Création d'une API avec le langage de requête GraphQL et son utilisation avec le framework Flutter

Bénédicta MANGBA & François TOYI

3 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 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.

1.4 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.

3 Création d'un projet GraphQL

3.1 Étape 1 : Initialiser un projet Node.js

Ouvrez votre terminal et créez un nouveau dossier pour votre projet :

```
mkdir mon-projet-graphql
cd mon-projet-graphql
```

Ensuite, initialisez un projet Node.js :

```
npm init -y
```

3.2 Étape 2 : Installer les dépendances

Nous allons installer les packages nécessaires pour GraphQL :

```
npm install @apollo/server graphql
```

3.3 Étape 3 : Créer un fichier index.js

Dans le dossier de votre projet, créez un fichier index.js et ajoutez le code suivant :

```
// Importation des modules n cessaires
import { ApolloServer } from '@apollo/server';
import { startStandaloneServer } from '@apollo/server/standalone';

// D finition du sch ma GraphQL
const typeDefs = `#graphql
  type Query {
    hello: String
  }
`;

// D finition des r solveurs
const resolvers = {
  Query: {
    hello: () => 'Bonjour tout le monde',
  },
};

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

// D marrage du serveur
const { url } = await startStandaloneServer(server, {
  listen: { port: 4000 },
});

console.log(`Server ready at ${url}`);
```

3.4 Étape 4 : Lancer le serveur

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

```
node index.js
```

Vous devriez voir le message suivant dans votre terminal :

```
Server ready at http://localhost:4000/
```

3.5 Étape 5 : Tester l'API

Ouvrez votre navigateur et allez à l'adresse http://localhost:4000. Vous verrez l'interface GraphQL Playground. Entrez la requête suivante pour tester votre API :

```
query {
  hello
}
```

Vous devriez obtenir la réponse suivante :

```
{
  "data": {
    "hello": "Bonjour tout le monde"
  }
}
```

4 Tests pratiques

4.1 Ajout de nouveaux types et requêtes

Modifiez le fichier index.js pour ajouter un nouveau type User et une requête pour récupérer un utilisateur :

```
const typeDefs = `#graphql
  type Query {
    hello: String
    user(id: ID!): User
  }

  type User {
    id: ID!
    name: String!
    email: String!
  }
`;

const users = [
  { id: '1', name: 'benedicta', email: 'benedicta@example.com' },
  { id: '2', name: 'francois', email: 'francois@example.com' },
];

const resolvers = {
  Query: {
    hello: () => 'Bonjour tout le monde',
    user: (parent, args) => users.find(user => user.id === args.id)
    ,
  },
};
```

Testez la nouvelle requête dans GraphQL Playground :

```
query {
  user(id: "1") {
    id
    name
    email
  }
}
```

4.2 Utilisation de variables dans les requêtes

Dans GraphQL Playground, vous pouvez utiliser des variables pour rendre vos requêtes plus dynamiques. Par exemple :

```
query GetUser($userId: ID!) {
  user(id: $userId) {
    id
    name
    email
  }
}
```

Avec les variables suivantes :

```
{
  "userId": "2"
}
```

4.3 Introduction aux mutations

Ajoutez une mutation pour créer un nouvel utilisateur :

```
const typeDefs = `#graphql
  type Query {
    hello: String
    user(id: ID!): User
  }

  type Mutation {
    createUser(name: String!, email: String!): User
  }

  type User {
    id: ID!
    name: String!
    email: String!
  }
`;

const users = [];
let nextId = 1;

const resolvers = {
  Query: {
    hello: () => 'Bonjour tout le monde',
    user: (parent, args) => users.find(user => user.id === args.id)
    ,
  },
  Mutation: {
    createUser: (parent, args) => {
      const user = { id: String(nextId++), name: args.name, email:
    args.email };
      users.push(user);
      return user;
    },
  },
};
```

Testez la mutation dans GraphQL Playground :

```
mutation {
  createUser(name: "Charlie", email: "charlie@example.com") {
```

```
    id
    name
    email
  }
}
```

5 Fin de la section de test

Félicitations ! 🏆Vous avez créé votre première API GraphQL et effectué des tests pratiques. Dans la prochaine partie, nous aborderons des concepts plus avancés, comme les souscriptions et l'intégration avec une base de données. Amusez-vous bien ! ☺

6 Projet réaliste : Gestion des contacts avec Prisma et Flutter

6.1 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, Nodemon pour permettre le redémarrage à chaud et JWT pour la gestion de l'authentification. Plus tard, nous consommerons cette API avec Flutter pour créer une application mobile. Ce projet sera amusant et vous permettra de voir comment tout cela fonctionne ensemble.

6.2 Analogies avec les APIs REST

Pour ceux qui n'ont jamais utilisé d'APIs REST, imaginez que vous êtes dans un restaurant. Avec une API REST, c'est comme si vous deviez commander chaque plat séparément à différents serveurs. Par exemple, vous commandez votre entrée à un serveur, votre plat principal à un autre, et votre dessert à un troisième. Cela peut être fastidieux et inefficace.

Avec GraphQL, c'est comme si vous aviez un seul serveur qui prend votre commande complète en une seule fois. Vous pouvez demander exactement ce que vous voulez, sans avoir à passer par plusieurs serveurs. C'est plus flexible et plus efficace.

6.3 Création d'un nouveau projet structuré

Nous allons maintenant créer un nouveau projet structuré pour notre application de gestion des contacts. Suivez les étapes ci-dessous :

6.3.1 Étape 1 : Initialiser un nouveau projet Node.js

Créez un nouveau dossier pour votre projet :

```
mkdir gestion-contacts
cd gestion-contacts
```

Initialisez un projet Node.js :

```
npm init -y
```

6.3.2 Étape 2 : Installer les dépendances nécessaires

Installez les packages nécessaires pour GraphQL, Prisma, et Nodemon :

```
npm install @apollo/server graphql @prisma/client
npm install --save-dev prisma nodemon
```

6.3.3 Étape 3 : Configurer Prisma

Initialisez Prisma dans votre projet :

```
npx prisma init
```

Cela va créer un dossier prisma avec un fichier schema.prisma. Ouvrez ce fichier et configurez votre base de données. Par exemple, pour utiliser SQLite :

```
datasource db {
  provider = "sqlite"
  url = "file:./dev.db"
}

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

model User {
  id        Int      @id @default(autoincrement())
  name      String   @db.VarChar(100) @default("")
  email     String   @unique @db.VarChar(255)
  password  String   @db.VarChar(255)
  contacts  Contact[]
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}

model Contact {
  id        Int      @id @default(autoincrement())
  phone     String   @db.VarChar(15)
  address   String   @db.VarChar(255)
  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 :

```
npx prisma migrate dev --name init
npx prisma generate
```

6.4 Structure du projet

Voici la structure du projet `gestion-contacts` :

```
gestion-contacts/
          prisma/
                schema.prisma
          src/
                schema.js
                resolvers.js
                index.js
          package.json
          document.tex
```

6.5 Fichier schema.js

Voici le contenu du fichier `schema.js` :

```
// src/schema.js
import { gql } from 'graphql-tag';

const typeDefs = gql`
  type User {
    id: Int!
    name: String!
    email: String!
    password: String @deprecated(reason: "Not exposed in API")
    contacts: [Contact!]!
    createdAt: String!
    updatedAt: String!
  }

  type Contact {
    id: Int!
    phone: String!
    address: String!
    user: User!
    createdAt: String!
    updatedAt: String!
  }

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

  type Query {
    users: [User!]!
    user(id: Int!): User
    contacts: [Contact!]!
    contact(id: Int!): Contact
    hello: String
```

```
  }

  type Mutation {
    createUser(name: String!, email: String!): User!
    createContact(phone: String!, address: String!, userId: Int!): Contact!
    updateUser(id: Int!, name: String, email: String): User
    updateContact(id: Int!, phone: String, address: String): Contact
    deleteUser(id: Int!): Boolean
    deleteContact(id: Int!): Boolean
    signup(name: String!, email: String!, password: String!): AuthPayload!
    login(email: String!, password: String!): AuthPayload!
  }
`;

export default typeDefs;
```

6.6 Fichier resolvers.js

Voici le contenu du fichier `resolvers.js` :

```
// src/resolvers.js
import { PrismaClient } from '@prisma/client';
import { signup, login } from './auth.js';

const prisma = new PrismaClient();

const resolvers = {
  Query: {
    hello: () => 'Hello, World!',
    users: async () => await prisma.user.findMany({ include: { contacts: true } }),
    user: async (_, { id }) => {
      const user = await prisma.user.findUnique({ where: { id }, include: { contacts: true } });
      if (!user) throw new Error('User not found');
      return user;
    },
    contacts: async () => await prisma.contact.findMany({ include: { user: true } }),
    contact: async (_, { id }) => {
      const contact = await prisma.contact.findUnique({ where: { id }, include: { user: true } });
      if (!contact) throw new Error('Contact not found');
      return contact;
    },
  },
  Mutation: {
    createUser: async (_, { name, email }) => {
      return await prisma.user.create({ data: { name, email } });
    },
    createContact: async (_, { phone, address }, { user }) => {
      if (!user) throw new Error('Not authenticated');
      return await prisma.contact.create({
```

```
        data: { phone, address, userId: user.userId },
      });
    },
    updateUser: async (_, { id, name, email }) => {
      const userExists = await prisma.user.findUnique({ where: { id } });
      if (!userExists) throw new Error('User not found');
      return prisma.user.update({ where: { id }, data: { name, email } });
    },
    updateContact: async (_, { id, phone, address }) => {
      const contactExists = await prisma.contact.findUnique({ where: { id } });
      if (!contactExists) throw new Error('Contact not found');
      return prisma.contact.update({ where: { id }, data: { phone, address } });
    },
    deleteUser: async (_, { id }) => {
      const userExists = await prisma.user.findUnique({ where: { id } });
      if (!userExists) throw new Error('User not found');
      await prisma.user.delete({ where: { id } });
      return true;
    },
    deleteContact: async (_, { id }) => {
      const contactExists = await prisma.contact.findUnique({ where: { id } });
      if (!contactExists) throw new Error('Contact not found');
      await prisma.contact.delete({ where: { id } });
      return true;
    },
    signup,
    login,
  },
};

export default resolvers;
```

6.7 Fichier index.js

Voici le contenu du fichier `index.js` :

```
// src/index.js
import { ApolloServer } from '@apollo/server';
import { startStandaloneServer } from '@apollo/server/standalone';
import dotenv from 'dotenv';
import typeDefs from './schema.js';
import resolvers from './resolvers.js';
import jwt from 'jsonwebtoken';

dotenv.config();

const PORT = process.env.PORT || 4000;

const server = new ApolloServer({
  typeDefs,
```

```
  resolvers,
  context: async ({ req }) => {
    const authHeader = req.headers.authorization || '';
    const token = authHeader.replace('Bearer ', '');

    if (!token) {
      console.log('       Aucun token re u');
      return { userId: null };
    }

    try {
      const decoded = jwt.verify(token, process.env.JWT_SECRET);
      console.log('    Token d cod   :', decoded);
      return { userId: decoded.userId };
    } catch (error) {
      console.error('    Token invalide :', error.message);
      return { userId: null };
    }
  },
});

// D marre le serveur autonome
startStandaloneServer(server, {
  listen: { port: PORT },
})
  .then(({ url }) => {
    console.log(`Server ready at ${url}`);
  })
  .catch((err) => {
    console.error('    Error starting server:', err);
  });
```

6.7.1 Étape 4 : Configurer Nodemon

Ajoutez un script dans votre package.json pour utiliser Nodemon :

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "dev": "nodemon src/index.js"
},
```

Maintenant, vous pouvez démarrer votre serveur avec :

```
npm run dev
```

Nodemon surveillera les changements dans vos fichiers et redémarrera automatiquement le serveur, ce qui est très pratique pendant le développement.
Ne vous inquiettez pas par rapport aux volumes des codes des fichiers se trouvant dans le dossier src, nous allons expliquer tout cela pas à pas prochainement.

7 Conclusion

Félicitations ! Vous avez maintenant un projet GraphQL fonctionnel avec Prisma et Nodemon. Dans les prochaines étapes, nous allons explorer l'intégra-

tion avec Flutter pour créer une application mobile. Restez à l'écoute pour la suite ! ☺