Qu'est-ce que GraphQL ?

GraphQL est un langage de requête pour API et un environnement d'exécution créé par Facebook en 2012, rendu open-source en 2015. Contrairement à REST, GraphQL permet aux clients de demander exactement les données dont ils ont besoin.

Toutes les requêtes passent par une seule URL, contrairement aux multiples endpoints REST.

Cas d'usage adaptés :

- Applications mobiles avec contraintes de bande passante
- Interfaces complexes nécessitant des données variées
- APIs publiques servant divers clients aux besoins différents
- Agrégation de données provenant de multiples sources

Principes fondamentaux

Langage de requête déclaratif
Le client spécifie la structure exacte des données qu'il souhaite recevoir.

```graphql
{
  user(id: "123") {
    name
    email
    posts {
      title
      comments {
        content
      }
    }
  }
}
```

Schéma fortement typé

Tout est défini par un système de types qui sert de contrat entre client et serveur.

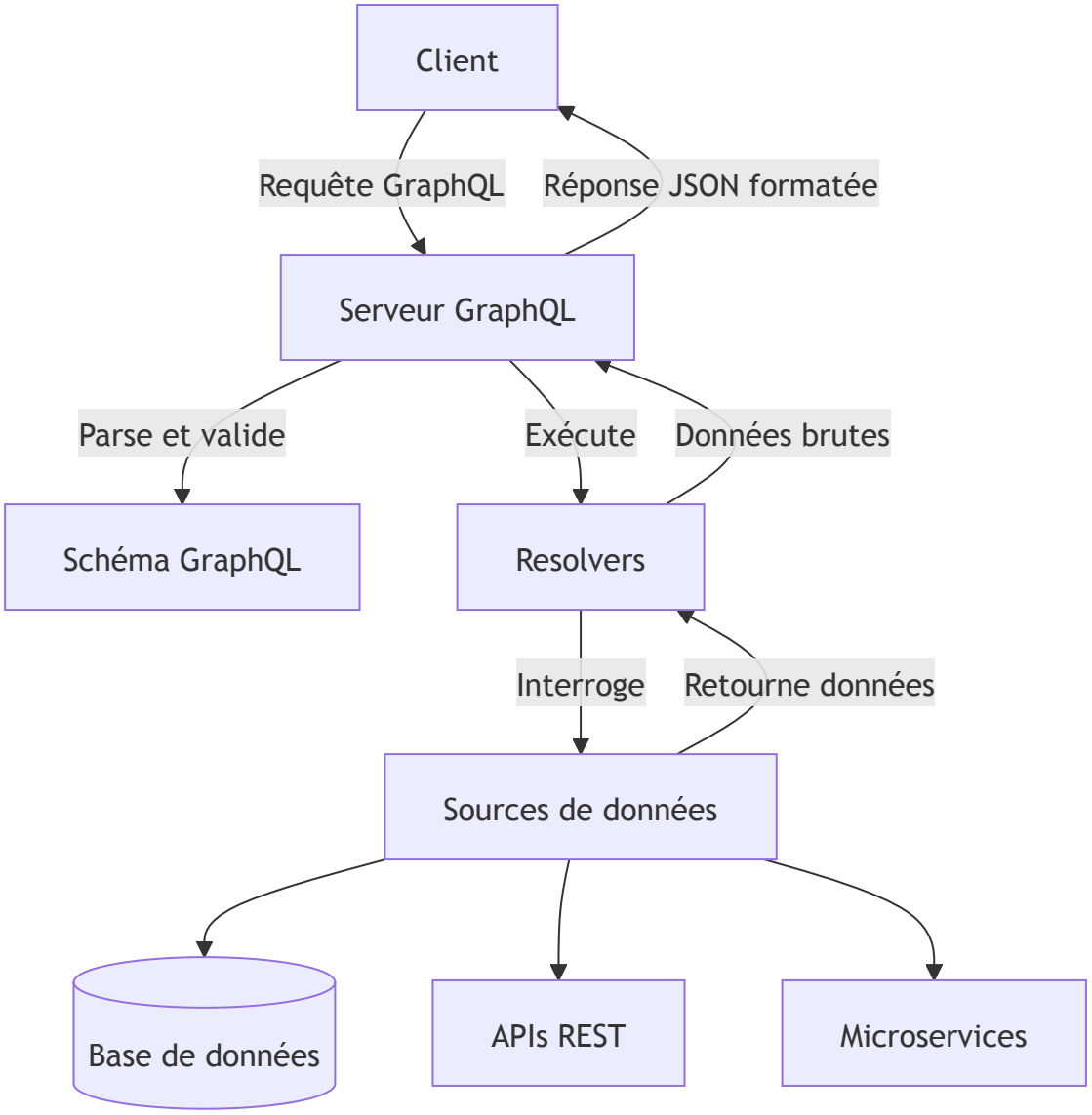
```graphql
type User {
  id: ID!
  name: String!
  email: String
  posts: [Post!]
}

type Post {
  id: ID!
  title: String!
  content: String!
  comments: [Comment!]
}
```

Architecture GraphQL

L'architecture GraphQL repose sur trois concepts clés :

1. **Schéma** : Définit les types de données et les opérations possibles
2. **Resolvers** : Fonctions qui déterminent comment obtenir les données pour chaque champ
3. **Moteur d'exécution** : Analyse les requêtes et orchestre l'exécution des resolvers



Le flux de traitement d'une requête GraphQL :

1. Le client envoie une requête au serveur
2. Le serveur valide la requête contre le schéma
3. Pour chaque champ demandé, le resolver correspondant est appelé
4. Les données sont assemblées selon la structure demandée
5. Le serveur renvoie une réponse JSON correspondant exactement à la requête

Cette architecture offre flexibilité et efficacité tout en maintenant une séparation claire des responsabilités.

Le Schéma GraphQL

https://www.apollographql.com/docs/apollo-server/schema/schema

---

Système de types

Le système de types de GraphQL est la pierre angulaire de son architecture. Il définit un contrat clair entre le client et le serveur.

Types scalaires intégrés :

Les types scalaires sont utilisés pour définir les types de données primaires.

- `Int` : Entier 32 bits signé
- `Float` : Nombre à virgule flottante
- `String` : Chaîne de caractères UTF-8
- `Boolean` : Valeur booléenne (true/false)
- `ID` : Identifiant unique, souvent utilisé comme clé primaire

Types objets :

Les types objets sont utilisés pour définir les entités de l'API.

```graphql
type Product {
  id: ID!
  name: String!
  price: Float!
  description: String
  category: Category
}
```

Interfaces :

Les interfaces sont utilisées pour définir un contrat commun entre différents types objets.

```graphql
interface Node {
  id: ID!
}

type User implements Node {
  id: ID!
  name: String!
}

type Product implements Node {
  id: ID!
  name: String!
  price: Float!
}
```

Types énumération :

Les types énumération sont utilisés pour définir des ensembles de valeurs possibles d'un type.

```graphql
enum OrderStatus {
  PENDING
  PROCESSING
  SHIPPED
  DELIVERED
  CANCELLED
}
```

Types union :

Les types union sont utilisés pour définir un ensemble de types possibles.

```graphql
union SearchResult = User | Product | Article
```

Types d'entrée :

```graphql
input ProductInput {
  name: String!
  price: Float!
  description: String
  categoryId: ID!
}
```

Modificateurs de types :

- `Type!` : Non-nullable (valeur obligatoire)
- `[Type]` : Liste
- `[Type!]!` : Liste non-nullable contenant des éléments non-nullables

---

Définition d'un schéma

Un schéma GraphQL définit les capacités de l'API à travers trois types d'opérations principales :

Type Query : Point d'entrée pour les opérations de lecture.

```graphql
type Query {
  product(id: ID!): Product
  products(category: ID, limit: Int = 10): [Product!]!
  categories: [Category!]!
}
```

Type Mutation : Point d'entrée pour les opérations de modification.

```graphql
type Mutation {
  createProduct(input: ProductInput!): Product!
  updateProduct(id: ID!, input: ProductInput!): Product!
```

```
  deleteProduct(id: ID!): Boolean!
}
```

Type Subscription : Point d'entrée pour les opérations en temps réel.

```graphql
type Subscription {
  productUpdated: Product!
  newOrder: Order!
}
```

Documentation intégrée :

```graphql
"""
Représente un utilisateur du système.
"""
type User {
  """
  Identifiant unique de l'utilisateur.
  """
  id: ID!

  "Nom complet de l'utilisateur."
  name: String!
}
```

Requêtes et Mutations

Structure des requêtes

Les requêtes GraphQL permettent au client de spécifier précisément les données qu'il souhaite obtenir.

Sélection de champs basique :

```graphql
query GetUsers {
  users {
    id
    name
    email
    products {
      id
      name
      price
    }
  }
}
```

> Remarque le nom de la requête est arbitraire et n'a pas d'importance.

Arguments : Les arguments permettent de filtrer ou paramétrer les requêtes.

```graphql
query GetProducts {
  products(category: "electronics", limit: 5) {
    id
    name
    price
  }
}
```

Il est possible de demander plusieurs resources dans une seule requête GraphQL.

Les **alias** permettent de renommer les champs dans la réponse ou d'interroger le même champ plusieurs fois.

```graphql
{
  newProducts: products(limit: 5, sort: "createdAt_DESC") {
    id
    name
  }
  popularProducts: products(limit: 5, sort: "sales_DESC") {
    id
    name
  }
}
```

Fragments : Les fragments sont des ensembles réutilisables de champs.

Ils serviront dans les clients pour factoriser les requêtes et eviter la répétition de code dans les differentes requêtes.

```graphql
{
  user(id: "123") {
    ...UserBasicInfo
    orders {
      id
      total
    }
  }
  orders(userId: "123") {
    id
    total
    user {
      ...UserBasicInfo
    }
  }
}
```

```graphql
fragment UserBasicInfo on User {
  id
  name
  email
  avatarUrl
}
```

Variables : Les variables permettent de paramétrer les requêtes de façon dynamique. Elles doivent être déclérées dans la requête.

```graphql
query GetUser($userId: ID!, $limit: Int) {
  user(id: $userId) {
    id
    name
    orders(limit: $limit) {
      id
      total
    }
  }
}

# Variables JSON
{
  "userId": "123",
  "limit": 10
}
```

Directives : Les directives contrôlent l'inclusion conditionnelle de champs.

```graphql
query GetUser($withReviews: Boolean!) {
  user(id: "123") {
    name
     " include si $withReviews est true "
    reviews @include(if: $withReviews) {
      rating
      comment
    }
```

```
    " skip si $withReviews est true "
    address @skip(if: $withReviews) {
      street
      city
    }
  }
}
```

Mutations

Les mutations permettent de modifier les données sur le serveur.

Structure d'une mutation :

```graphql
mutation CreateProduct($input: ProductInput!) {
  createProduct(input: $input) {
    id
    name
    price
  }
}

# Variables
{
  "input": {
    "name": "Smartphone XYZ",
    "price": 699.99,
    "categoryId": "electronics"
  }
}
```

Mutations multiples :

```graphql
mutation ManageCart($addId: ID!, $removeId: ID!) {
  addToCart(productId: $addId) {
    id
```

```
      items {
        product {
          name
        }
        quantity
      }
    }
    removeFromCart(productId: $removeId) {
      id
      totalItems
    }
  }
```

Bonnes pratiques pour les mutations :

- Retourner l'objet modifié
- Utiliser des types d'entrée pour les arguments complexes
- Structurer les réponses pour inclure les statuts et erreurs
- Nommer les mutations avec des verbes (create, update, delete)
- Assurer l'atomicité des opérations

Erreurs

Lors d'une mutation, si on souhaite retourner des erreurs (validation, permissions, ...) on évitera de rejeter la requete comme on le ferait en REST avec une erreur HTTP 400.

Au lieu de ça, on essaiera de toujours retourner des 200 et plutôt retourner des informations de succès ou d'erreur systematiquement dans la réponse.

graphql

```
mutation CreateOrder($order: OrderInput!) {
  createOrder(order: $order) {
    success
    error {
      code
      message
```

```
    }
    validationErrors {
      field
      message
    }
    order {
      id
      name
    }
  }
}
```

Implémentation côté serveur

Serveurs GraphQL

Plusieurs frameworks permettent d'implémenter un serveur GraphQL, chacun avec ses spécificités, mais ils sont tous basés sur le même principe.

La librairie la plus populaire est Apollo, et c'est elle que l'on étudiera ici. Elle propose un serveur GraphQL complet et une librairie de client GraphQL.

https://www.apollographql.com/docs/apollo-server

typescript

```typescript
// Exemple de code pour un serveur Apollo Server avec express
import { ApolloServer } from "@apollo/server";
import { expressMiddleware } from "@apollo/server/express4";
import express from "express";

const app = express();
const database = new Database();

const server = new ApolloServer<MyContext>({
  typeDefs,
  resolvers,
  context: ({ req }) => ({
    user: req.user,
    dataSources: {
      database,
    },
  }),
});
await server.start();

app.use("/graphql", express.json(), expressMiddleware(server));
```

Context

Le contexte est un objet partagé entre tous les resolvers pour la requête en cours. Il peut contenir des informations comme l'utilisateur authentifié, les données récupérées depuis les headers de la requête, les sources de données, etc.

https://www.apollographql.com/docs/apollo-server/data/context

Certains des éléments qui sont présents dans le contexte pourront être communs à toutes les requêtes (connection à la base de donnée) ou bien spécifiques à une requête (authentification, etc).

typescript

```typescript
type MyContext = {
  user: User;
  dataSources: {
    database: Database;
  };
};
```

Typedefs

Les typedefs sont le schéma GraphQL compilé. Ils définissent les types de données, les champs, les relations entre les types, etc.

typescript

```typescript
import { gql } from "graphql-tag";

const typeDefs = gql`
  type Query {
    user(id: ID!): User
    products: [Product!]
  }

  type User {
```

```
    id: ID!
    name: String!
  }
`;
```

En général on récupérera automatiquement tous les fichiers `*.graphql` dans le projet et on les assemblera en un seul `typeDefs` pour le serveur.

Resolvers

Les resolvers sont des fonctions qui déterminent comment obtenir les données pour chaque champ du schéma.

https://www.apollographql.com/docs/apollo-server/data/resolvers

typescript

```typescript
const resolvers = {
  Query: {
    user: (parent, args, context, info) => {
      return context.dataSources.database.getUser(args.id);
    },
    products: (_, __, context, info) => {
      return context.dataSources.database.getProducts();
    },
  },

  User: {
    orders: (parent, args, context) => {
      // parent contient les données de l'utilisateur déjà résolues
      return context.dataSources.database.getOrdersByUser(parent.id)
    },
  },
  Mutation: {
    createOrder: (parent, args, context) => {
      if (!context.user) {
        throw new GraphQLError(
          "You must be logged in to perform this action.",
          {
```

```
            extensions: {
              code: "FORBIDDEN",
            },
          }
        );
      }
      return context.dataSources.database.createOrder(args, context.
    },
  },
};
```

Paramètres des resolvers :

- `parent` : Résultat du resolver parent (objet contenant)
- `args` : Arguments passés au champ dans la requête
- `context` : Objet partagé entre tous les resolvers (authentification, sources de données) pour la requête en cours
- `info` : Informations sur l'exécution de la requête (rarement utilisé directement)

Lorsque l'on n'utilise pas certains arguments, on peut les ignorer avec `_` :

Sandbox

Les serveur GraphQL sont generalement fournis avec une sandbox pour tester les requetes, qui est une interface web permettant de consulter le schema et executer des requetes.

Generalement accessible à l'url http://localhost:4000/

- Apollo Sandbox
- GraphiQL

Codegen

https://www.apollographql.com/docs/apollo-server/workflow/generate-types

Lorsque l'on développe en typescript, il est necessaire d'avoir accès aux types définis par le schema GraphQL.

Pour cela, il existe des librairies de codegen qui permettent de generer automatiquement du code à partir du schema GraphQL, notamment les types Typescript.

```ts
import type { CodegenConfig } from "@graphql-codegen/cli";

const config: CodegenConfig = {
  schema: "./src/**/*.graphql",
  generates: {
    "./src/types.ts": {
      plugins: ["typescript", "typescript-resolvers"],
      config: {
        contextType: "./index#ResolversContext",
        mappers: {
          User: "./datasource#DBUser",
          Song: "./datasource#DBSong",
        },
      },
    },
  },
};

export default config;
```

Integrations

Il est possible d'integrer le serveur GraphQL dans une application existante, comme une application Next.js.

https://www.apollographql.com/docs/apollo-server/integrations/integration-index

Clients GraphQL

Il existe plusieurs clients GraphQL pour différents langages de programmation.

Le plus populaire est Apollo Client, qui permet de faire des requêtes et des mutations, et possède une integration native avec React.

https://www.apollographql.com/docs/react

Autre clients:

- Relay
- URQL

Apollo Client

Apollo Client est une bibliothèque complète de gestion d'état côté client qui simplifie l'interaction avec un serveur GraphQL.

Configuration de base :

```javascript
import { ApolloClient, InMemoryCache, HttpLink } from "@apollo/clien

const client = new ApolloClient({
  link: new HttpLink({
    uri: "https://api.example.com/graphql",
    headers: {
      authorization: localStorage.getItem("token")
        ? `Bearer ${localStorage.getItem("token")}`
        : "",
    },
  }),
  cache: new InMemoryCache(),
});

// Dans une application React
```

```
import { ApolloProvider } from "@apollo/client";

function App({ children }) {
  return (
    <ApolloProvider client={client}>
      <div className="App">{children}</div>
    </ApolloProvider>
  );
}
```

Queries

https://www.apollographql.com/docs/react/data/queries

Apollo client propose des **hooks** pour react et permet de faire des requêtes avec `useQuery`.

javascript

```
import { useQuery, gql } from "@apollo/client";

const GET_PRODUCTS = gql`
  query GetProducts($category: ID) {
    products(category: $category) {
      id
      name
      price
      imageUrl
    }
  }
`;

function ProductList({ categoryId }) {
  const { loading, error, data, refetch } = useQuery(GET_PRODUCTS, {
    variables: { category: categoryId },
    skip: !categoryId, // Ne pas exécuter si categoryId est absent
    pollInterval: 60000, // Rafraîchir toutes les 60 secondes
  });
```

```
  if (loading) return <p>Chargement...</p>;
  if (error) return <p>Erreur : {error.message}</p>;

  return (
    <div>
      <div className="product-grid">
        {data.products.map((product) => (
          <div key={product.id} className="product-card">
            <img src={product.imageUrl} alt={product.name} />
            <h3>{product.name}</h3>
            <p>{product.price}€</p>
          </div>
        ))}
      </div>
      <div>
        <button onClick={() => refetch()}>Rafraîchir</button>
      </div>
    </div>
  );
}
```

Mutations

https://www.apollographql.com/docs/react/data/mutations

Apollo client permet de faire des mutations avec `useMutation`.

javascript

```
import { useMutation, gql } from "@apollo/client";

const CREATE_PRODUCT = gql`
  mutation CreateProduct($product: CreateProductInput!) {
    createProduct(input: $product) {
      id
    }
  }
`;
```

```
function AddToCartButton({ productId }) {
  const [createProduct, { loading, error }] = useMutation(CREATE_PRO

  return (
    <div>
      <button
        onClick={() =>
          createProduct({ variables: { product: { name: "Produit 1"
        }
        disabled={loading}
      >
        {loading ? "Creation en cours..." : "Créer un produit"}
      </button>
      {error && <p className="error">Erreur: {error.message}</p>}
    </div>
  );
}
```

Codegen

La librairie codegen sera également très utile dans le client, car elle permettra de connaitre les types des données renvoyées par le serveur. Elle permettra également de vérifier que les requêtes utilisées sont correctes et correspondent bien au schema sur serveur.

Côté client, il lui faudra se connecter au serveur GraphQL pour analyser son schema et générer les types.

- https://www.apollographql.com/docs/react/development-testing/static-typing

---

Server-side rendering

Afin d'améliorer les performances des frontend, il est recommandé de faire du SSR (Server-side rendering) côté serveur, avec l'aide de frameworks comme

Next.js. Cela permettra aux utilisateurs de recevoir des pages entièrement chargées des données dès leur arrivée sur leur navigateur. Avec juste du CSR (Client-side rendering), le navigateur devra attendre que l'application soit chargé et que les données soient récupérées avant de pouvoir afficher la page.

- https://www.apollographql.com/docs/react/performance/server-side-rendering
- https://nearform.com/open-source/urql/docs/advanced/server-side-rendering/

Caching

Les clients GraphQL possèdent généralement un cache integré, qui permet de dédupliquer les requêtes, qui peuvent être lancées par differents composants en même temps. Celui-ci aggregera differentes requetes du meme type et effectura des requetes groupées, puis redistribuera les données aux composants qui en ont besoin. Si les données sont déjà presentes dans le cache, elles ne seront pas récupérées sur le serveur.

Pour avoir des données affichées à jour, il faudra alors bien configurer ce cache et lui indiquer quand il doit être invalidé. Lors de mutations, il est possible de mettre à jour les données en cache directement en utilisant la réponse de la mutation, sans avoir besoin d'effectuer une nouvelle requete pour récupérer les données à jour.

- https://www.apollographql.com/docs/react/caching/overview

Optimistic UI

- https://www.apollographql.com/docs/react/performance/optimistic-ui

Optimisations et Bonnes Pratiques

Problème N+1 queries

Le problème N+1 est l'un des défis les plus courants dans les API GraphQL. Il survient lorsqu'une requête récupère une liste d'éléments, puis effectue une requête supplémentaire pour chaque élément afin d'obtenir des données associées.

https://shopify.engineering/solving-the-n-1-problem-for-graphql-through-batching

Exemple problématique :

```javascript
const resolvers = {
  Query: {
    posts: async () => {
      // 1 requête pour obtenir tous les posts
      return await Post.find();
    },
  },
  Post: {
    author: async (post) => {
      // N requêtes, une pour chaque post
      return await User.findById(post.authorId);
    },
  },
};
```

Ici, on récupère tous les posts, et pour chaque post, on récupère l'auteur. Si plusieurs posts ont le même auteur, on fait plusieurs requêtes à la base de données.

DataLoader et batching :

DataLoader est une bibliothèque qui permet de regrouper et mettre en cache les requêtes.

Elle est compatible avec n'importe quel méthode pour récupérer les données car c'est à nous d'implémenter la logique de récupération. Le loader prend en entrée un tableau d'identifiants et retourne un tableau de résultats dans le même ordre que les identifiants.

On créera un loader pour chaque requete dans le contexte, et celui-ci évitera des duplications de requetes.

```javascript
const DataLoader = require("dataloader");

// Dans le contexte de l'application
const createContext = () => {
  const userLoader = new DataLoader(async (userIds) => {
    // Une seule requête pour récupérer tous les utilisateurs nécess
    const users = await User.find({ _id: { $in: userIds } });

    // Réorganiser les résultats dans le même ordre que les IDs dema
    return userIds.map((id) =>
      users.find((user) => user.id.toString() === id.toString())
    );
  });

  return { userLoader };
};

// Dans les resolvers
const resolvers = {
  Post: {
    author: async (post, _, { userLoader }) => {
      return await userLoader.load(post.authorId);
    },
  },
};
```

Caching

https://www.apollographql.com/docs/apollo-server/performance/caching

Apollo Server propose plusieurs stratégies de cache :

```javascript
const { ApolloServer } = require("apollo-server-express");
const { InMemoryLRUCache } = require("apollo-server-caching");

const server = new ApolloServer({
  typeDefs,
  resolvers,
  cache: new InMemoryLRUCache({
    maxSize: 1000000, // ~1MB
    ttl: 300, // 5 minutes
  }),
  dataSources: () => ({
    productsAPI: new ProductsAPI(),
  }),
});

// Dans une source de données personnalisée
class ProductsAPI extends RESTDataSource {
  constructor() {
    super();
    this.baseURL = "https://api.example.com/";
  }

  async getProduct(id) {
    return this.get(`products/${id}`, null, {
      cacheOptions: { ttl: 60 }, // Cache pour 60 secondes
    });
  }
}
```

Pagination

Lorsqu'une requete récupère une liste d'éléments, il est souvent nécessaire de la paginer. En effet, sans la pagination, si la base de données contient beaucoup d'éléments, la requete peut prendre être très longue à charger, très volumineuse et ralentir le client et le serveur voir même generer des erreurs.

```graphql
input PaginationInput {
  page: Int = 1
  pageSize: Int = 10
}

type Query {
  products(pagination: PaginationInput): ProductConnection!
}
```

> Attention, la technique limit/offset peut souvent poser problème (changement de données entre les requetes) et on preferera utiliser des curseurs !

Implémentation des resolvers :

```javascript
const resolvers = {
  Query: {
    products: async (_, { pagination }) => {
      return Product.find()
        .sort({ _id: 1 })
        .skip(pagination.page * pagination.pageSize)
        .limit(pagination.pageSize);
    },
  },
};
```

Cursor-based

La pagination par limit/offset peut souvent poser problème (changement de données entre les requetes, perte/duplication de données) et on preferera utiliser des curseurs.

La pagination par curseur, au lieu d'etre définie par un nombre d'elements à retourner et le numéro de la page, se basera sur le dernier element récupéré pour récupérer la page suivante.

```graphql
input PaginationInput {
  cursor: String
  limit: Int = 10
}
```

La pratique commune est de suivre la specification Relay Cursor Connections Specification avec des edges/node pour retourner les données.

```graphql
type ProductConnection {
  edges: [ProductEdge!]!
  pageInfo: PageInfo!
}

type ProductEdge {
  node: Product!
  cursor: String!
}

type PageInfo {
  hasNextPage: Boolean!
  hasPreviousPage: Boolean!
  startCursor: String
  endCursor: String
}
```

Le type de curseur dépendra de l'implémentation en base de données. En postgres, on remplacera les clauses LIMIT/OFFSET par des clauses ORDER BY et des clauses WHERE avec des curseurs.

```sql
SELECT * FROM products
ORDER BY id
```

```
LIMIT 10
OFFSET 20;
```

Remplacé par:

```sql
SELECT * FROM products
WHERE id > '20'
ORDER BY id
LIMIT 10;
```

Sécurité

Validation des entrées

Le serveur Apollo valide le schema GraphQL, mais il ne verifie pas les données envoyées en entrées par le client, il faudra le faire soi-même, avec une librairie telle que zod

```typescript
import { UserInputError } from "apollo-server-express";
import { z } from "zod";

const productInputSchema = z.object({
  name: z.string().min(3).max(100),
  price: z.number().positive(),
  description: z.string().max(1000),
  categoryId: z.string(),
});

const resolvers = {
  Mutation: {
    createProduct: async (_, { input }) => {
      try {
        // Valider l'entrée
        await productInputSchema.parseAsync(input);
```

```
        // Procéder à la création
        return await Product.create(input);
      } catch (error) {
        throw new GraphQLError(error.message, {
          extensions: { code: "BAD_USER_INPUT" },
        });
      }
    },
  },
};
```

Projet d'exemple

Un projet de démonstration illustrant les principales fonctionnalités de GraphQL est disponible dans le dépôt GitHub graphql.

Celui-ci servira de base pour les exercices.

Structure du projet

Cette application de démonstration implémente une bibliothèque de musique avec des utilisateurs et les chansons qu'ils ont publiés.

Serveur

Le serveur est construit avec Apollo Server.

Il propose un schema GraphQL basique avec generation automatique des types avec codegen.

Afin de simplifier le projet, aucune base de données n'est utilisée, et les données sont stockées en mémoire et accessibles depuis la classe `FakeORM` qui simule un ORM basique d'une base de données.

Client

Le client est construit avec Next.js et shadcn/ui.

Il propose des pages pour afficher les utilisateurs et les chansons.

Developpement

L'application complete est lançable avec Docker depuis la racine du projet avec la commande `docker compose up`.

> Attention, lancé avec Docker, l'application sera lancée en mode "production" et ne sera pas mise à jour avec les derniers changements du code.

Le frontend sera accessible à l'adresse `http://localhost:3000`.

La sandbox Apollo sera accessible à l'adresse `http://localhost:4000/graphql`.

Lancement en mode developpement

Pour lancer l'application en mode developpement, installer les dépendances de chaque projet avec `npm install` et lancer les chacun d'eux avec `npm run dev`.

Exercices GraphQL

Ces exercices seront à faire par dessus le code existant de démonstration.

Vous devrez forker le repo de répart et créer une pull request sur celui ci une fois votre travail fini.

https://github.com/opac-teach/graphql

> Si vous le souhaitez, vous pouvez commencer par suivre le guide d'introduction officiel pour vous familiariser avec le framework.

Backend

Ces exercices sont à faire dans le dossier `server-apollo`.

Queries

Pour tester les requêtes, vous pouvez utiliser le playground que vous trouverez sur l'url: http://localhost:4000/

- Creer une query qui renvoie tous les utilisateurs
- Creer une query qui renvoie tous les utilisateurs avec leurs chansons
- Creer une query qui renvoie un utilisateur par son id et les chansons qu'il a créé
- Creer une mutation qui crée un utilisateur

Songs

Song User

- Ajouter un champ user dans le schema Song qui renverra l'utilisateur lié à la chanson et modifier le resolver.

Song by ID

- Rajouter un resolver `song(id)` qui renverra une chanson par son id.

Genre

- Ajouter un modèle `Genre` qui regroupera les chansons du meme genre.
- Creer le schema, les données mockées, les resolvers, mutations, les relations, ...
- Ajouter un argument optionnel à la query `songs` pour filtrer par genre

---

Songs Count

- Ajouter un champ `songsCount` dans les schema `User` et `Genre` qui renverra le nombre de chansons liées à l'utilisateur ou au genre.

Mutations

- Creer une mutation pour créer une chanson. N'autoriser que les utilisateurs connectés à effectuer cette action
- Creer des mutation qui permettront de modifier ou supprimer un utilisateur ou une chanson. Prendre en compte l'utilisateur qui effecture l'action et ne l'autoriser à modifier ou supprimer que ses propres données.

Pagination

- Ajouter un système de pagination sur la liste des utilisateurs et des chansons, qui permettra de limiter le nombre de données renvoyées par la requête.
- Ajouter également la pagination sur les chaines de resolvers.

Loaders

- Creer une query qui recuperera toutes les chansons, et l'utilisateur associé à chacune d'elle.
- Observer les messages de log dans la console et repérer les appels à la base de données qui sont dupliqués.
- Utiliser les loaders pour optimiser les requêtes et régler le probleme N+1.

[Bonus]

Rôles

Ajouter un système de rôles, et n'autoriser que les utilisateurs avec le role "ADMIN" à utiliser les mutations sur les genres.

> Vous pourrez au choix utiliser un header "role" dans la requete HTTP pour simuler un utilisateur avec un rôle spécifique ou bien passer sur une vraie authentification avec du JWT par exemple.

Subscription

Modifier le serveur pour qu'il supporte les subscriptions et signaler aux client lors de l'ajout de nouvelles chansons.

https://www.apollographql.com/docs/apollo-server/data/subscriptions

Pagination

Remplacer la pagination limit/offset par une pagination par curseur. Vous devrez certainement modifier le FakeORM ou bien utiliser une vraie base de données.

Tests

Creer des tests d'integration pour les resolvers.

Validation des données

Utiliser `zod` pour valider les entrées des mutations.

Frontend

Ces exercices sont à faire dans le dossier `client-next` qui contient une application Next.js.

Songs

- Ajouter la page `/song/[id]` qui affichera les détails d'une chanson
- Afficher les utilisateur qui ont créé les chansons sur les pages `/songs` et `/songs/[id]` avec des liens vers ceux-ci

---

Genres

- Afficher les données sur les genres
  - Creer les pages `/genre` et `/genre/[id]`
  - Sur les autres pages, ajouter les infos de genre quand on affiche des chansons

---

- Afficher les nombre de chansons pour chaque genre et chaque utilisateur sur les pages concernées

---

Formulaires

Ajouter des formulaires pour:

- Ajouter une chanson
- Ajouter un genre
- Au choix: modifier/supprimer des chansons, utilisateurs ou un genres ...

[Bonus]

- Utiliser des fragments dans le frontend pour factoriser les requêtes
- Utiliser la pagination
- Mettre a jour le cache lors des mutations pour éviter les refetch
- Utiliser les subscriptions pour recevoir en temps réel les nouvelles chansons

Next

- Passer sur du rendu côté serveur (SSR) pour toutes les pages
- Integrer le serveur GraphQL directement dans l'application Next.js

NestJS

Creer un nouveau projet NestJS dans le dossier `server-nestjs` et transferer le travail effectué precedement dedans.

Ce nouveau serveur devra:

- Implementer l'API GraphQL à la manière "code-first"
- Stocker les données dans une base de données avec TypeORM

- Valider les entrées des mutations
- Avoir un système d'authentification et proteger les routes qui en ont besoin(par ex. un utilisateur ne doit pouvoir editer que ses propres chansons)
- Implementer la pagination par curseur
- Supporter les subscriptions

> Si vous le souhaitez, au lieu de recréer un nouveau projet, vous pouvez reprendre le projet NestJS créé précedement et rajouter une API GraphQL qui propose les mêmes fonctionnalités que les endpoints REST existants.

[Projet] Bibliothèque musicale

Développer une application qui permettra à ses utilisateurs de consulter leur bibliothèque musicale sur les plateformes de streaming existantes (Spotify, Deezer)

L'application servira de "proxy" aux differentes plateformes et proposera une API GraphQL commune pour l'ensemble des plateformes.

Fonctionnalités

En mode public (utilisateur anonyme)

- Recherche de chansons, artistes, albums
- Détails sur une chanson, artiste, album

En mode privé (utilisateur connecté)

- Connexion des utilisateurs en OAuth sur les plateformes existantes
- Récupération des playlists de l'utilsateur connecté et des chansons qu'elles contiennent

- Ajout/supression de chansons à une playlist
- Creation/modification/suppression de playlists
- Synchronisation de la bibliothèque musicale des utilisateurs entre les plateformes

---

Implémentation

- Backend: Serveur Apollo+Express ou NestJS (recommandé). Base de données Redis ou PostgreSQL selon les besoins.
- Frontend: NextJS (optionnel)

- Nestjs
- tests