


# Création d'une API avec le langage de requête GraphQL

Bénédicta MANGBA & François TOYI

14 mars 2025

## Table des matières

<b>1</b>	<b>Introduction à GraphQL</b>	<b>3</b>
1.1	Qu'est-ce que GraphQL ?	3
1.2	Concepts de base de GraphQL	3
1.3	Les types en GraphQL ?	4
1.4	Avantages de GraphQL	5
1.5	Limites de GraphQL	5
<b>2</b>	<b>Prérequis pour utiliser GraphQL sur Ubuntu</b>	<b>5</b>
2.1	Installation des outils nécessaires	5
2.2	Passons à la pratique! 🐦	5
<b>3</b>	<b>Configuration de l'environnement de travail</b>	<b>6</b>
<b>4</b>	<b>Initialisation du projet</b>	<b>6</b>
4.1	Commandes de base et explication	6
<b>5</b>	<b>Modification du fichier package.json</b>	<b>7</b>
<b>6</b>	<b>Installation des dépendances</b>	<b>7</b>
6.1	Commandes de base et explication	7
<b>7</b>	<b>Ajout du code dans index.js</b>	<b>8</b>
<b>8</b>	<b>Exécution du serveur</b>	<b>9</b>
<b>9</b>	<b>Test pratique</b>	<b>10</b>
9.1	Ajout de nouveaux types et requêtes ✍️📊	10
<b>10</b>	<b>Fin de la section de test</b>	<b>20</b>

<b>11</b>	<b>Projet réaliste : Gestion des contacts avec Prisma et Flutter</b>	<b>20</b>
11.1	Concept . . . . .	20
11.2	Analogies avec les APIs REST . . . . .	21
11.3	Restructuration du projet  . . . . .	21
11.3.1	Étape 3 : Configurer Prisma . . . . .	22
11.4	Structure du projet . . . . .	22
11.5	Fichier schema.js . . . . .	23
11.6	Fichier resolvers.js . . . . .	24
11.7	Fichier index.js . . . . .	25
11.7.1	Étape 4 : Configurer Nodemon . . . . .	26
<b>12</b>	<b>Conclusion</b>	<b>26</b>

# 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 ?



### Les types GraphQL expliqués simplement

#### 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)

```
1 type Livre {  
2   titre: String!  
3   pages: Int!  
4   disponible: Boolean!  
5 }
```

#### 3. Types racines >\_ (Points d'entrée)

```
1 # Lire des donnees  
2 type Query {  
3   livres: [Livre!]!  
4 }  
5  
6 # Modifier des donnees  
7 type Mutation {  
8   ajouterLivre(titre: String!): Livre!  
9 }  
10  
11 # Recevoir en temps reel  
12 type Subscription {  
13   nouveauLivre: Livre!  
14 }
```

#### 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 </>

```
1 type Query {  
2   produits: [Produit!]!  
3 }  
4  
5 type Produit {  
6   id: ID!  
7   nom: String!  
8   prix: Float!  
9   enStock: Boolean!  
10 }
```

## 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 :

```
1 sudo apt update
2 sudo apt install nodejs npm
3
```

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 :

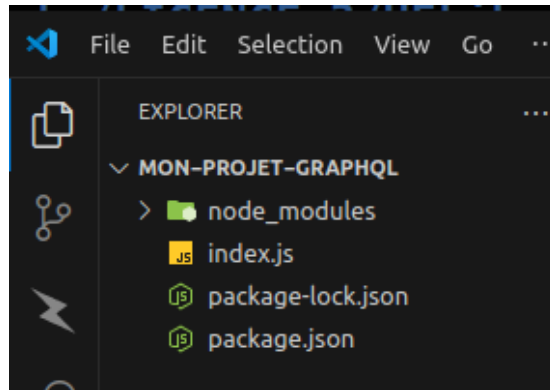
```
1 sudo snap install --classic code
2
```

## 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 :

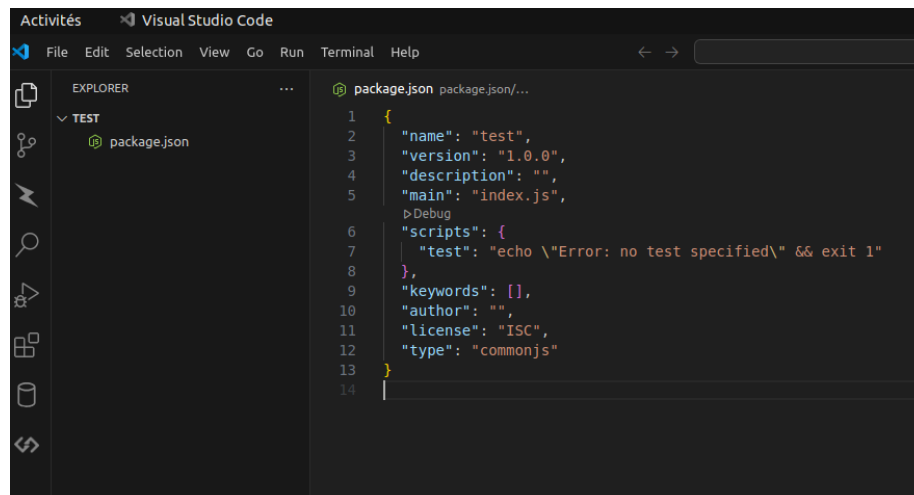


### 4 Initialisation du projet

#### 4.1 Commandes de base et explication

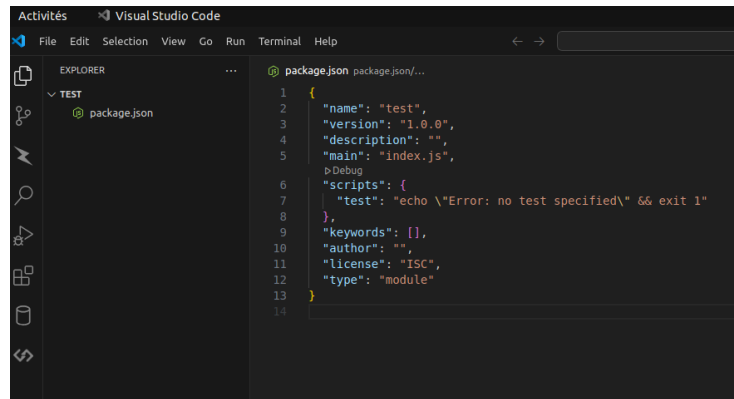
```
1 mkdir test           // Creation du dossier du projet
2 cd test              // Acces au dossier du projet
3 npm init -y          // Initialisation d'un projet Node.js
```

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

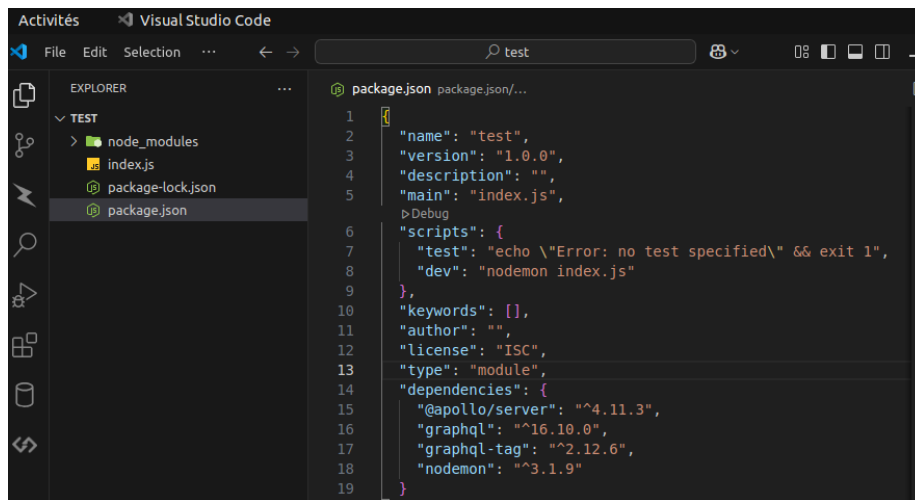
### 6.1 Commandes de base et explication

Accéder à votre terminal et exécutez les commandes suivantes

```
1 npm install @apollo/server graphql // Installation des dépendances
2 npm install graphql-tag // Installation de la bibliothèque graphql-tag
3 npm install nodemon // pour permettre le redémarrage à chaud du serveur
```

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

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



## 7 Ajout du code dans index.js

```

1 import { ApolloServer } from '@apollo/server';
2 import { gql } from 'graphql-tag';
3 import { startStandaloneServer } from "@apollo/server/standalone";
4
5 // D finition du schema GraphQL
6 const typeDefs = gql`
7   type Query {
8     hello: String
9   }
10 `;
11
12 // Resolveur pour le schema
13 const resolvers = {
14   Query: {
15     hello: () => 'Hello, world!',
16   },
17 };
18
19 // Creer une instance d'ApolloServer
20 const server = new ApolloServer({
21   typeDefs,
22   resolvers,
23 });
24
25 // Lancer le serveur en utilisant startStandaloneServer
26 startStandaloneServer(server, {
27   listen: { port: 4000 }
28 }).then(({ url }) => {
29   console.log(`Server ready at ${url}`);
30 });

```

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) :**

```
1   type Query {  
2     hello: String # Expose une requete 'hello'  
3   },  
4   }
```

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

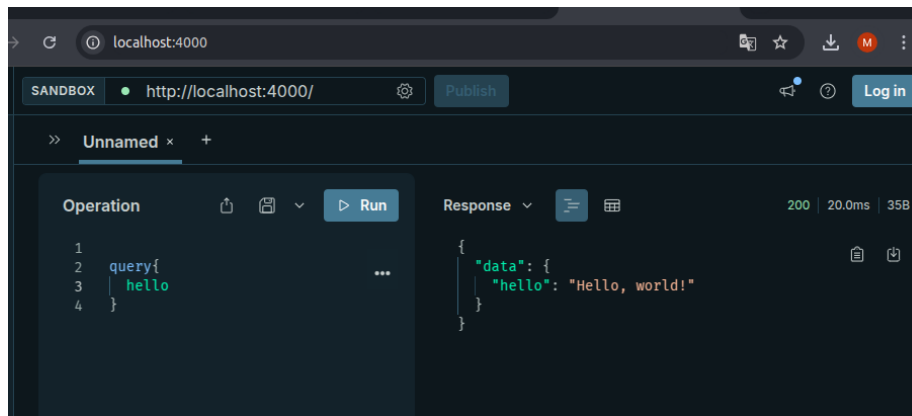
Dans votre terminal, exécutez la commande suivante pour démarrer le serveur :

```
1 npm run dev
```

Vous devriez voir le message suivant dans votre terminal :

```
yetnam@yetnam:~/Documents/LICENCE_3/UELibre/finalisation_cours_a_ramasser/test$ npm run dev  
 > test@1.0.0 dev  
 > nodemon index.js  
  
[nodemon] 3.1.9  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,cjs,json  
[nodemon] starting `node index.js`  
Server ready at http://localhost:4000/  
█
```

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 :

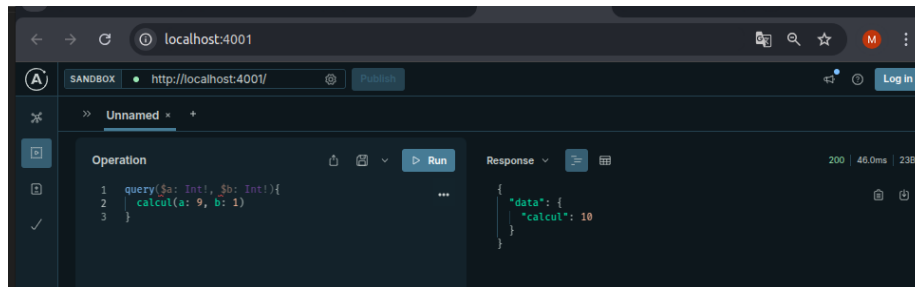
- `</>` Créer une requête calcul pour additionner deux nombres
- 🧩 Déclarer les types GraphQL appropriés
- ⚙️ Résolveur effectuant  $a + b$

```
12 mars 20:39
...
index.js index.js(resolvers)
1 import { ApolloServer } from '@apollo/server';
2 import { gql } from 'graphql-tag';
3 import { startStandaloneServer } from '@apollo/server/standalone';
4
5 // Définition du schéma GraphQL
6 const typeDefs = gql`
7   type Query {
8     hello: String
9     calcul(a: Int!, b: Int!): Int
10  }
11 `;
12
13 // Résolvants pour le schéma
14 const resolvers = {
15   Query: {
16     hello: () => 'Hello, world!',
17     calcul: (_, args) => {
18       return args.a+args.b;
19     }
20   },
21 };
22
23 // Créer une instance d'ApolloServer
24 const server = new ApolloServer({
25   typeDefs,
26   resolvers,
27 });
28
29 // Lancer le serveur en utilisant startStandaloneServer
30 startStandaloneServer(server, {
31   listen: { port: 4001 }
32 }).then(({ url }) => {
33   console.log(`Server ready at ${url}`);
34 });
35
36
37
```

 Capture du code modifié

#### Checklist :

- ✓ 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 sérieux et comprenons le code suivant, cela nous est très important :

## Structure du code

### 1. Schéma GraphQL :

```
1 type Query {  
2   "Additionne deux entiers (a + b)"  
3   calcul(a: Int!, b: Int!): Int  
4 }
```

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

### 2. Résolveur JavaScript :

```
1 const resolvers = {  
2   Query: {  
3     calcul: (_, args) => {  
4  
5       // 2. Calcul  
6       const result = args.a + args.b;  
7  
8       // 3. Retour  
9       return result;  
10    }  
11  }  
12 };
```

## 💡 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` :

```
1 type Tache {  
2   id: ID!  
3   titre: String!  
4   terminer: Boolean!  
5 }
```

## 2. Définition de la requête

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

```
1 taches: [Tache]
```

## 3. Définition du resolveur

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

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

## 4. Exemple de données

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

```
1 let tasks = [  
2   { id: 1, titre: "Apprendre GraphQL", terminer: false },  
3   { id: 2, titre: "Faire les courses", terminer: true }  
4 ];
```

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

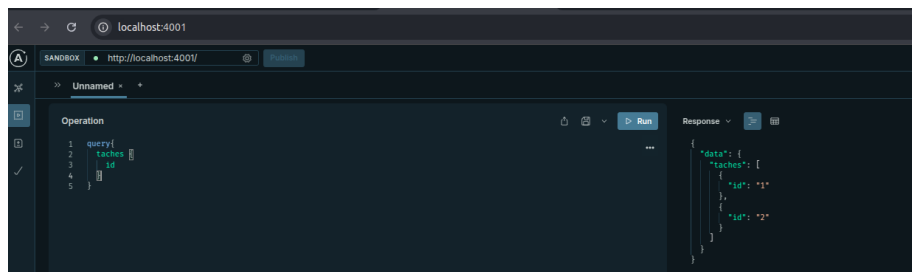
```
1 import { ApolloServer } from '@apollo/server';  
2 import { gql } from 'graphql-tag';  
3 import { startStandaloneServer } from "@apollo/server/standalone";  
4  
5 // Définition du schéma GraphQL  
6 const typeDefs = gql`  
7   type Tache {  
8     id: ID!
```

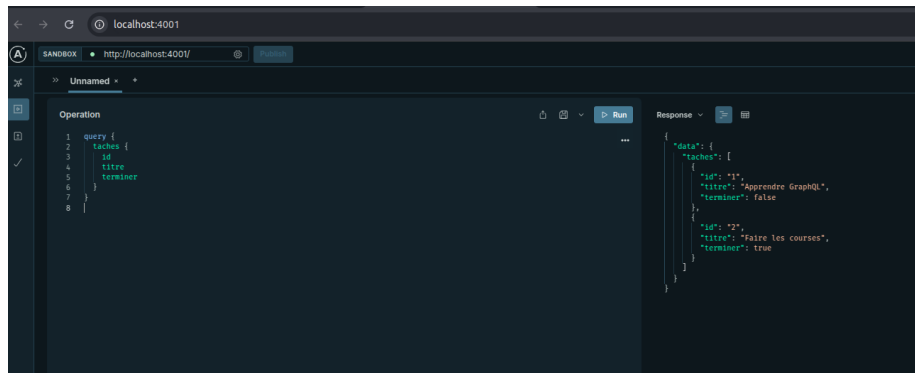
```

9     titre: String!
10     terminer: Boolean!
11 }
12
13 type Query {
14     hello: String
15     calcul(a: Int!, b: Int!): Int
16     taches: [Tache]
17 }
18
19
20
21 let tasks = [
22     { id: 1, titre: "Apprendre GraphQL", terminer: false },
23     { id: 2, titre: "Faire les courses", terminer: true }
24 ];
25
26 // Resolvants pour le schema
27 const resolvers = {
28     Query: {
29         hello: () => 'Hello, world!',
30         calcul: (_, args) => args.a + args.b,
31         taches: () => tasks
32     }
33 };
34
35 // Creer une instance d'ApolloServer
36 const server = new ApolloServer({
37     typeDefs,
38     resolvers,
39 });
40
41 // Lancer le serveur en utilisant startStandaloneServer
42 startStandaloneServer(server, {
43     listen: { port: 4001 }
44 }).then(({ url }) => {
45     console.log('Server ready at ${url}');
46 });

```

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 pourrions maintenant manipuler notre type **Mutation**

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

```
1
2 type Tache {
3   id: ID!
4   titre: String!
5   terminer: Boolean!
6 }
```

Declarer la mutation

```
1
2 type Mutation{
3   ajouterTache(titre: String!, terminer: Boolean!): Tache
4
5 }
```

Apres definir le corps de la mutation dans le resolveur



```

1
2 Mutation:{
3   ajouterTache: (_, {titre, terminer}) => {
4     const nouvelleTache = {
5       id: tache.length + 1,
6       titre,
7       terminer
8     };
9     tache.push(nouvelleTache);
10    return nouvelleTache;
11  }
12 }

```

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

```

1
2 import { ApolloServer } from '@apollo/server';
3 import { gql } from 'graphql-tag';
4 import { startStandaloneServer } from "@apollo/server/
5   standalone";
6
7 // D finition du sch ma GraphQL
8 const typeDefs = gql`
9   type Tache {
10     id: ID!
11     titre: String!
12     terminer: Boolean!
13   }
14   type Query {
15     hello: String
16     calcul(a: Int!, b: Int!): Int
17     taches: [Tache]
18   }
19   type Mutation{
20     ajouterTache(titre: String!, terminer: Boolean!): Tache
21   }
22 `;
23 let taches = [
24   { id: 1, titre: "Apprendre GraphQL", terminer: false },
25   { id: 2, titre: "Faire les courses", terminer: true }
26 ];
27 // Resolvants pour le schema
28 const resolvers = {
29   Query: {
30     hello: () => 'Hello, world!',
31     calcul: (_, args) => args.a + args.b,
32     taches :()=> taches
33   },
34   Mutation:{
35     ajouterTache: (_, {titre, terminer}) => {

```

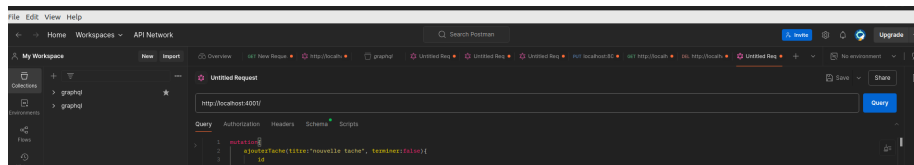
```

35     const nouvelleTache = {
36       id: taches.length + 1,
37       titre,
38       terminer
39     };
40     taches.push(nouvelleTache);
41     return nouvelleTache;
42   }
43 }
44 };
45 // Creer une instance d'ApolloServer
46 const server = new ApolloServer({
47   typeDefs,
48   resolvers,
49 });
50 // Lancer le serveur en utilisant startStandaloneServer
51 startStandaloneServer(server, {
52   listen: { port: 4001 }
53 }).then(({ url }) => {
54   console.log(`Server ready at ${url}`);
55 });

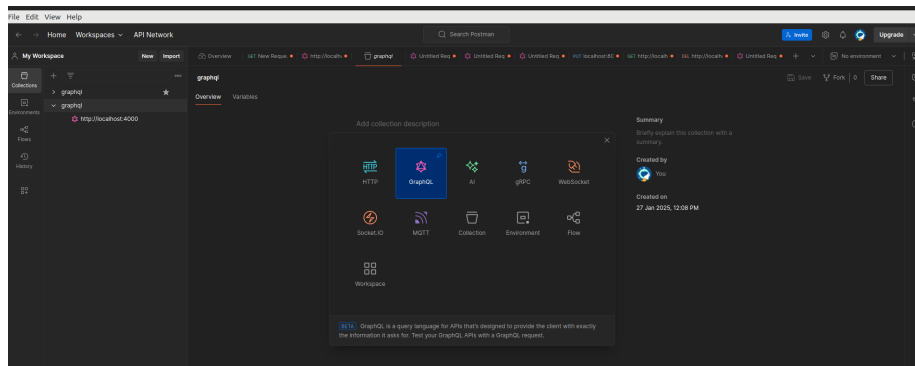
```

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.

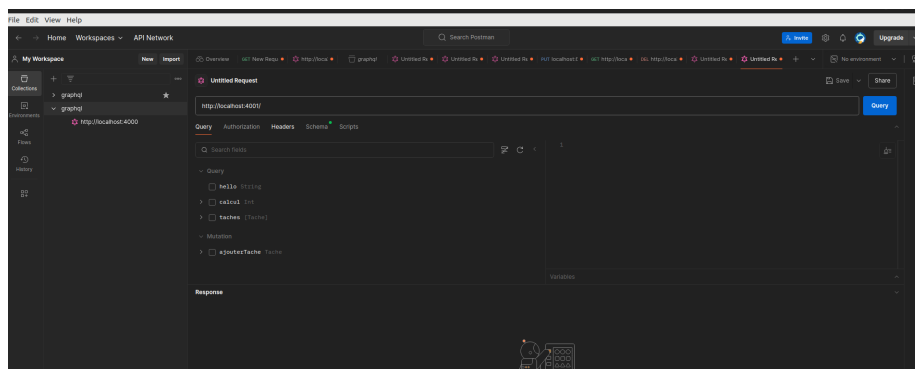
D'abord, téléchargez Postman en suivant ce lien : [Télécharger Postman](#). Une fois l'installation terminée, si vous ne l'avez pas encore configuré, voici comment procéder :



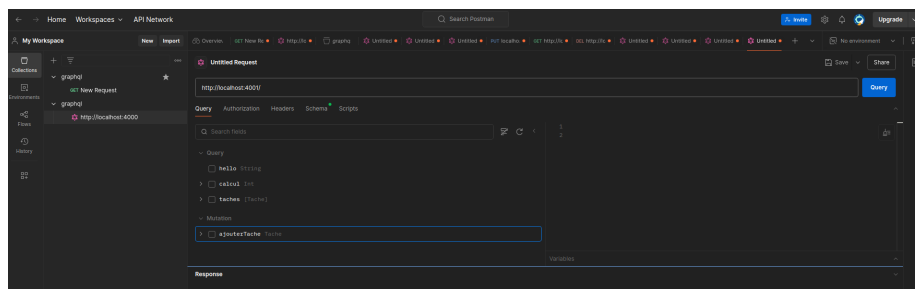
Ensuite, en haut à gauche, là où il est écrit New et Import, cliquez sur New pour ouvrir un nouvel onglet. Sélectionnez ensuite GraphQL. Voici l'illustration :



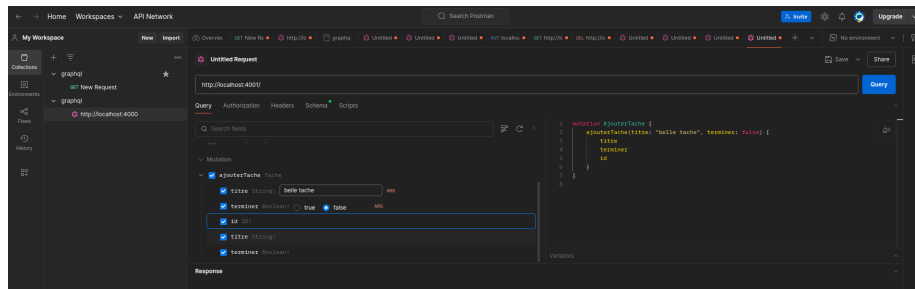
Une fois que vous avez cliqué dessus, entrez l'URL de votre endpoint unique en haut, comme montré ci-dessous. N'oubliez pas de remplacer l'exemple par le vôtre.



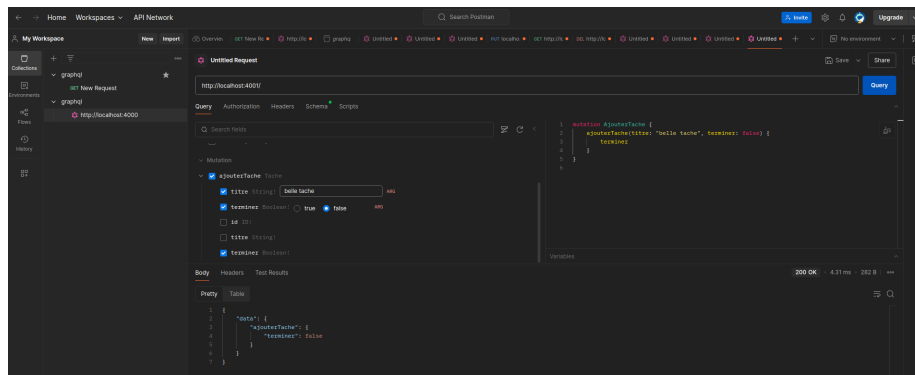
Maintenant que tout est prêt, voici comment tester la création d'une tâche dans Postman. Une fois que vous accédez à Postman, vous verrez la liste de vos requêtes et mutations, comme montré ci-dessous :



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 manipuler 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 avec Flutter pour créer une application mobile. Ce projet vous permettra de voir comment tout cela fonctionne ensemble.

## 11.2 Analogies avec les APIs REST

Pour ceux qui n'ont jamais utilisé d'APIs REST et qui se sentent perdus par rapport à ce concept au début du cours, nous avons pensé à vous. Imaginez que vous êtes dans un restaurant. Avec une API REST, c'est comme si vous deviez commander chaque plat séparément auprès de différents serveurs. Par exemple, vous passez votre commande pour l'entrée à un serveur, le plat principal à un autre, et le dessert à un troisième. Cela peut devenir fastidieux et inefficace.

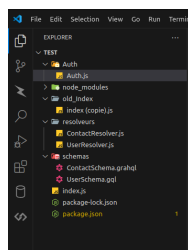
En revanche, avec GraphQL, c'est comme si vous aviez un seul serveur qui prend votre commande complète en une seule fois. Vous pouvez spécifier exactement ce que vous souhaitez, sans avoir à passer par plusieurs serveurs. C'est une approche plus flexible et plus efficace.

## 11.3 Restructuration du projet 📁

Nous allons maintenant restructurer notre projet pour l'application de gestion des contacts. Suivez les étapes ci-dessous :

Pour organiser proprement notre projet, reproduisez minutieusement l'architecture suivante. Les autres fichiers créés dans les dossiers **resolvers**, **schemas**, et **auth** sont vides. En revanche, pour le dossier **old\_index**, il suffira de faire une copie de notre fichier `index` et de le déplacer dans ce dossier afin de sauvegarder les quelques tests effectués avant la refonte du projet.

Ensuite, revenez dans le fichier **index.js** se trouvant à la racine du projet et videz-le. Nous allons maintenant écrire un code **propre** et bien **structuré**. Voici ce à quoi doit ressembler la nouvelle structure :



### 11.3.1 Étape 3 : Configurer Prisma

Initialisez Prisma dans votre projet :

```
1 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 :

```
1 datasource db {
2   provider = "sqlite"
3   url = "file:./dev.db"
4 }
5
6 generator client {
7   provider = "prisma-client-js"
8 }
9
10 model User {
11   id          Int          @id @default(autoincrement())
12   name        String       @db.VarChar(100) @default("")
13   email       String       @unique @db.VarChar(255)
14   password    String       @db.VarChar(255)
15   contacts    Contact[]
16   createdAt   DateTime     @default(now())
17   updatedAt   DateTime     @updatedAt
18 }
19
20 model Contact {
21   id          Int          @id @default(autoincrement())
22   phone       String       @db.VarChar(15)
23   address     String       @db.VarChar(255)
24   userId      Int
25   user        User         @relation(fields: [userId], references: [id])
26   createdAt   DateTime     @default(now())
27   updatedAt   DateTime     @updatedAt
28 }
```

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

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

## 11.4 Structure du projet

Voici la structure du projet `gestion-contacts` :

```
1 gestion-contacts/
2   prisma/
3     schema.prisma
4   src/
5     schema.js
6     resolvers.js
7     index.js
8   package.json
9   document.tex
```

## 11.5 Fichier schema.js

Voici le contenu du fichier `schema.js` :

```
1 // src/schema.js
2 import { gql } from 'graphql-tag';
3
4 const typeDefs = gql`
5   type User {
6     id: Int!
7     name: String!
8     email: String!
9     password: String @deprecated(reason: "Not exposed in API")
10    contacts: [Contact!]!
11    createdAt: String!
12    updatedAt: String!
13  }
14
15  type Contact {
16    id: Int!
17    phone: String!
18    address: String!
19    user: User!
20    createdAt: String!
21    updatedAt: String!
22  }
23
24  type AuthPayload {
25    token: String!
26    user: User!
27  }
28
29  type Query {
30    users: [User!]!
31    user(id: Int!): User
32    contacts: [Contact!]!
33    contact(id: Int!): Contact
34    hello: String
35  }
36
37  type Mutation {
38    createUser(name: String!, email: String!): User!
39    createContact(phone: String!, address: String!, userId: Int!):
40    Contact!
41    updateUser(id: Int!, name: String, email: String): User
42    updateContact(id: Int!, phone: String, address: String):
43    Contact
44    deleteUser(id: Int!): Boolean
45    deleteContact(id: Int!): Boolean
46    signup(name: String!, email: String!, password: String!):
47    AuthPayload!
48    login(email: String!, password: String!): AuthPayload!
49  }
50 `;
51
52 export default typeDefs;
```

## 11.6 Fichier resolvers.js

Voici le contenu du fichier `resolvers.js` :

```
1 // src/resolvers.js
2 import { PrismaClient } from '@prisma/client';
3 import { signup, login } from './auth.js';
4
5 const prisma = new PrismaClient();
6
7 const resolvers = {
8   Query: {
9     hello: () => 'Hello, World!',
10    users: async () => await prisma.user.findMany({ include: {
11      contacts: true } }),
12    user: async (_, { id }) => {
13      const user = await prisma.user.findUnique({ where: { id },
14        include: { contacts: true } });
15      if (!user) throw new Error('User not found');
16      return user;
17    },
18    contacts: async () => await prisma.contact.findMany({ include: {
19      user: true } }),
20    contact: async (_, { id }) => {
21      const contact = await prisma.contact.findUnique({ where: { id },
22        include: { user: true } });
23      if (!contact) throw new Error('Contact not found');
24      return contact;
25    },
26  },
27  Mutation: {
28    createUser: async (_, { name, email }) => {
29      return await prisma.user.create({ data: { name, email } });
30    },
31    createContact: async (_, { phone, address }, { user }) => {
32      if (!user) throw new Error('Not authenticated');
33      return await prisma.contact.create({
34        data: { phone, address, userId: user.userId },
35      });
36    },
37    updateUser: async (_, { id, name, email }) => {
38      const userExists = await prisma.user.findUnique({ where: { id } });
39      if (!userExists) throw new Error('User not found');
40      return prisma.user.update({ where: { id }, data: { name,
41        email } });
42    },
43    updateContact: async (_, { id, phone, address }) => {
44      const contactExists = await prisma.contact.findUnique({ where: { id } });
45      if (!contactExists) throw new Error('Contact not found');
46      return prisma.contact.update({ where: { id }, data: { phone,
47        address } });
48    },
49    deleteUser: async (_, { id }) => {
50      const userExists = await prisma.user.findUnique({ where: { id } });
51      if (!userExists) throw new Error('User not found');
```



```

46     await prisma.user.delete({ where: { id } });
47     return true;
48   },
49   deleteContact: async (_, { id }) => {
50     const contactExists = await prisma.contact.findUnique({ where
51       : { id } });
52     if (!contactExists) throw new Error('Contact not found');
53     await prisma.contact.delete({ where: { id } });
54     return true;
55   },
56   signup,
57   login,
58 };
59
60 export default resolvers;

```

## 11.7 Fichier index.js

Voici le contenu du fichier index.js :

```

1  // src/index.js
2  import { ApolloServer } from '@apollo/server';
3  import { startStandaloneServer } from '@apollo/server/standalone';
4  import dotenv from 'dotenv';
5  import typeDefs from './schema.js';
6  import resolvers from './resolvers.js';
7  import jwt from 'jsonwebtoken';
8
9  dotenv.config();
10
11  const PORT = process.env.PORT || 4000;
12
13  const server = new ApolloServer({
14    typeDefs,
15    resolvers,
16    context: async ({ req }) => {
17      const authHeader = req.headers.authorization || '';
18      const token = authHeader.replace('Bearer ', '');
19
20      if (!token) {
21        console.log('Aucun token re u');
22        return { userId: null };
23      }
24
25      try {
26        const decoded = jwt.verify(token, process.env.JWT_SECRET);
27        console.log('Token d cod :', decoded);
28        return { userId: decoded.userId };
29      } catch (error) {
30        console.error('Token invalide :', error.message);
31        return { userId: null };
32      }
33    },
34  });
35

```

```

36 // D marre le serveur autonome
37 startStandaloneServer(server, {
38   listen: { port: PORT },
39 })
40 .then(({ url }) => {
41   console.log('Server ready at ${url}');
42 })
43 .catch((err) => {
44   console.error('    Error starting server:', err);
45 });

```

### 11.7.1 Étape 4 : Configurer Nodemon

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

```

1 "scripts": {
2   "test": "echo \"Error: no test specified\" && exit 1",
3   "dev": "nodemon src/index.js"
4 },

```

Maintenant, vous pouvez démarrer votre serveur avec :

```

1 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 inquiétez pas par rapport aux volumes des codes des fichiers se trouvant dans le dossier src, nous allons expliquer tout cela pas à pas prochainement.

## 12 Conclusion

Félicitations! Vous avez maintenant un projet GraphQL fonctionnel avec Prisma et Nodemon. Dans les prochaines étapes, nous allons explorer l'intégration avec Flutter pour créer une application mobile. Restez à l'écoute pour la suite! 😊