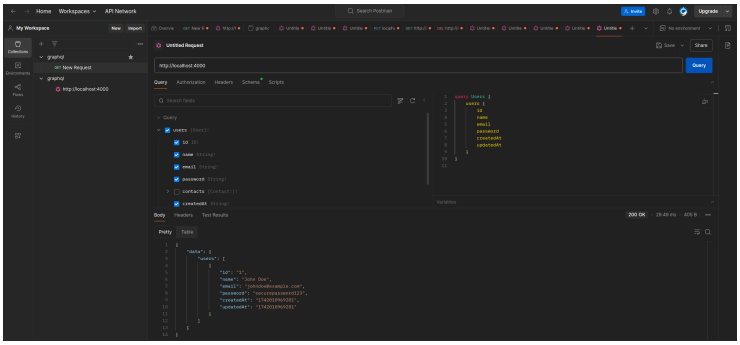
Voici le contenu duu fichier **UserResolver.js**

```
const prisma = new PrismaClient();
const userResolvers={
    Query: {
    users: async () =>{
         return await prisma.user.findMany
        (
            { include: { contacts: true } }
        )
        },
    },
};
export default userResolvers;
```

et voici pour **ContactResolver**

```

import { PrismaClient } from "@prisma/client";
const contactResolvers = {
    Query: {
        contacts: async () => await Prisma.contact.findMany({
    include: { user: true } }),
    },
};
export default contactResolvers;
```

Nous pouvons allez dans postman pour faire le test.



Vous pouvez également tester avec les contacts. Vous remarquerez que Postman permet d'afficher les contacts associés à un utilisateur, ainsi que l'utilisateur auquel chaque contact est rattaché. Cela permet une navigation fluide entre les utilisateurs et leurs contacts, facilitant ainsi l'exploitation des relations entre ces entités.

Maintenant nous allons afficher un contact specifique , comme d'habitude, nous

allons definir la requette d'abord puis apres le resolveur.

Voici la maniere dont nous avons définit la requette .

```
 contact(id: ID!): Contact!
```

Le fichier **typedefs/ContactTypedef.graphql** devrait ressembler à ceci :

```
type Contact {
  id:         ID!
  firstName:  String!
  lastName:   String!
  phone:      String!
  email:      String!
  address:    String!
  userId:     ID!
  user:       User!
  createdAt:  String!
  updatedAt:  String!
}
type Query{
    contacts: [Contact]!
    contact(id: ID!): Contact!
}
```

etn voici le resolveur :

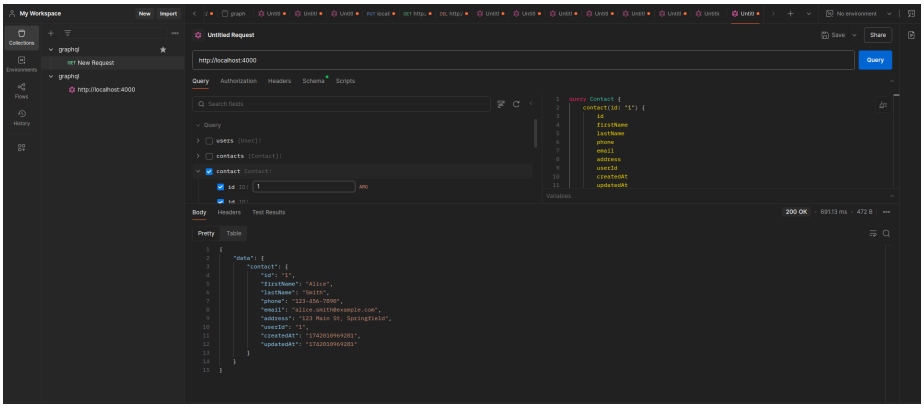
```
 contact: async(_, {id})=> await prisma.contact.findUnique({
            where: {id: parseInt(id)},
           include:  {user:true},
        })
```

Le fichier **resolveurs/ContactResolver.js** devrait ressembler à ceci :

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

const contactResolvers = {
    Query: {
        contacts: async () => await prisma.contact.findMany({
    include: { user: true } }),

        contact: async(_, {id})=> await prisma.contact.findUnique({
            where: {id: parseInt(id)},
           include:  {user:true},
        })
    },

    Mutation:{

    },
};

export default contactResolvers;
```

Si vous retournez dans Postman et actualisez l'interface, vous verrez qu'une nouvelle requête a été ajoutée. En cliquant dessus, un champ apparaîtra, vous

permettant de saisir l'ID de l'utilisateur que vous souhaitez afficher. Après avoir inséré l'ID, les informations correspondantes s'afficheront.



13 Nous allons aborder le concept de mutation (create, update, delete)

, toujour dans le fichier **typedefs/ContactTypedef.graphql**, ajoutez donc ce nouveau code.

```

type Mutation {
  ajouterContact(firstName: String, lastName:String,phone:String,
    email:String, address:String, userId:String):Contact
 }
```

Ce fichier doit maintenant ressebler a ceci

```
type Contact {
  id:          ID!
  firstName:   String!
  lastName:    String!
  phone:       String!
  email:       String!
  address:     String!
  userId:      ID!
  user:        User!
  createdAt:   String!
  updatedAt:   String!
}

type Query{
    contacts: [Contact]!
    contact(id: ID!): Contact!
}

type Mutation {
  ajouterContact(firstName: String, lastName:String,phone:String,
    email:String, address:String, userId:String):Contact
 }
```

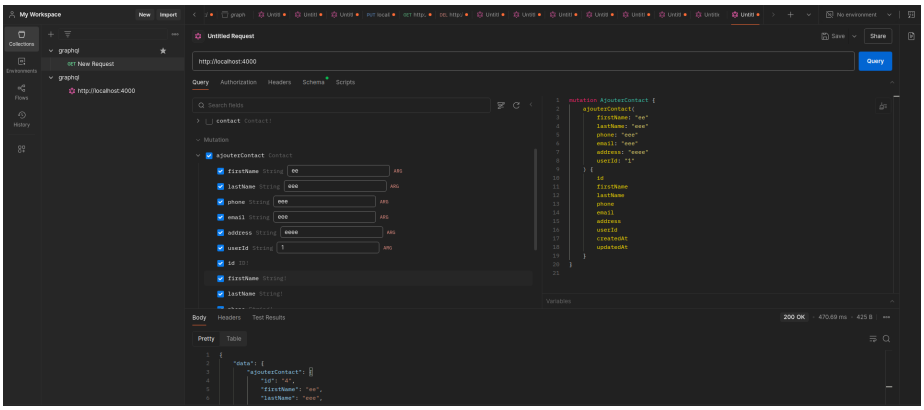
Si nous finissons, maintenant definissons le resolveur , voici le code.

```
   Mutation: {
        ajouterContact: async (_, { firstName, lastName, phone,
    email, address, userId }) => {
          const nouvoContact = await prisma.contact.create({
            data: {
              firstName,
              lastName,
              phone,
              email,
              address,
              userId: parseInt(userId, 10)
            }
          });

          return nouvoContact;
        }
      }
};
```

Le fichier devrait ressembler a ceci

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();
const contactResolvers = {
    Query: {
        contacts: async () => await prisma.contact.findMany({
    include: { user: true } }),
        contact: async(_, {id})=> await prisma.contact.findUnique({
            where: {id: parseInt(id)},
           include:  {user:true},
        })
    },

    Mutation: {
        ajouterContact: async (_, { firstName, lastName, phone,
    email, address, userId }) => {
          const nouvoContact = await prisma.contact.create({
            data: {
              firstName,
              lastName,
              phone,
              email,
              address,
              userId: parseInt(userId, 10)
            }
          });

          return nouvoContact;
        }
      }
};

export default contactResolvers;
```

Voici un exemple de test.



Faisons la mise a jour

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();
const contactResolvers = {
    Query: {
        contacts: async () => await prisma.contact.findMany({
    include: { user: true } }),
        contact: async(_, {id})=> await prisma.contact.findUnique({
            where: {id: parseInt(id)},
           include:  {user:true},
        })
    },

    Mutation: {
        ajouterContact: async (_, { firstName, lastName, phone,
    email, address, userId }) => {
          const nouvoContact = await prisma.contact.create({
            data: {
              firstName,
              lastName,
              phone,
              email,
              address,
              userId: parseInt(userId, 10)
            }
          });

          return nouvoContact;
        }
      }
};

export default contactResolvers;
```

Voici la definition de mise a jour

```
modifierContact(id:ID, firstName: String, lastName:String,phone:
   String, email:String, address:String, userId:String):Contact
```

Maintenant voici le resolveur

```
1
2   modifierContact: async (_, { id, firstName, lastName, phone,
      email, address }) => {
3           const contactIdInt = parseInt(id, 10);
4
5             if (isNaN(contactIdInt)) {
6               throw new Error("L'ID doit  tre  un nombre valide.");
7             }
8
9             try {
10              const updatedContact = await prisma.contact.update({
11                where: {
12                  id: contactIdInt,
13                },
14                data: {
15                  firstName: firstName || undefined,
16                  lastName: lastName || undefined,
17                  phone: phone || undefined,
18                  email: email || undefined,
19                  address: address || undefined,
20                },
21              });
22
23              return updatedContact;
24            } catch (error) {
25              throw new Error("Erreur lors de la mise    jour du
      contact : " + error.message);
26            }
27          },
```

Voici le contenu de tout le fichier jusqua present

```
1
2  import { PrismaClient } from "@prisma/client";
3
4 const prisma = new PrismaClient();
5 const contactResolvers = {
6     Query: {
7         contacts: async () => await prisma.contact.findMany({
      include: { user: true } }),
8         contact: async(_, {id})=> await prisma.contact.findUnique({
9             where: {id: parseInt(id)},
10           include:  {user:true},
11        })
12    },
13
14    Mutation: {
15        ajouterContact: async (_, { firstName, lastName, phone,
      email, address, userId }) => {
16          const nouvoContact = await prisma.contact.create({
17            data: {
18              firstName,
19              lastName,
20              phone,
21              email,
22              address,
23              userId: parseInt(userId, 10)
```

```
            }
          });

          return nouvoContact;
        },

        modifierContact: async (_, { id, firstName, lastName, phone, email, address }) => {
          const contactIdInt = parseInt(id, 10);

            if (isNaN(contactIdInt)) {
              throw new Error("L'ID doit  tre  un nombre valide.");
            }

            try {
              const updatedContact = await prisma.contact.update({
                where: {
                  id: contactIdInt,
                },
                data: {
                  firstName: firstName || undefined,
                  lastName: lastName || undefined,
                  phone: phone || undefined,
                  email: email || undefined,
                  address: address || undefined,
                },
              });

              return updatedContact;
            } catch (error) {
              throw new Error("Erreur lors de la mise    jour du contact : " + error.message);
            }
          },
        },
      };
export default contactResolvers;
```

Vous pouvez faire les tests a votre guise.

Suppression de contact

Pour supprimer un comtact , c'est toujours la meme procedure : on definit le schema la mutation comme telle

```
  supprimerContact(id:ID):Boolean

```

Voici a quoi devrait maintenant ressembler le fichier

```
type Contact {
  id:          ID!
  firstName:   String!
  lastName:    String!
  phone:       String!
  email:       String!
  address:     String!
  userId:      ID!
```

```
  user:       User!
  createdAt:  String!
  updatedAt:  String!
}

type Query{
    contacts: [Contact]!
    contact(id: ID!): Contact!
}

 type Mutation {
  ajouterContact(firstName: String, lastName:String,phone:String,
    email:String, address:String, userId:String):Contact,
  modifierContact(id:ID, firstName: String, lastName:String,phone:
    String, email:String, address:String, userId:String):Contact,
  supprimerContact(id:ID):Boolean
 }


```

ensuite on definit le resolveur

```
supprimerContact: async (_, { id }) => {
      const contactIdInt = Number(id);

      if (isNaN(contactIdInt)) {
        throw new Error("L'ID doit etre un nombre valide.");
      }

      try {
        const deletedContact = await prisma.contact.delete({
          where: {
            id: contactIdInt,
          },
        });

        return true;
      } catch (error) {
        throw new Error("Erreur lors de la suppression du contact :
     " + error.message);
      }
    },

```

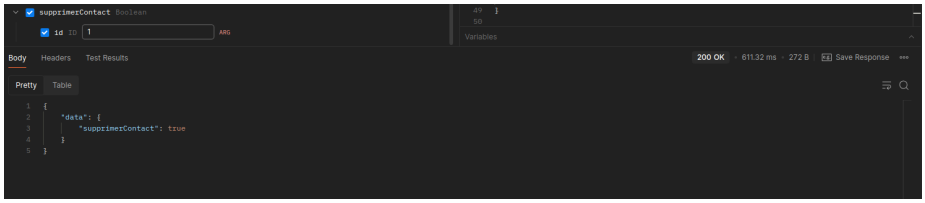
Apres le fichier resolver devrait ressembler a ceci

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();
import { verifyToken } from '../Auth/AuthUtils.js';
const contactResolvers = {
  Query: {

    contacts: async () => await prisma.contact.findMany({ include:
    { user: true } }),


    contact: async (_, { id }) =>
      await prisma.contact.findUnique({
```

```
      where: { id: parseInt(id) },
      include: { user: true },
    }),
},

Mutation: {
  ajouterContact: async (_, { firstName, lastName, phone, email,
  address, userId }) => {
    const nouvoContact = await prisma.contact.create({
      data: {
        firstName,
        lastName,
        phone,
        email,
        address,
        userId: parseInt(userId, 10),
      },
    });

    return nouvoContact;
  },

  modifierContact: async (_, { id, firstName, lastName, phone,
  email, address }) => {
    const contactIdInt = parseInt(id, 10);

    if (isNaN(contactIdInt)) {
      throw new Error("L'ID doit  tre  un nombre valide.");
    }

    try {
      const updatedContact = await prisma.contact.update({
        where: {
          id: contactIdInt,
        },
        data: {
          firstName: firstName || undefined,
          lastName: lastName || undefined,
          phone: phone || undefined,
          email: email || undefined,
          address: address || undefined,
        },
      });

      return updatedContact;
    } catch (error) {
      throw new Error("Erreur lors de la mise    jour du contact
  : " + error.message);
    }
  },

  supprimerContact: async (_, { id }) => {
    const contactIdInt = Number(id);

    if (isNaN(contactIdInt)) {
      throw new Error("L'ID doit  tre  un nombre valide.");
    }
```

```
67
68        try {
69          const deletedContact = await prisma.contact.delete({
70            where: {
71              id: contactIdInt,
72            },
73          });
74
75          return true;
76        } catch (error) {
77          throw new Error("Erreur lors de la suppression du contact :
      " + error.message);
78        }
79      },
80    },
81
82  };
83
84  export default contactResolvers;
85
86
```

Voici le resultat :



14 Implémentation de l'authentification avec JWT et BCRYPTJS

Pour implémenter un système d'authentification en utilisant GraphQL, voici le processus étape par étape :

14.1 Définition du Schéma GraphQL

Dans un premier temps, nous devons définir deux mutations dans notre schéma GraphQL :

- **Mutation register** : Cette mutation permet à un utilisateur de s'enregistrer en fournissant un nom , un email et un mot de passe.
- **Mutation login** : Cette mutation permet à un utilisateur de se connecter avec son email et son mot de passe.

Ces mutations retourneront un objet contenant un **token JWT** et les **informations de l'utilisateur**.

14.2 Modèle User

Le modèle `User` contient déja les champs suivants : `name`, `email` et `password`. Nous allons **hacher** le mot de passe avant de le stocker dans la base de données. Pour ce faire, nous utiliserons un module comme `bcryptjs` afin de sécuriser les mots de passe des utilisateurs.

14.3 Processus de la Mutation `register`

Lors de l'inscription, la mutation `register` devra effectuer plusieurs étapes :
- Vérifier si un utilisateur avec le même `email`
- Si l'utilisateur n'existe pas, un nouvel utilisateur sera créé avec les informations fournies.
- Le mot de passe de l'utilisateur sera **haché** à l'aide de `bcryptjs`.
- Un **token JWT** sera généré pour l'utilisateur, afin qu'il puisse être utilisé dans les requêtes futures.
- Le **token** et les informations de l'utilisateur (par exemple `id`, `email`) seront retournées en réponse.

14.4 Processus de la Mutation `login`

Lors de la connexion, la mutation `login` devra suivre ces étapes :
- Vérifier si l'utilisateur existe en recherchant par `email`.
- Comparer le mot de passe fourni par l'utilisateur avec le mot de passe haché stocké dans la base de données en utilisant `bcryptjs`.
- Si les mots de passe correspondent, un **token JWT** sera généré pour l'utilisateur et retourné.
- En cas d'échec de l'authentification (mauvais mot de passe), une erreur sera renvoyée.

14.5 Gestion du Token JWT

Le token JWT généré lors de l'inscription ou de la connexion contient les informations de l'utilisateur (par exemple, `id`) et une date d'expiration. Ce **token JWT** est envoyé au client et doit être inclus dans les en-têtes des requêtes futures afin d'accéder aux ressources protégées. Le token sera généralement placé dans l'en-tête `Authorization` des requêtes.

14.6 Sécurisation des Résolveurs

Afin de protéger certaines routes de l'application, un middleware sera utilisé pour vérifier la validité du token JWT. Cela garantit que seules les requêtes contenant un token valide sont autorisées à accéder aux données sensibles ou à effectuer certaines actions sur le serveur.

14.7 Résumé du Processus

En résumé, voici les principales étapes de l'implémentation de l'authentification :

- Création des mutations `register` et `login` dans le schéma GraphQL.
- Hachage des mots de passe avec `bcryptjs`.
- Génération d'un **token JWT** lors de l'inscription et de la connexion.
- Vérification du token JWT dans les requêtes futures pour sécuriser les ressources.

Ainsi, ce processus permet de mettre en place une authentification sécurisée dans une application GraphQL en utilisant les mutations `register` et `login`, avec un système de gestion des tokens JWT pour l'accès aux ressources protégées.

Maintenant que tout est bien expliqué et détaillé , nous allons commencer par maniser cela :

Pour se faire, nous allons d'abord l'installer avec la commande :

```
npm install jsonwebtoken

```

Cryptage du mot de passe pour des raisons de securites

```
npm install bcryptjs

```

Comme dit allons creer les mutation login et register mais avant tout definissons le type personnalise **AuthPayload** pour gérer la réponse d'authentification après une opération de connexion ou d'inscription.

```

type AuthPayload {
  token: String!
  user: User!
}

type Mutation {
  register(name: String!, email: String!, password: String!):
    AuthPayload!
  login(email: String!, password: String!): AuthPayload!
}

```

Le fichier **typedefs/UserTypedef.gql** devrait ressembler a ceci :

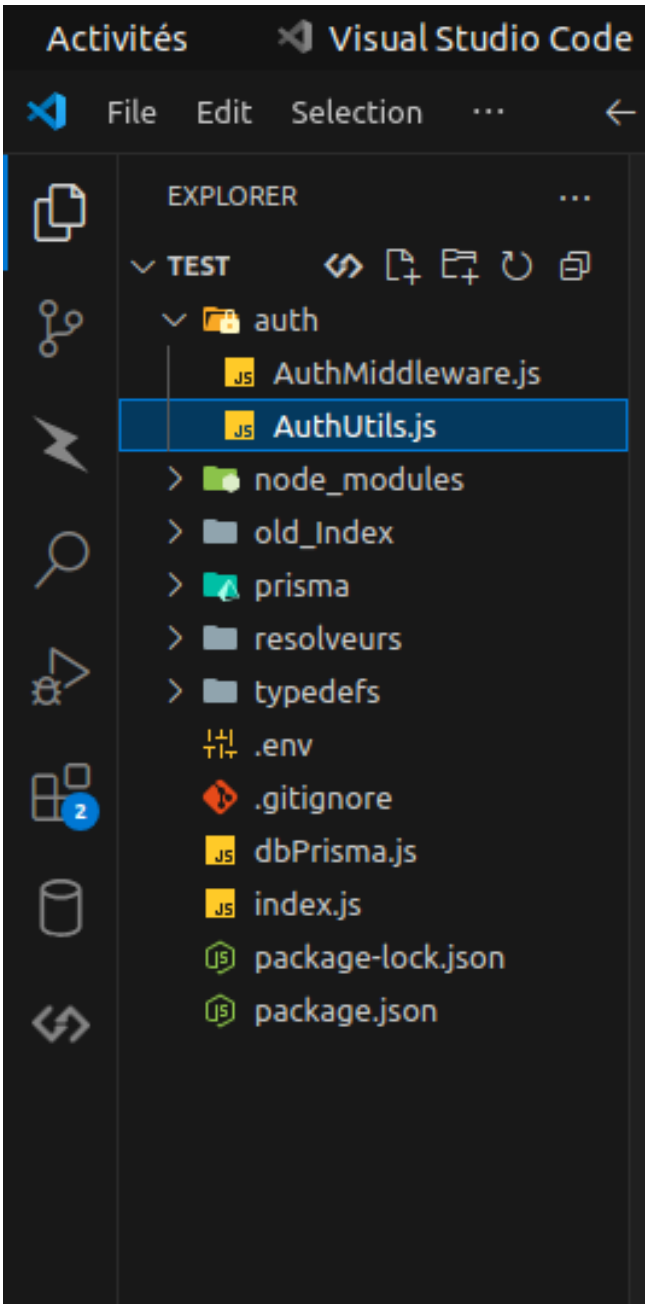
```

type User {
  id:       ID!
  name:    String!
  email: String!
  password: String!
  contacts:[Contact!]!
  createdAt: String!
  updatedAt: String!
}
```

```
11
12 type Query {
13     users:[User]!
14 }
15
16 type AuthPayload {
17   token: String!
18   user: User!
19 }
20
21 type Mutation {
22
23   register(name: String!, email: String!, password: String!):
       AuthPayload!
24
25
26   login(email: String!, password: String!): AuthPayload!
27 }
28
29
```

Maintenant que cela est fait, nous allons nous attaquer au dossier **Auth**, que nous n'avons pas encore modifié jusqu'à présent. Bien que ce ne soit pas absolument nécessaire, pour une meilleure organisation et cohérence, il est conseillé de renommer la lettre **A**, qui était en majuscule, en **a** afin de correspondre aux autres noms de dossiers.

Ensuite, le fichier **Auth.js** que nous avons créé dans ce dossier devra être renommé en **AuthMiddleware.js**. Enfin, il sera nécessaire d'ajouter un nouveau fichier, intitulé **AuthUtils.js**.

Voici un apercu :



Voici le contenu de **AuthUtils.js**

```
import bcrypt from "bcryptjs";
import jwt from "jsonwebtoken";

const SECRET_KEY = process.env.JWT_SECRET || "";
```

```
5
6  // Hashage du mot de passe
7  export const hashPassword = async (password) => {
8    return await bcrypt.hash(password, 10);
9  };
10
11 // V rification du mot de passe
12 export const verifyPassword = async (password, hashedPassword) => {
13   return await bcrypt.compare(password, hashedPassword);
14 };
15
16 // G n ration du token JWT
17 export const generateToken = (user) => {
18   return jwt.sign({ id: user.id, email: user.email }, SECRET_KEY, {
       expiresIn: "1h" });
19 };
20
21 // V rification du token JWT
22 export const verifyToken = (token) => {
23   try {
24     return jwt.verify(token, SECRET_KEY);
25   } catch (error) {
26     return null;
27   }
28 };
29
```

Voici le contenu de **AuthMiddleware.js**

```
1
2  import { verifyToken } from "./AuthUtils.js";
3
4  export const authMiddleware = (resolver) => async (parent, args,
       context, info) => {
5    const token = context.req.headers.authorization || "";
6
7    // V rifier si le token est valide
8    const user = verifyToken(token.replace("Bearer ", ""));
9
10   if (!user) throw new Error("Non authentifi  !");
11
12   // Ajouter l'utilisateur au contexte pour les r solveurs
13   context.user = user;
14
15   return resolver(parent, args, context, info);
16 };
17
18
```

14.8 Arguments des Résolveurs GraphQL

Cette fonction veut permettre d'expliquer le pourquoi cette fois ci le resolveur a pris 4 argument maintenant. Les résolveurs en GraphQL acceptent quatre arguments principaux :

— **parent** : Ce paramètre représente l'objet renvoyé par le résolveur du niveau supérieur. Pour la première requête, ce paramètre est généralement

`undefined` ou `null`. Dans les mutations ou les résolveurs imbriqués, il contient le résultat de l'appel précédent.

— **args** : Cet argument contient les arguments envoyés par le client dans la requête. Par exemple, si la requête demande un utilisateur avec un `id`, l'argument `args` contiendra cet `id` : { `id: 1` }. C'est grâce à cet argument que le résolveur sait quelles données doivent être récupérées ou modifiées.

— **context** : Le contexte est un objet partagé à travers toute la requête, utilisé pour passer des informations globales comme l'utilisateur authentifié, les paramètres de session, ou d'autres valeurs importantes. Cela permet aux résolveurs d'accéder à ces données sans les passer explicitement dans chaque appel.

— **info** : Cet argument contient des informations sur la requête en cours, telles que les champs demandés et le schéma de la requête. Il est généralement utilisé pour des opérations avancées comme la gestion de la sécurité ou le suivi des requêtes.

Apres avoir definit le schema passons au resolvers

```
Mutation: {
       // Inscription d'un utilisateur
       register: async (_, { name, email, password }) => {
           const existingUser = await prisma.user.findUnique({
   where: { email } });
           if (existingUser) throw new Error("Email  d  j
   utilis  !");

           const hashedPassword = await hashPassword(password);
           const user = await prisma.user.create({
             data: { name, email, password: hashedPassword },
           });

           const token = generateToken(user);

           return { token, user };
         },

         // Connexion d'un utilisateur
         login: async (_, { email, password }) => {
           const user = await prisma.user.findUnique({ where: {
   email } });
           if (!user) throw new Error("Utilisateur non trouv ");

           const valid = await verifyPassword(password, user.
   password);
           if (!valid) throw new Error("Mot de passe incorrect");

           const token = generateToken(user);

           return { token, user };
         },
       }

```

Le fichier **resolvers/UserResolvers.js** apres ajout du code du resolver devrait ressembler a ceci

```
import { PrismaClient } from "@prisma/client";
import { hashPassword, verifyPassword, generateToken } from "../Auth/AuthUtils.js";
const prisma = new PrismaClient();

const userResolvers={
    Query: {
    users: async () =>{
         return await prisma.user.findMany
        (
            { include: { contacts: true } }
        )

        },
    },

    Mutation: {
        // Inscription d'un utilisateur
        register: async (_, { name, email, password }) => {
            const existingUser = await prisma.user.findUnique({ where: { email } });
            if (existingUser) throw new Error("Email  d  j  utilis  !");

            const hashedPassword = await hashPassword(password);
            const user = await prisma.user.create({
              data: { name, email, password: hashedPassword },
            });

            const token = generateToken(user);

            return { token, user };
          },

          // Connexion d'un utilisateur
          login: async (_, { email, password }) => {
            const user = await prisma.user.findUnique({ where: { email } });
            if (!user) throw new Error("Utilisateur non trouv ");

            const valid = await verifyPassword(password, user.password);
            if (!valid) throw new Error("Mot de passe incorrect");

            const token = generateToken(user);

            return { token, user };
          },
        }
};
export default userResolvers;

```

Apres la definition du resolvers, Voici comment nous devrons adpter notre index, maintenant pour crrer l'instance d'apollo server il faut ajouter en plus de

typedefs, et resolver ajouter **Context** car le context les donnees comme : Informations d'authentification : Par exemple, l'ID de l'utilisateur ou le rôle de l'utilisateur, extrait du token JWT.

Données de la requête : Comme les en-têtes, les cookies, ou toute autre donnée associée à la requête HTTP

Accès aux services externes : Par exemple, une instance de base de données, un client API, ou des fonctions utilitaires.
le contenu de index :

```
1
2
3  import { ApolloServer } from "@apollo/server";
4  import { startStandaloneServer } from "@apollo/server/standalone";
5  import gql from "graphql-tag";
6  import fs from 'fs';
7  import userResolvers from './resolveurs/UserResolver.js';
8  import contactResolvers from './resolveurs/ContactResolver.js';
9  import { verifyToken , generateToken} from './Auth/AuthUtils.js';
       // Import de la fonction de v rification du token
10
11 import dotenv from 'dotenv';
12 dotenv.config();
13 // Charger les typedefs    partir des fichiers externes
14 const typeDefs = gql(`
15   ${fs.readFileSync('typedefs/UserTypedef.gql', 'utf8')}
16   ${fs.readFileSync('typedefs/ContactTypedef.graphql', 'utf8')}
17 `);
18
19 // Combine les r solveurs de contacts et utilisateurs
20 const resolvers = {
21   Query: {
22     ...userResolvers.Query,
23     ...contactResolvers.Query,
24   },
25   Mutation: {
26     ...userResolvers.Mutation,
27     ...contactResolvers.Mutation,
28   },
29 };
30
31
32 // Middleware pour v rifier le token JWT
33 const authContext = ({ req }) => {
34   const token = req.headers.authorization || '';
35
36   if (token) {
37     try {
38       const decoded = verifyToken(token.replace('Bearer ', ''));
39       if (decoded) {
40         return { userId: decoded.id };
41       }
42     } catch (error) {
43       throw new Error("Token invalide ou expir ");
44     }
```

```
  }

  throw new Error("Authentification requise");
};



const server = new ApolloServer({
  typeDefs,
  resolvers,
  context: authContext,
});


startStandaloneServer(server, {
  listen: { port: 4000 },
  path: "/graphql",
}).then(({ url }) => {
  console.log(`Serveur d marr  sur ${url}`);
});

```

apres il faut aller proteger le resolver que nous voulons, nous allons le faire pour contact.

```
contacts : async (_, args, context) => {
    if (!context.userId) {
        console.log(context.userId);

      throw new Error("Vous devez  tre  authentifi  pour acc der aux contacts");
    }
    return await prisma.contact.findMany({
        include: { user: true },
      });     },
```

Voici le resultat du test de la route register

Maintenant , il faut copier le tocken pour aller mettre dans le header
Voici le resultat :