GraphQL

Francois.Rioult@unicaen.fr

2 février 2021

Table des matières

1 Préliminaires

1.1 Les nouveautés de la syntaxe Javascript ES6

Voir `http://ccoenraets.github.io/es6-tutorial/`.

Les dernières versions d'ECMAScript [1] bouleversent la syntaxe traditionnelle de Javascript et ne sont pas supportées par les navigateurs classiques. Lorsqu'on utilisera React.js par exemple, il faudra transpiler à l'aide de Babel.

1. ECMA - European Computer Manufacturers Association, est une organisation de standardisation active dans le domaine de l'informatique.

— les modèles de libellés : ce sont des chaînes de caractères délimités par des *backquote* permettant d'insérer des fragments de code Javascript :
```
var a = 5;
var b = 10;
console.log(`Quinze vaut ${a + b}`);
```
— les fonctions flèche (*arrow function*) :
```
amortization.forEach(month => console.log(month));
```
Leur intérêt est que la valeur de `this` est la même à l'intérieur qu'à l'extérieur.
— les imports sont plus agréables :
```
import {
  nomDExportDeModuleVraimentVraimentLong as nomCourt,
  unAutreNomDeModuleLong as court
} from '/modules/mon-module.js';
```
— les `export` précisent les éléments d'un module qui seront réutilisés par `import`.

export nommé : pour exporter plusieurs valeurs.

```
export { maFonction };
export const machin = Math.sqrt(2);
```

export par défaut : pour exporter une seule valeur par module, éventuellement anonyme, et qui sera nommée lors de l'import :

```
export default function (x) {
  return x * x * x;
}
```

puis

```
import cube from 'mon-module';
```

— les promesses (*promise*) : permettent d'effectuer de la programmation asynchrone et viennent remplacer les *callback* :
```
service.findAll()
    .then(rates => {...})
    .catch(e => console.log(e));
```

1.2 Les promesses (*promise*)

Ces objets permettent de réaliser des opérations asynchrones. Une promesse indique une valeur disponible maintenant, plus tard ou jamais. Ses états sont :

— pending (en attente) : état initial, la promesse n'est ni remplie, ni rompue ;
— fulfilled (tenue) : l'opération a réussi ;
— rejected (rompue) : l'opération a échoué ;
— settled (acquittée) : la promesse est tenue ou rompue mais elle n'est plus en attente.

voir `https://mdn.mozillademos.org/files/15911/promises.png`
Pour retourner depuis une promesse (et donc la tenir), on utilise `.resolve(valeur)`.

2 GraphQL

GraphQL est une technologie de Facebook permettant d'implémenter des services de base de données par un point de connexion (*endpoint*) permettant d'envoyer des requêtes en HTTP et de recevoir du JSON. Il s'agit d'une interface servant des données, qui ne fait pas de supposition sur le gestionnaire interrogé (SQL, Mongo, etc.). C'est une alternative aux API REST (representational state transfer).

L'une des motivations est de fournir une passerelle HTTP entre un gestionnaire de bases de données et un serveur d'application, pour découpler ce gestionnaire et ce serveur, par exemple dans le cadre de l'utilisation d'image Docker.

Cette architecture permet de concevoir des applications web dans lesquelles la partie Modèle du pattern MVC est séparée en du code applicatif et un ensemble de requêtes à effectuer sur un point d'accès à une base de données. On retrouve ici la philosophie des frameworks comme Symfony.

Attention : GraphQL est une spécification de la façon de décrire un modèle de données et d'effectuer des requêtes. Ce n'est pas en soi une technologie et il faudra se tourner vers des acteurs comme Apollo pour trouver des implémentations dans les différents langages (PHP, Python, Node, etc.).

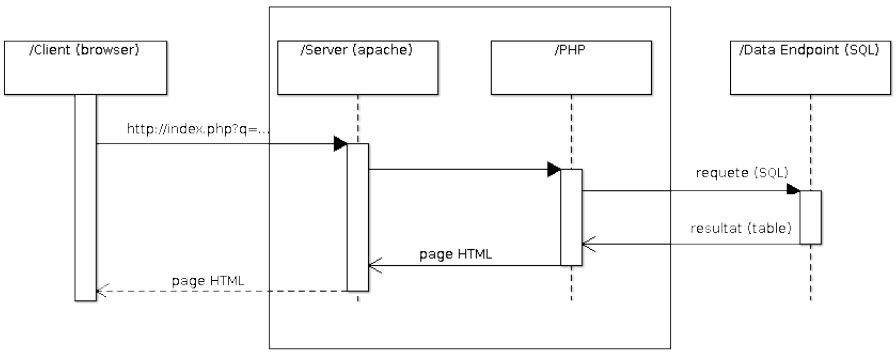

FIGURE 1 – Architecture traditionnelle

On peut opposer deux architectures pour la réalisation d'une application web :

1. l'architecture traditionnelle (figure 1) demande au serveur (Apache / PHP) d'effectuer beaucoup de calculs :
   (a) requête HTTP sur le serveur Apache
   (b) exécution sur le serveur du script PHP
   (c) le script PHP transmet une requête SQL au serveur de base de données
   (d) la base de données de données renvoie une table

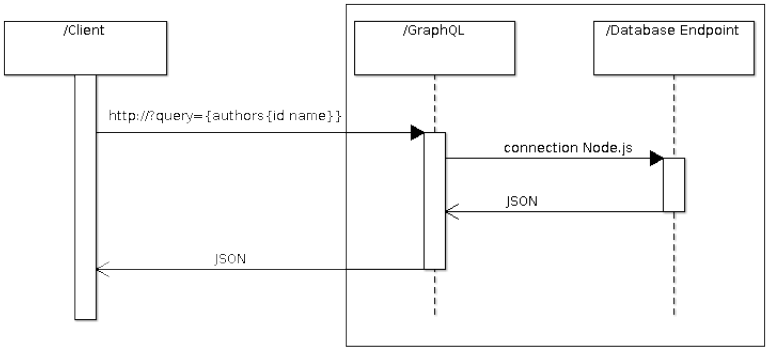

FIGURE 2 – Database Endpoint par GraphQL

(e) le script PHP transforme ces données en HTML

(f) Apache renvoie une page HTML au client

2. une architecture sans serveur applicatif (figure 2) :

(a) le point d'accès GraphQL reçoit une requête HTTP contenant la requête à effectuer

(b) la requête est transmise à la base de données

(c) la base de données retourne du JSON

(d) le JSON est transmis au client.

Cette manière de procéder a plusieurs avantages (et donc des inconvénients) :

— le serveur effectue moins de calcul car il n'a plus à exécuter des scripts en PHP par exemple ni à transformer les données d'une table à sa vision en HTML : il se contente de transférer des données au format JSON

— c'est le client qui effectue la majorité des calculs : la majeure partie de l'application est exécutée côté client.

— la société qui développe l'application n'a plus besoin de maintenir une architecture de serveurs applicatifs, elle n'a besoin que d'un serveur de base de données. La gestion de ce dernier peut être déléguée à un autre acteur dont c'est le métier.
— la mise à disposition de l'application peut être déléguée à un store, ce qui allège encore la tâche du développeur.

3 Les concepts d'une requête GraphQL

Cette présentation est basée sur le tutoriel disponible à `http://graphql.github.io/learn/`.

Les requêtes en GraphQL sont définies par un graphe rédigé en JSON simplifié, uniquement les clés, par exemple :

```
query maRequete {
  hero {
    name
    appearsIn
    friends {
        name
    }
  }
```

Les données obtenues ont la même forme, en JSON :

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ],
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
}
```

Cette similarité entre requêtes et données est un point essentiel de GraphQL.

Le préambule `query maRequete` indique à GraphQL que le type opératoire utilisé ici est “query” et fourni un nom à l’opération. Ce préambule n’est pas obligatoire.

3.1 Paramètres

Il est possible d’indiquer des paramètres, analogues au WHERE du SQL, pour chaque noeud du graphe :

```
{
  human(id: "1000") {
    name
    height
  }
}
```

3.2 Alias

Les alias permettent de modifier les clés des noeuds du graphe résultat :

```
{
  hero {
    sonNom: name
}
```

On obtient :

```
{
  "data": {
    "hero": {
      "sonNom": "R2-D2"
    }
  }
}
```

3.3 Fragments

GraphQL offre la possibilité de mutualiser des portions de requêtes, afin de construire des requêtes élaborées à la manière d’une programmation à l’aide de fonctions :

```
{
  leftComparison: hero(episode: EMPIRE) {
    ...comparisonFields
  }
  rightComparison: hero(episode: JEDI) {
    ...comparisonFields
  }
}
```

```
fragment comparisonFields on Character {
  name
  appearsIn
  friends {
    name
  }
```

3.4 Variables

Les variables permettent de paramétrer les requêtes, pour les rendre plus dynamiques. Il ne faut pas les confondre avec les paramètres, qui sont internes aux requêtes.

```
query HeroNameAndFriends($episode: Episode = JEDI) {
  hero(episode: $episode) {
    name
    friends {
      name
    }
  }
```

La valeur des variables doit être spécifiée dans la section dédiée (dictionnaire des variables) du transport de la requête.

3.5 Directives

Les directives sont des arguments de la requête qui permettent d'influencer le rendu des données obtenues :

```
query Hero($episode: Episode, $withFriends: Boolean!) {
  hero(episode: $episode) {
    name
    friends @include(if: $withFriends) {
      name
    }
  }
}
```

Les directives permettent l'inclusion (`@include`) ou l'exclusion (`@skip`).

4 Mutations

Les techniques de requêtage ci-dessus ne permettent que de récupérer des données (*fetching*). Pour modifier les données, il est nécessaire de faire appel aux *mutations*.

Côté modèle, on définit un type spécial pour les mutations, dans lequel on énumère les requêtes de mutation. Par exemple ici, des requêtes chargées d'enregistrer du JSON dans MongoDB, et qui renvoient l'identifiant du document :

```
scalar JSON

schema{
    query: Query
    mutation: Mutation
}

type Mutation{
    saveToDev(serial: JSON): ID!
}
```

Côté résolveurs, on récupère les données dans les arguments :

```
const resolvers = {
    Mutation: {
        saveToDev: (parent, args) => {
              return new Promise((resolve, reject) => {
                const db = client.db(dbName);
                db.collection("dev").insertOne(args.serial, function (err, res)
                    if (err)
                        throw err;
                    console.log("1 document inserted");
                    resolve(res.insertedId);
                    //console.log("res", res);
                });
            }).then(result => {
                console.log("result", result);
                return result
            });
        },
 ...
```

Il n'est pas possible d'utiliser GET. En PHP par exemple, on utilisera `curl` :

```
$query = '{"query":"mutation($json:JSON){saveToDev(serial:$json)}",' .
                    '"variables":{"json": ...  }}');
        $ch = curl_init();

        curl_setopt($ch, CURLOPT_URL, GRAPHQL_SERVER);
        curl_setopt($ch, CURLOPT_POST, TRUE);
        curl_setopt($ch, CURLOPT_HTTPHEADER, array('Content-type: application/js
        curl_setopt($ch, CURLOPT_POSTFIELDS, $query);

        // Receive server response ...
        //curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
        $server_output = curl_exec($ch);
```

```
curl_close($ch);

// Further processing ...
if ($server_output == "OK") {
    echo "OK<br>";
} else {
    echo "KO<br>";
}
```

5 Utilisation par HTTP

5.1 GET

`http://myapi/graphql?query={me{name}}`

5.2 POST

La requête doit utiliser le type de contenu `application/json`.
Le corps de la requête est du JSON, par exemple :

```
{
  "query": "...",
  "operationName": "...",
  "variables": { "myVariable": "someValue", ... }
}
```

6 Schémas et types

L'objet de base d'un schéma est le *type*, composé de champs (*field*). On dispose de types élémentaires, dits *scalaires* : ID, Int, Float, Boolean et String. Les crochets définissent un tableau, le point d'exclamation indique que le champ ne peut être *null*.

```
type Character {
  name: String!
  appearsIn: [Episode]!
}
```

Les champs peuvent être pourvus d'argument, par exemple un type d'unité :

```
type Starship {
  id: ID!
  name: String!
  length(unit: LengthUnit = METER): Float
}
```

On peut ajouter ses propres scalaires :

```
scalar Date
```

ou des énumérations :

```
enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}
```

Si la plupart des objets du schéma sont des types, on considèrera les deux types particuliers que chaque service GraphQL doit posséder, *query* et *mutation* :

```
schema {
  query: Query
  mutation: Mutation
}

type Query {
  hero(episode: Episode): Character
  droid(id: ID!): Droid
}
```

6.1 Interface

Les interface sont des types abstraits dont les champs sont obligatoires dans les types implémentant l'interface :

```
interface Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
}

type Human implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  starships: [Starship]
  totalCredits: Int
}

type Droid implements Character {
  id: ID!
  name: String!
```

```
  friends: [Character]
  appearsIn: [Episode]!
  primaryFunction: String
}
```

Les interfaces permettent donc d'introduire des types généraux (pères), recouvrant plusieurs types plus spécifiques (fils). La spécification s'effectue à l'aide de fragments en ligne, avec la directive `... on <type>`;

```
query HeroForEpisode($ep: Episode!) {
  hero(episode: $ep) {
    name
    ... on Droid {
      primaryFunction
    }
  }
}
```

Une aternative aux interface est l'utilisation de type *union* :

```
union SearchResult = Human | Droid | Starship

{
  search(text: "an") {
    ... on Human {
      name
      height
    }
    ... on Droid {
      name
      primaryFunction
    }
    ... on Starship {
      name
      length
    }
  }
```

6.2 Types réservés aux mutations

Dans le cas de mutations, les arguments d'une requête peuvent être d'un type complexe, qui nécessite d'être défini à l'avance :

```
input ReviewInput {
  stars: Int!
  commentary: String
}
```

```
mutation CreateReviewForEpisode($ep: Episode!, $review: ReviewInput!) {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}
```

Les variables de la requête sont alors structurées :

```
{
  "ep": "JEDI",
  "review": {
    "stars": 5,
    "commentary": "This is a great movie!"
  }
}
```