GraphQL

M2 MIAGE ISIE - 2019

Emilien ESCALLE - Directeur Technique

Objectifs

Découvrir la technologie GraphQL

Plan

1. Rappel GraphQL
2. Fonctionnement

1. GraphQL : rappel

GraphQL - Définition

Langage de requête pour les API, et environnement d'exécution des requêtes coté serveur, en n'utilisant rien d'autre qu'un système de type défini par soit même. GraphQL n'utilise pas de base de données ou de moteur de stockage de données interne, il fonctionne uniquement via le code et les données.

Un service GraphQL est crée par la définition de types, et par la définitions de fonctions à exécuter sur les champs de ces types.

GraphQL permet :

- De n'obtenir que les informations demandées par la requête (et uniquement celles-ci) ;
- De faire des requêtes imbriquées dans d'autres requêtes (comme obtenir d'une seule traite, les billets et les commentaires qui leurs sont associés) ;
- D'avoir un typage fort des données, là où une API REST n'offre aucune garantie. Vous pouvez savoir avec exactitude que tel paramètre devra être un entier et pas un booléen ;
- D'avoir une documentation "automatique" qui reflète la structure même des données qu'il est possible d'obtenir. GraphQL le prévoit et il y a des outils qui permettent en temps réel de savoir la structure d'un résultat pour choisir ce que l'on voudra afficher ou pas, les relations des entités entre elles, etc.
- De ne pas être dépendant d'une version ou d'une autre. Puisque l'on obtient ce que l'on a demandé… Quand un champ existant n'est plus utilisé, on peut le déprécier pour avertir l'équipe technique, mais l'information reste disponible pour la durée souhaitée.

GraphQL - Exemple : requête imbriquée

Avec GraphQL il n'y a plus qu'un seul endpoint sur lequel les requêtes sont exécutées. Par exemple :

```
{
 post {
   title,
   content,
   date,
   author,
   comments {
     author {
       name,
       avatar
     },
     content
   }
 }
}
```

```
{
 "data" {
   "post": {
     "title": "Grande nouvelle",
     "content": "..."
     "author": "Emma Kant",
     "comments": [
      {
         "author": {
            "name": "Joe",
            "avatar": "x1"
         },
         "content": "..."
      }
     ]
   }
 }
}
```

En une seule requête, il est possible de recevoir toutes les informations structurées comme demandées (et uniquement ces dernières).

Avec REST il aurait fallu faire une requête pour récupérer *l'id* de l'auteur, puis une seconde pour récupérer les *posts* ayant *l'id* de cet auteur, puis pour chaque *post*, d'autres requêtes pour récupérer les bons commentaires

GraphQL - Exemple : requête paginée

Exemple de requête permettant de récupérer les posts part auteur avec une pagination :

```
{
  author {
    name
    posts(first:2) {
      totalCount
      edges {
        node {
          title
        }
        cursor
      }
      pageInfo {
        endCursor
        hasNextPage
      }
    }
  }
}
```

```
{
  "data": {
    "author": {
      "name": "Emma Kant",
      "posts": {
        "totalCount": 3,
        "edges": [
          {
            "node": {
              "title": "Grande Nouvelle"
            },
            "cursor": "Y3Vyc29yMg=="
          },
          {
            "node": {
              "title": "Une autre histoire"
            },
            "cursor": "Y3Vyc29yMw=="
          }
        ],
        "pageInfo": {
          "endCursor": "Y3Vyc29yMw==",
          "hasNextPage": false
        }
      }
    }
  }
}
```

LES TYPES

Le premier concept GraphQL est les types de données qu'il est possible de requêter.

Comme pour une base de données classique il existe des types `Scalar`:

- `Int` un entier
- `Float` un double
- `String` une chaîne de caractères
- `Boolean` un booléen
- `ID` représente un identifiant unique

Le type `Enum` qui est considéré comme un `scalar` permet de valider les valeurs d'un argument.

```
enum Status {
      ONLINE
      ERROR
      REVIEW
}
```

Il est aussi possible de gérer des `list` de `scalar` ou de `type` . Pour cela il suffit de mettre en place des `[]`.

Dans l'ensemble des cas, il est possible de forcer le `non-null` en ajoutant un `!` à la fin du typage.

La création d'un nouveau type se fait via le mot clé `type`.

```
type Article {
  name: String!
  status: Status!
}
```

Comme pour des objets dans un langage orienté objet, il est possible d'utiliser des interfaces via le mot clé `interface`.

```
interface Publish {
  id: ID!
  name: String!
  createdAt: Date
}
```

Et donc l'utiliser dans les types :

```
type Article implements Publish {
  content: String!
}
```

Une fois les types définis, il est nécessaire de définir les types principaux : query & mutation

QUERY

Le type `query` représente l'ensemble des fonctions utilisables pour la récupération des données.

Chaque `query` peut prendre des paramètres et renvoie un `type`.

```
type Query {
  article(id: ID!): Article
  articles(): [Article]
}
```

MUTATION

Comme dans une API rest la mutation permet d'effectuer des changements dans la base de données.

Elle se configure comme la `query` il faut définir la fonction avec ses paramètres et sa sortie. Les paramètres d'une mutation sont de type `input`.

Le type `input` se configure comme un type classique en utilisant le mot clé `input`.

```
input ArticleInput {
  id: Int!
  title: String
}
```

Vous pouvez utiliser vos inputs dans vos mutations : `saveArticle(input: ArticleInput!): Article`

RESOLVER

Un fois les types définis, la déclaration des resolvers permet à GraphQL de connaître la méthode de récupération des données.

Chaque `resolver` est une fonction qui permet d'aller chercher la donnée au bon endroit.

La plus value des serveurs GraphQL est de savoir que la donnée a déjà été chargée et ne pas la recharger.

C'est aussi de permettre via une seule requête GraphQL d'aller chercher dans plusieurs bases.