GRAPHQL

PARTIE I

Introduction à GraphQL GraphQL est un langage de requête pour les API, conçu pour rendre les interactions entre les clients (comme une application front-end ou mobile) et les serveurs (comme un back-end) plus efficaces, flexibles et intuitives. Développé initialement par Facebook en 2012 et rendu open source en 2015, GraphQL a depuis gagné une popularité croissante grâce à ses nombreux avantages.

1. Pourquoi utiliser GraphQL ?

Problèmes des API REST traditionnelles :

- **Sur-récupération de données :** Une requête REST peut parfois retourner plus de données que nécessaire, ce qui alourdit les performances.
- **Sous-récupération de données :** Parfois, pour obtenir toutes les informations nécessaires, un client doit faire plusieurs appels à différentes ressources.
- **Structure rigide :** Les endpoints REST sont prédéfinis, ce qui limite leur flexibilité pour répondre aux besoins des clients.

GraphQL résout ces problèmes :

- **Flexibilité :** Le client demande exactement les données dont il a besoin, ni plus, ni moins.
- **Une seule requête :** Les données provenant de différentes sources peuvent être récupérées en une seule requête.
- **Évolution facile :** Les modifications apportées au schéma de l'API n'affectent pas les clients existants.

---

2. Concepts clés de GraphQL

1. **Schéma (Schema) :** Le schéma définit la structure des données disponibles sur le serveur, comme les types d'objets et les relations entre eux.

Exemple d'un schéma basique en GraphQL :

```
var schema = buildSchema('
    type Query {
        hello: String
    }'
)
```

buildSchema : Crée un schéma GraphQL à partir d'une chaîne de caractères.

type Query : Définit les types de requêtes disponibles. Ici, Query est l'entrée principale de GraphQL (similaire à une API REST GET).

hello : String : Déclare une requête appelée hello qui retourne une chaîne de caractères (String).

Le schéma décrit donc une API avec une seule requête appelée hello, qui retourne un message de type chaîne de caractères.

2. **Requêtes (Queries) :** Les requêtes permettent de lire des données depuis le serveur.

   Exemple de requête GraphQL :

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

   Résultat attendu :

   ```
   {
       "data": {
           "user": {
               "name": "Alice",
               "email": "alice@example.com"
           }
       }
   }
   ```

3. **Mutations :** Les mutations permettent de modifier ou de créer des données.

   Exemple de mutation :

   ```
   mutation {
       createUser(name: "Bob", email: "bob@example.com") {
           id
           name
   ```

```
        email
    }
}
```

Résultat attendu :

```
{
    "data": {
        "createUser": {
            "id": "2",
            "name": "Bob",
            "email": "bob@example.com"
        }
    }
}
```

4. **Types scalaires :** GraphQL supporte des types de données basiques comme `String`, `Int`, `Float`, `Boolean`, et `ID`.
5. **Resolvers :** Les resolvers sont des fonctions qui déterminent comment les données sont récupérées ou modifiées en fonction de la requête.

3. Exemple pratique avec GraphQL

Schéma complet :

```
type Query {
    user(id: ID!): User
    posts(userId: ID!): [Post]
}

type Mutation {
    createUser(name: String!, email: String!): User
    createPost(userId: ID!, content: String!): Post
}

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

type Post {
    id: ID!
    content: String
    author: User
}
```

Requête complexe : Récupérer un utilisateur avec ses publications :

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

Résultat attendu :

```
{
    "data": {
        "user": {
            "name": "Alice",
            "email": "alice@example.com",
            "posts": [
                { "content": "First post!" },
                { "content": "GraphQL is awesome!" }
            ]
        }
    }
}
```

Mutation pour créer une publication :

```
mutation {
    createPost(userId: "1", content: "Learning GraphQL is fun!") {
        id
        content
        author {
            name
        }
    }
}
```

Résultat attendu :

```
{
    "data": {
        "createPost": {
            "id": "3",
            "content": "Learning GraphQL is fun!",
            "author": {
                "name": "Alice"
```

```
            }
        }
    }
}
```

4. Avantages de GraphQL

1. **Requêtes précises :** Vous pouvez demander uniquement les champs nécessaires.
2. **Documentation intégrée :** Les schémas GraphQL sont auto-documentés.
3. **Pas de versionnage d'API :** Les champs peuvent être ajoutés sans impacter les anciens clients.
4. **Idéal pour les applications modernes :** Très utile pour les applications mobiles ou front-end où la bande passante est limitée.

5. Déploiement de GraphQL

- **Serveurs GraphQL populaires :**
  - **Apollo Server** : Une solution complète pour construire des API GraphQL.
  - **GraphQL Yoga** : Facile à démarrer.
  - **Relay** : Développé par Facebook pour un usage avec React.
- **Exemple de serveur Apollo (Node.js) :**

```
const { ApolloServer, gql } = require('apollo-server');

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

// Résolveurs
const resolvers = {
    Query: {
        hello: () => "Bonjour, GraphQL !"
    }
};

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

// Lancement du serveur
server.listen().then(({ url }) => {
```

```
    console.log(` Serveur prêt à : ${url}`);
});
```

GraphQL est une technologie puissante qui offre flexibilité, efficacité et une meilleure expérience pour les développeurs et les utilisateurs finaux. Avec ses concepts comme les requêtes, les mutations et les resolvers, il permet de construire des API modernes adaptées aux besoins actuels des applications.

PARTIE II

Fonctionnalités avancées de GraphQL

1. **Fragments** :
   - Permettent de réutiliser des blocs de requêtes pour éviter les répétitions.
   - Exemple :

```
fragment UserDetails on User {
    id
    name
    email
}

query {
    user(id: "1") {
        ...UserDetails
        posts {
            content
        }
    }
}
```

2. **Directives** :
   - Ajoutent des fonctionnalités dynamiques aux requêtes comme des conditions.
   - Exemple :

```
query getUser($includeEmail: Boolean!) {
    user(id: "1") {
        name
        email @include(if: $includeEmail)
    }
}
```

3. **Gestion des erreurs** :

- Les réponses GraphQL incluent une section `errors` pour indiquer des problèmes spécifiques.
- Exemple de réponse avec erreur :

```
{
    "data": null,
    "errors": [
        {
            "message": "User not found",
            "locations": [{ "line": 2, "column": 3 }],
            "path": ["user"]
        }
    ]
}
```

4. **Subscriptions** :
   - Permettent une communication en **temps réel** (comme les notifications ou les mises à jour en direct).
   - Exemple de subscription (avec WebSocket) :

```
subscription {
    postAdded {
        id
        content
        author {
            name
        }
    }
}
```

5. **Types d'union et interfaces** :
   - Les **types d'union** permettent de retourner plusieurs types possibles pour un même champ.
   - Les **interfaces** définissent des structures communes pour plusieurs types.
   - Exemple :

```
interface SearchResult {
    id: ID!
}

type User implements SearchResult {
    id: ID!
    name: String
}
```

```
type Post implements SearchResult {
    id: ID!
    content: String
}

type Query {
    search(query: String!): [SearchResult]
}
```

6. **Fédérations de services (GraphQL Federation)** :
   - Permettent de composer plusieurs microservices GraphQL en un seul point d'entrée.
   - Utilisé avec **Apollo Federation** pour orchestrer un schéma unifié.
7. **Pagination et filtrage** :
   - Utilisation de paramètres comme `limit`, `offset` ou des **connections** pour des paginations sophistiquées.
   - Exemple de pagination avec connections (modèle Relay) :

```
query {
    posts(first: 10, after: "cursor123") {
        edges {
            node {
                id
                content
            }
        }
        pageInfo {
            hasNextPage
            endCursor
        }
    }
}
```

8. **Batching et DataLoader** :
   - Optimise les requêtes pour éviter les appels redondants à la base de données ou aux API externes.
9. **Sécurité et authentification** :
   - Implémenter des mécanismes pour limiter les accès aux données :
     - Authentification (JWT, OAuth, etc.).
     - Règles au niveau des resolvers pour gérer les autorisations.
10. **Monitoring et performance** :

- Utilisation d'outils comme **Apollo Studio**, **GraphQL Voyager**, ou des middlewares pour suivre les performances des requêtes.

Quand GraphQL n'est-il pas adapté ?

Malgré ses nombreux avantages, GraphQL peut ne pas convenir dans certains cas :

- Si l'application est simple et ne nécessite pas de flexibilité.
- Lorsque la surcharge en termes de performances ou de complexité d'implémentation n'est pas justifiée.
- Si l'équipe n'a pas suffisamment d'expérience pour bien architecturer un schéma efficace.