GraphQL

API orientée requêtes

Principes, avantages et inconvénients

REST :

- Une API REST suit une structuration claire orientée ressources
- Si toutefois la taille des données devient importante les problèmes suivants arrivent :
  - Envoi d'un grand nombre de données, parfois inutiles, sur le réseau
    - "over-fetching"
  - Besoin d'un grand nombre de points d'entrée et de chemins très longs pour définir la ressource
  - Côté client besoin de combiner plusieurs requêtes pour obtenir les données souhaitées ce qui va surcharger aussi le réseau et surtout le nombre de requêtes au serveur
    - "under-fetching"

Principes, avantages et inconvénients

GraphQL :

- GraphQL structure les données sous la forme d'un graphe (d'où son nom) et offre un **langage de requête** pour parcourir les données et les récupérer de façon structurée
- GraphQL est aussi le **moteur** permettant de comprendre et répondre aux requêtes
- Chaque client créée la requête qu'il souhaite et récupère uniquement les données utiles ce qui décharge le réseau
- GraphQL implique une difficulté de mise en cache des données car les requêtes (et donc les réponses) sont toujours différentes

Fonctionnement concret

Client

- HTTP/1.1 POST
- body = GraphQL query
- point d'entrée unique

Serveur

- réception de la requête
- résolution de la requête
- réponse en JSON

Définition de l'API ou du schéma

- On parle aussi de schéma en GraphQL au lieu d'API
- Il s'agit de définir la structure des éléments auxquels on peut accéder par l'API
- Il y a trois types principaux dans un schéma
  - les "queries" qui définissent les requêtes possibles sur le serveur,
  - les "mutations" qui définissent les mises à jour, suppressions ou créations de données,
  - les modèles de données qui permettent de définir des objets représentant les données manipulées par les queries et les mutations.

API/schéma

- On définit des `ObjectType`
  - `Query` et `Mutation` sont des types existants
  - On ajoute les types souhaités
    - Character
    - Planet
    - Species
- Chaque type est constitué de **fields**
  - Les fields peuvent prendre des **arguments** en entrée
  - On peut ajouter un `!` collé à un type pour spécifier que ce field est **obligatoire**
    - “name : String!”
- Les autres types :
  - **scalar** (String, Int, Float etc.)
  - **lists** “[Character]”
  - **enumerations**
  - **interfaces**
  - ...

field : type

```
type Query {
    hero : Character
}

type Character {
    name : String
    friends : [Character]
    homeWorld : Planet
    species : Species
}

type Planet {
    name : String
    climate : String
}

type Species {
    name : String
    lifespan : Int
    origin : Planet
}
```

field(argument : type) : type

```
type Query {
    hero(name: String) : Character
}

type Character {
    name : String
    friends : [Character]
    homeWorld : Planet
    species : Species
}

type Planet {
    name : String
    climate : String
}

type Species {
    name : String
    lifespan : Int
    origin : Planet
}
```

Tous les détails ici !

Résolution (exécution)

- En GraphQL chaque **field** est en fait une **fonction** ou une méthode qui retourne le type suivant à résoudre
  - chaque field peut être associé à une fonction appelée “**resolver**”
    - si le type suivant est un **scalar** la résolution s’arrête et on remonte dans le graphe
    - si le type suivant est un **objet** on appelle le resolver associé
  - il faut au minimum un resolver par query et mutation
  - les resolvers correspondent au **code métier** et sont fournis par le développeur du serveur
  - un resolver prend en entrée l’**objet parent du field et les arguments du field** à résoudre ainsi que des éléments de contexte
- Le point de départ de toute résolution est une requête ou une mutation
  - **root types**

Détails : https://graphql.org/learn/execution/

Langage de requêtes

Le langage de requête GraphQL consiste à construire une structure faite des fields et des objets souhaités

```
{
  hero(name : "Luke Skywalker") {
    name
    friends {
      name
      homeWorld {
        name
        climate
      }
      species {
          name
          lifespan
          origin {
            name
          }
      }
    }
  }
}
```

```
type Query {
    hero(name: String) : Character
}

type Character {
    name : String
    friends : [Character]
    homeWorld : Planet
    species : Species
}

type Planet {
    name : String
    climate : String
}

type Species {
    name : String
    lifespan : Int
    origin : Planet
}
```

Exemple d'exécution d'une requête

schéma

```
type Query {
    hero(name: String) : Character
}

type Character {
    name : String
    friends : [Character]
    homeWorld : Planet
    species : Species
}

type Planet {
    name : String
    climate : String
}

type Species {
    name : String
    lifespan : Int
    origin : Planet
}
```

1 2 3 4 7 5 6 8 9 10

requête

```
{
  hero(name : "Luke Skywalker") {
    name
    friends {
      name
      homeWorld {
        name
        climate
      }
      species {
        name
        lifespan
        origin {
          name
        }
      }
    }
  }
}
```

Object -> resolver *Scalar* *Object -> resolver* *Scalar* *Object -> resolver* *Scalar* *Scalar* *Object -> resolver* *Scalar* *Scalar* *Object -> resolver* *Scalar*

réponse

```
{
  "data": {
    "hero": {
      "name": "Luke Skywalker",
      "friends" : [
        {
          "name" : "R2-D2",
          "homeWorld" : {
            "name" : "X",
            "climate" : "good"
          },
          "species" : {
            "name" : "robot",
            "lifespan" : "eternal",
            "origin" : {
              "name" : "null"
            }
          }
        },
        {
          "name" : "C3PO",
          "homeWorld" : {
            ...
          },
          "species" : {
            ...
          }
        }
      ]
    }
  }
}
```

Tuto

GraphQL

Sujet sur ma page
Lien sur Moodle

Avantages et inconvénients

	Performance API	Debug API	Découverte API	Explosion nb fonctions ou entrées	Flexibilité
REST / OpenAPI					
gRPC					
GraphQL					