# Projet de GraphQL


**MANGBA Bénédicta & TOYI François** Date: February 1, 2025

# 1 Définition

GraphQL : est un langage de requête pour les API.

# 2 Installation et Configuration

– Installer Node.js et npm (si ce n'est pas encore fait)
– Installer un framework backend Express, Apollo Server (nous allons utiliser Apollo Server)
– Installation de Prisma (pour la gestion de la base de données)

# 3 Concepts Fondamentaux de GraphQL

## Schéma

Un schéma définit la structure de l'API GraphQL. Il spécifie les types de données, les requêtes et les mutations disponibles. Le schéma est écrit en utilisant le langage de définition de schéma GraphQL.

## Types

GraphQL utilise des types pour décrire les données. Les types de base incluent :

- Scalar Types : Types primitifs comme Int, Float, String, Boolean, et ID.

- Object Types : Types définis par l'utilisateur qui contiennent des champs.

- Enum Types : Types qui définissent un ensemble de valeurs possibles.

- Interface Types : Types qui définissent un ensemble de champs que d'autres types doivent implémenter.

- Union Types : Types qui permettent à un champ de retourner plusieurs types différents.

## Requêtes (Queries)

Les requêtes sont utilisées pour lire les données. Elles permettent aux clients de spécifier exactement quelles données ils souhaitent récupérer, ce qui évite le problème du "over-fetching" (récupération de trop de données).

## Mutations

Les mutations sont utilisées pour modifier les données (ajouter, mettre à jour ou supprimer). Comme les requêtes, les mutations permettent également de spécifier les données à retourner après la modification.

## Résolveurs

Les résolveurs permettent d'écrire la logique du schéma.

Résolveurs : Ils définissent comment récupérer les données pour chaque champ dans le schéma.

Logique du schéma : Ils mettent en œuvre la logique de la façon dont les données doivent être obtenues (par exemple, depuis une base de données, une API externe, etc.).

# 4 Faisons notre premier pas en GraphQL (Affichons 'Hello')

```
import { ApolloServer, gql } from 'apollo-server';

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

```
// Définition des résolveurs
const resolvers = {
  Query: {
    hello: () => 'Bonjour tout le monde'
  }
};

// Création du serveur Apollo
const server = new ApolloServer({
  typeDefs,
  resolvers
});

// Démarrage du serveur
server.listen().then(({ url }) => {
  console.log(`Server ready at ${url}`);
});
```

# 5 Réalisons un petit projet de gestion de contact

## 5.1 Les commandes

Ici vous allez lancé un nouveau projet graphql avec la commande **npm init -y** puis copié le contenu de notre fichier **package.json** et collé sa dans votre fichier **package.json** mais je vous conseille de copié uniquement la partie des **dependencies** et ajouté la partie script avec nodemon et ensuite tapé la commande **npm install** .

```
{
  "name": "graphql-gestion-contacts",
  "version": "1.0.0",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "dev" : "nodemon src/index.js"
  },
  "keywords": [],
```

```
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "@apollo/server": "^4.11.3",
    "@prisma/client": "^6.2.1",
    "dotenv": "^16.4.7",
    "express": "^4.21.2",
    "graphql": "^16.10.0",
    "graphql-tag": "^2.12.6",
    "nodemon": "^3.1.9",
    "prisma": "^6.2.1"
  }
}
```

## 5.2 Créons un serveur GraphQL

Ici maintenant vous créer une arborescence comme suis, on a :

- **src** et dans lui on auras les fichier suivant :
- **index.js**
- **schema.gql** soit **schema.graphql** soit encore **schema.js** en se qui concerne les projet graphql il est conseillé que le fichier **schema** aient une extensions mentionnant que c'est vraiment du **graphql** que nous faisons mais dans notre cas on vas faire le nomée **schema.js** mais pour utiliser avoir une extentions graphql il faut installé certains dépandences et on as aussi un autre exemple avec sur un autre projet qu'on vous donneras pour ceux qui sont intéressé.
- Pour résumé on as à la racine packages.json, src qui contient les autres fichiers(index,schema et resolvers) en suite on à prisma et son .env et sont .gitignore à la racine et en fin notre node_module aussi à la racine de notre projet .

### 5.2.1 Fichier de configuration de prisma schema.prisma:

```
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
  contacts  Contact[]
}

model Contact {
  id        Int      @id @default(autoincrement())
  phone     String
  address   String
  userId    Int
  user      User     @relation(fields: [userId], references: [id])
}
```

### 5.2.2 Fichier index.js :

```
import { ApolloServer } from "@apollo/server";
import { startStandaloneServer } from "@apollo/server/standalone"; // Importer l
import typeDefs from "./schema.js";
import resolvers from "./resolvers.js";

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

// Démarre le serveur autonome
startStandaloneServer(server, {
  listen: { port: 4000 }, // Spécifie le port ici
}).then(({ url }) => {
  console.log(`Server ready at ${url}`);
});
```

### 5.2.3 Fichier schema.graphql :

```
import { gql } from 'graphql-tag';

const typeDefs = gql`
    type User {
        id: Int!
        name: String!
        email: String!
        contacts: [Contact!]! # Relation entre User et Contact
    }

    type Contact {
        id: Int!
        phone: String!
        address: String!
        user: User! # Relation entre Contact et User
    }

    type Query {
        users: [User!]! # Liste de tous les utilisateurs
        user(id: Int!): User # Détails d'un utilisateur spécifique
        contacts: [Contact!]! # Liste de tous les contacts
        contact(id: Int!): Contact # Détails d'un contact spécifique
        hello: String
    }

    type Mutation {
        createUser(name: String!, email: String!): User! # Créer un utilisateur
        createContact(phone: String!, address: String!, userId: Int!): Contact!
        updateUser(id: Int!, name: String, email: String): User # Mettre à jour
        updateContact(id: Int!, phone: String, address: String): Contact # Mettr
        deleteUser(id: Int!): Boolean # Supprimer un utilisateur
        deleteContact(id: Int!): Boolean # Supprimer un contact
    }
`;

export default typeDefs;
```

### 5.2.4 Fichier resolver.js

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

const prisma = new PrismaClient();

const resolvers = {
  Query: {
    hello: () => "Hello, World!", // Définir le résolveur pour "hello"
    users: async (parent, args) => {
      // console.log(await prisma.user.findMany({ include: { contacts: true } }))
      return await prisma.user.findMany({ include: { contacts: true } });
    },
    user: async (parent, { id }) => {
      return await prisma.user.findUnique({
        where: { id },
        include: { contacts: true },
      });
    },
    contacts: async (parent, args) => {
      return await prisma.contact.findMany({ include: { user: true } });
    },
    contact: async (parent, { id }) => {
      return await prisma.contact.findUnique({
        where: { id },
        include: { user: true },
      });
    },
  },
  Mutation: {
    createUser: async (parent, { name, email }) => {
      return await prisma.user.create({
        data: { name, email },
      });
    },
    createContact: async (parent, { phone, address, userId }) => {
      return await prisma.contact.create({
        data: { phone, address, userId },
      });
    },
    updateUser: async (parent, { id, name, email }) => {
```

```
      return prisma.user.update({
        where: { id },
        data: { name, email },
      });
    },
    updateContact: async (parent, { id, phone, address }) => {
      return prisma.contact.update({
        where: { id },
        data: { phone, address },
      });
    },
    deleteUser: async (parent, { id }) => {
      await prisma.user.delete({ where: { id } });
      return true;
    },
    deleteContact: async (parent, { id }) => {
      await prisma.contact.delete({ where: { id } });
      return true;
    },
  },
};

export default resolvers;
```

### 5.3 Tests de l'api avec postman

Adresse pour les test une fois la commande **npm run dev**, on as :
**http://localhost:4000/**

# 6 Authentification de notre api