Comprendre GraphQL

Date : 20 avril 2022
Speakers : Guillaume Scheibel et Geoffroy Couprie
Format : Université (3h)
URL : https://github.com/Geal/DevoxxFR-2022-University-Comprendre_GraphQL

Introduction

Problématique récurrente : les applications mobiles doivent appréhender le style d'API des vingt backends qu'elles doivent interroger.
Or, sans norme partagée et respectée à l'échelle de l'entreprise, chaque API a son propre design. Exemple : *firstName* avec camelcase, snakecase, ou juste *name* …

Un **pattern** apparu il y'a quelques années est le **Backend for Frontend** (BFF). Les clients appellent une seule API unifiée, orientée frontend (peut même comporter des références au design system). Ce pattern repousse le problème car au bout d'un moment, le BFF va exploser.

Promesse de GraphQL : palier ces défauts.

1. GraphQL Basic

Historique
GraphQL a été créé en 2012 par Facebook qui avait beaucoup de clients et d'API. GraphQL permet de récupérer les valeurs d'un seul champ. Une première spécification est sortie en 2015. La dernière spéc en date date de 2021.
En 2018 est créée la GraphQL Fondation.
URL : https://spec.graphql.org/ se lit très facilement. La spécification fait fois par rapport aux implémentations. Ne pas avoir peur de s'y référer.

Understanding GraphQL
Fausse croyance : GraphQL n'est pas une base NoSQL orientée graphe (comme Neo4j).

Le plus petit schéma possible en GraphQL :

```
1.  schema {
2.    query : Query
3.  }
```

Le **type Query** permet de faire du requêtage (lecture).
L'écriture (ajout, suppression, modification) passe par le **type Mutation**.
Le **type Subscription** est moins utilisé et permet de faire du streaming côté serveur.

```
type Query {
  "this property is cool"
  propertyName : MyType
}
```

Syntaxe proche du JSON avec possiblité de documentation entre "
Certains clients supportent le markdown pour afficher la doc.

```
type MyType {
  // autres propriétés
}
```

Types de base : **Int**, **Float**, **Boolean**, **String**, **Enum**, **List**
A noter qu'il n'existe pas nativement de Map.

Exemple:

```
type SpaceCat {
    name: Sring!
    age: Int
    missions: [Mission!]
}
```

Les crochets définissent une liste de missions.
Le point d'exclamation ! dénote la non nullabilité d'un champ. Dans l'exemple précédent, age peut être null, mais missions ne peut pas l'être.
Cet aspect est fort pratique avec Kotlin.

Exemple en TypeScript / JS

Dépendances Node :

- @graphql-tools/schema
- @graphql-tools/utils
- appolo-server : rendu GraphQL côté serveur

En TS un resolver permet de résoudre un nœud GraphQL.
Exemple :

```
const resolvers = {
    Query : {
       spaceCat : () -> {
         return {
           Name : "Neil Castrong"
         }
     }
   }
}
```

Plusieurs clients web GraphQL existent : Playground ou GraphiQL

Lorsqu'on écrit une requête, il faut spécifier explicitement l'ensemble des champs que l'on

veut récupérer. Le wildcard * ne fonctionne pas. On peut choisir l'ordre des champs.
Le serveur sait dédupliquer les données renvoyées.

Exemple de recherche par identifiant :

```
{
      spaceCatById(id : 1) {
          name
    }
}
```

Backend en Kotlin

Librairies : GraphQL Java
Il y'a quelques années, alors qu'il était chez Expedia, Guillaume a participé à la création de la librairie GraphQL Kotlin.
Utilisation du starter Spring Boot :

Dépendances Maven :

- com.expediagroup:graphql-kotlin-spring-server
- com.expediagroup:graphql-kotlin-hooks-provider

GraphQL Kotlin fonctionne par réflexion. Tous les champs publics vont être exposés dans le schéma.

```
@Component
class SpaceCatQueries : Query {
    fun spaceCat() = SpaceCat()
 }

data class SpaceCat(
   val name : String = "Neil CatSrong"
   val id: Int?
)
```

Configuration nécessaire dans le *application.yaml* pour ne scanner que les objets métiers (et pas par exemple les UIID Java).

Types Polymorphiques

```
interface Mission

data class NearOrbitMission(
      orbit : Orit
): Mission

enum class Orbit { LOW, HIGH }

data class HighOrbitMission(
      val distance: Int
): Mission
```

Étant donné que le type *Mission* n'a pas de propriété, ce dernier va être traduit en type polymorphique.

Rendu GraphQL :

```
union Mission : NearOrbitMission | HighOrbitMission
```

L'union permet de faire de l'agrégation de contenu.

GraphQL vient avec des types réservés
Exemple : _typeName

On doit spécifier les champs à récupérer pour chaque type de mission pouvant être renvoyé.
Syntaxe :

```
lowMission {
    … on NearOrbitMission {
      orbit
   }
}
```

GraphQL supporte la notion d'interface (comme en Java).

Nouvel exemple visant à remplacer le champs *name* par une fonction.
Avec GraphQL Kotlin, la fonction n'est appelée que si le client demande de ramener ce champ : cela permet d'optimiser les performances et éviter, par exemple, les N+1 select.

En input object, il n'est pas possible de passer une interface.

Questions :

- L'annotation *@GraphQLIgnore* permet de ne pas faire apparaitre les beans injectés dans le schéma GraphQL
- Pas de base de données particulièrement recommandées pour GraphQL

2. GraphQL Avancé

Le langage Kotlin vient avec des coroutines, sortes de threads très légères permettant de paralléliser du code de manière performant.

```
class SpaceCat (
    val name : String
) {
    fun mission() = Mission()
}

class Mission {
    fun where(): String {
        Thread.sleep(2000)  // simule latence réseau
        return "Somewhere"
    }
    fun orbit(): String {
        Thread.sleep(100)
        return "LOW"
```

```
    }
}
```

L'exécution de cette requête prend un peu plus de 2,1 secondes.
La directive **@defer** permettrait de différer l'arrivée de where qui est plus lent. Mais cette directive n'est pas implémentée dans GraphQL Kotlin.

```
{
    spaceCat : {
         mission {
            orbit @defer
            where @defer
         }
    }
}
```

Defer repose sur du multipart avec 2 réponses qui arrivent.

En Kotlin on utilise les coroutines pour mettre de suspendre l'appel à une fonction et passer à la suivante :

```
class Mission {
    suspend fun where(): String  = coroutineScope {
      delay(2000)
      "Somewhere"
   }
    suspend fun orbit(): String  = coroutineScope {
      delay(100)
      "LOW"
   }
}
```

On n'attend plus que 2 secondes.

Remarque : le client peut utiliser des **alias** pour récupérer plusieurs fois le même champ.

Directives client

Les **directives client @include** et **@skip** (inverse de @include) sont couramment utilisées.

```
Query myQuery($condition: Boolean!) {
  spaceCat {
    mission {
        orbit @include(if : $condition)
        where
    }
  }
}
```

Directives server

```
const val LOWER_CASE_DIRECTIVE  = "LowerCase"
```

```
2.  @GraphQLDirective(
3.      name = LOWER_CASE_DIRECTIVE
4.      locations = { Introspection.DirectiveLocation.FIELD}
5.  )
6.  annotation class LowerCaseDirective
7.
8.  class LowerCaseDirectiveWiring:KotlinSchemaDirectiveWiring {
9.      override fun onField(environment: KotlinFieldDirectiveEnvironment):
    GraphQLFieldDefinition {
10.         val field = environment.element
11.         val originalDataFetcher = environment.getDataFetcher()
12.         val lowercased = DataFetcherFactories.wrapDataFetcher(originalDataFetcher) {
13.                 _ , value -> value.toString().lowercase()
14.         }
15.         environment.setDataFetcher(lowercased)
16.         return field
17.     }
18. }
```

Nécessite de la glue technique supplémentaire pour faire fonctionner cette directive.

Scalaires
Les types primitifs peuvent être augmentés.
Exemple : lorsque le serveur rencontre un UIID, appel un *toString()* à la place de la réflexion

```
1.  val graphqlUUID = GraphQLScalarType.newScalar()
2.      .name("UUID")
3.      .coercing(UUIDCoercing)
4.      .build()
5.
6.  object UUIDCoercing: Coercing<UUID, String> {
7.       override sun serialize( ….
8.  }
```

Le Contexte
Des éléments du contexte d'exécution peuvent être passés tout au long de la résolution du graphe.

Request
{
  "query"
  "operationName"
  "variables"
}

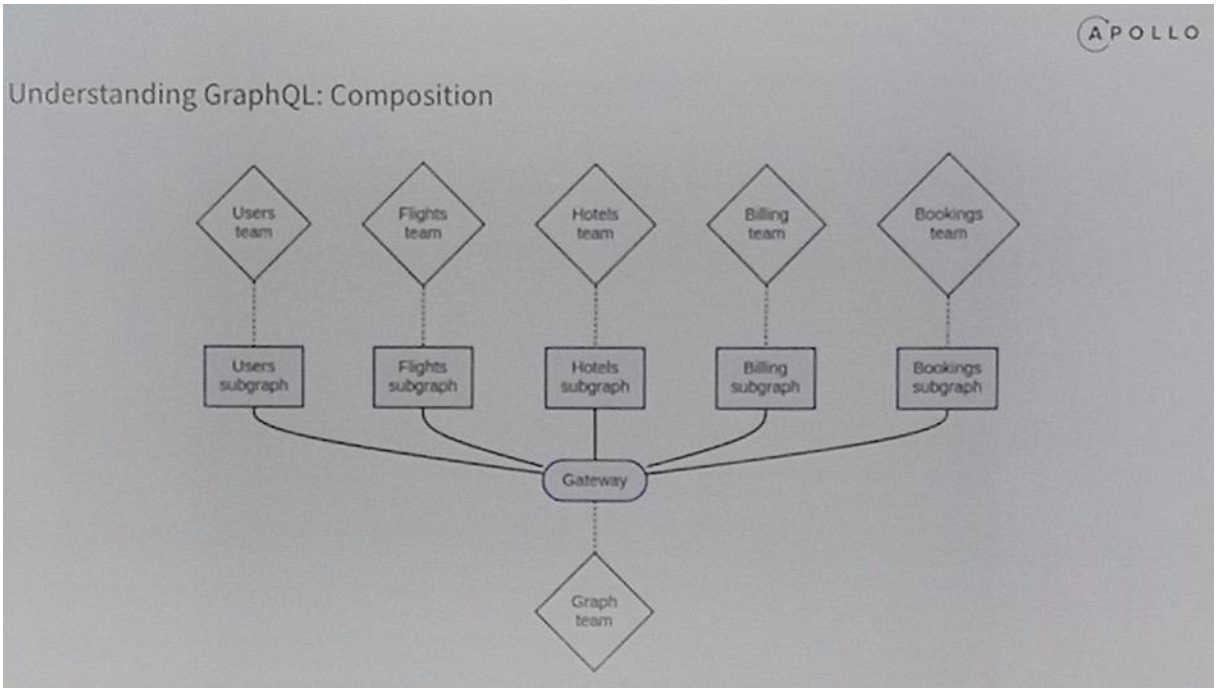
Response
{
  "data" :
  "error" :
  "extensions" :
}

En GraphQL, contrairement à l'approche REST, toutes les requêtes HTTP sont envoyées avec le verbe POST et tous les retours sont en OK 200

Réponses aux questions :

- Le chaînage des requêtes n'est pas supporté
- La plupart du temps, il n'y a pas besoin de versioning en GraphQL. GraphQL vient avec du tracing : on peut avoir de l'outillage Apollo permettant de savoir qui utilise quelle donnée pouvant être nullable afin de pouvoir changer le schéma. GraphQL permet de déprécier des champs via une directive.

3. Composition de schéma



Chaque équipe défini un sous-graphe et une gateway les agrège.
Possibilité de faire des jointures entre graphes.

Stitching

Exemple avec 3 services : Kotlin, TypeScript et Gateway
La gateway effectue du « stiching ». GraphQL stitching a été introduit avec GraphQL v12.
Implémentation disponible sur Node.
Code à ajouter lorsqu'il y'a des conflits entre le nom des types des différents sous-schémas.
Sur la gateway, on peut étendre le type SpaceCat pour ajouter une Mission.

Difficulté du stitching : nécessite une équipe dédiée à la gateway. Cette dernière devient le goulet d'étranglement.

Apollo Router

Encore en preview, Apollo Router est écrit en Rust et remplacera à termes la Gateway.

Apollo Studio combine tous les sous-graphes. En amont, les différentes équipes envoient leur spécs au même endroit avec rover.
Un changelog est disponible.
Lorsqu'un dév supprime un champ utilisé par des consommateurs, la CI bloque.
Nécessité d'ajouter du code dans le routeur pour implémenter des jointures entre sous-graphes.