

UE Services FISE LOGIN A3 - GraphQL

2022-2023

Hélène Coullon



IMT Atlantique

Bretagne-Pays de la Loire

École Mines-Télécom

Table of Contents

1. Pourquoi GraphQL ?
2. Principes et fonctionnement de GraphQL

Pourquoi GraphQL ?

Avantages et inconvénients de REST

Avantages

- API orientée ressources
- Clarté de l'API : un point d'entrée, une ressource
- Simplicité et portabilité des requêtes HTTP

Inconvénients

- Toutes les données envoyées pour une requête donnée : **over-fetching**
- Besoin de combiner plusieurs requêtes pour obtenir les données : **under-fetching**
- Si on souhaite sous diviser les ressources il faut beaucoup de points d'entrée et l'API devient complexe
- Si on souhaite ajouter une ressource il faut un/des nouveaux points d'entrée

Avantages et inconvénients de GraphQL

Avantages

- Combinaison des principes de REST avec un langage de requête
 - demander et récupérer uniquement les données nécessaires
 - pas de multiplication du nombre de points d'entrée
 - Flexibilité d'ajouter des ressources sans ajouter de point d'entrée

Inconvénients

- Comme chaque requête est différente la mise en cache est plus difficile

Principes et fonctionnement de GraphQL

Client

- HTTP/1.1 méthode POST
- Requête HTTP sur un point d'entrée unique
- body = requête GraphQL

Serveur

- Réception de la requête
- Résolution de la requête
- Envoi de la réponse

Le “Schéma” GraphQL est la définition de l'API ([Détails ici](#))

- équivalent des points d'entrée en REST
- équivalent de fichier protocol buffers en gRPC

Contenu du schéma GraphQL

- *Object types* :
 - types existants racines : *Query*, *Mutation*
 - ajout de nouveaux types
- *Fields* : Le contenu des Object types
 - `field_name(argument:type):type`
 - les arguments sont optionnels
- *Type scalaires* : e.g., String, Int, Float, lists

Schéma GraphQL - Exemple

```
1  type Query {
2    hero(name:String) : Character
3  }
4
5  type Character {
6    name : String
7    friends : [Character]
8    homeworld : Planet
9    species : Species
10 }
11
12 type Planet {
13   name : String
14   climate : String
15 }
16
17 type Species {
18   name : String
19   lifespan : Int
20   origin : Planet
21 }
```

- 1 *field* dans l'*Object type Query*
- 3 *object types* : Character, Planet et Species
- 5 *fields* avec un type de retour *Object*
 - Query hero, Character friends, Character homeworld, Character species, Species origin
- 5 *fields* avec un type de retour scalaire
 - Character name, Planet name, Planet climate, Species name, Species lifespan

Le langage de requête GraphQL consiste à construire une structure des objets et fields souhaités dans la réponse

- Le format de la requête est proche d'un format JSON
- Les arguments sont donnés aux fields qui en demandent

Le langage de requête - Exemple

```
1  {
2  | hero(name : "Luke Skywalker") {
3  | | name
4  | | friends {
5  | | | name
6  | | | homeworld {
7  | | | | name
8  | | | | climate
9  | | | }
10 | | | species {
11 | | | | name
12 | | | | lifespan
13 | | | | origin {
14 | | | | | name
15 | | | | }
16 | | | }
17 | | }
18 | }
19 }
```

Par exemple on ne demande pas ici à recevoir

- ni l'espèce de Luke Skywalker, mais seulement son nom et sa liste d'amis
- ni le climat de la planète de ses amis, mais seulement le nom de cette planète

1. Chaque *field* qui retourne un *Object type* (non scalaire) est associé à un *Resolver*

Un *resolver* est une fonction contenant le *code métier*

- entrées : paramètres du *field* et informations complémentaires (voir tuto)
- sortie : une donnée correspondant à l'*Object type* de sortie du *field*
 - e.g., un dictionnaire, une liste

2. Si le type d'un *field* est un scalaire la résolution remonte dans l'arbre au *field* suivant

- Parcours en profondeur de l'arbre : *DFS - Depth First Search*

Des détails ici : <https://graphql.org/learn/execution/>

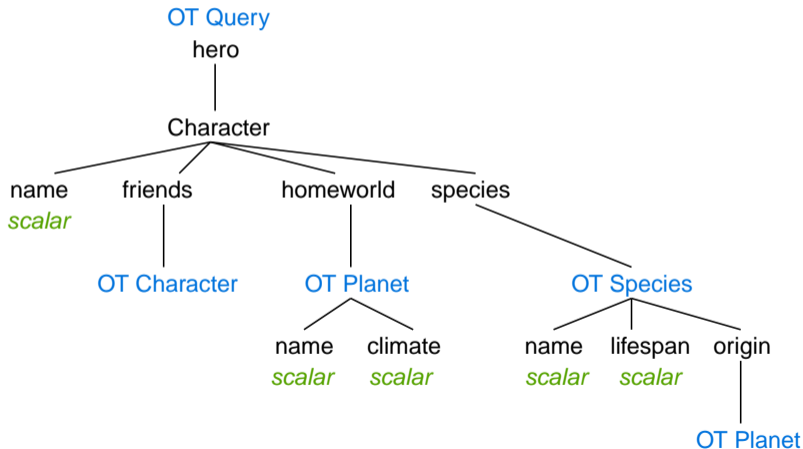
La résolution - Exemple

```
1  type Query {
2    hero(name:String) : Character
3  }
4
5  type Character {
6    name : String
7    friends : [Character]
8    homeworld : Planet
9    species : Species
10 }
11
12 type Planet {
13   name : String
14   climate : String
15 }
16
17 type Species {
18   name : String
19   lifespan : Int
20   origin : Planet
21 }
```

- 5 *fields* nécessitant un *resolver* car leur type de retour est un *Object*
 - Query hero
 - Character friends
 - Character homeworld
 - Character species
 - Species origin

La résolution - Exemple

Voici le graphe de résolution dont la racine est le field Query hero



La résolution - Exemple de réponse

```
1  {
2    "data": {
3      "hero": {
4        "name" : "Luke Skywalker",
5        "friends" : [
6          {
7            "name" : "R2-D2",
8            "homeworld" : {
9              "name" : "X",
10             "climate" : "good"
11            },
12            "species" : {
13              "name" : "robot",
14              "lifespan" : "eternal",
15              "origin" : {
16                "name" : "Y"
17              }
18            }
19          },
20          [
21            "name" : "C3PO",
22            "homeworld" : {
23              "name" : "Tatooine",
24            },
25            "species" : {
26              "name" : "Human",
27            }
28          ]
29        ]
30      }
31    }
32  }
```

Dans notre exemple la liste des amis peut elle même contenir une liste d'amis etc.

Est-ce qu'on crée une **boucle infinie** ?

La résolution - Pas de boucle

Non car la résolution **suit la requête** de l'utilisateur qui est elle nécessairement finie !

```
1  {
2  hero(name : "Luke Skywalker") {
3    name
4    friends {
5      name
6      friends {
7        name
8        friends {
9          name
10       }
11     }
12   }
13   homeworld {
14     name
15     climate
16   }
17   species {
18     name
19     lifespan
20     origin {
21       name
```

Comparaison REST, gRPC et GraphQL

	Performances	Debug	Découverte	Flexibilité	Nb fonctions
REST/OpenAPI	Orange	Green	Orange	Orange	Orange
gRPC	Green	Red	Orange	Red	Orange
GraphQL	Orange	Orange	Orange	Green	Green

Tutoriel GraphQL