GraphQL & React, expliqués aux développeurs Backend

par Christophe Jolas

OBJECTIFS

- Comparer REST et GraphQL
- Impact sur le développement en React
- Partage d’expérience sur les deux API

Attention : Cette présentation n’est pas un cours sur GraphQL et React.

CE QUE NOUS ALLONS VOIR

- Format
- Point d’entrée (endpoint)
- Base de donnée (database)
- Affichage des objets / manipulation des données
- Développement avec React
- Retour d’expérience sur GraphQL

FORMAT

REST :

- C’est le standard API le plus utilisé.
- Il est basé sur le protocol HTTP.
- Son contenu de réponse peut être sous format JSON, XML, CSV, RSS...

GraphQL :

- Langage de requête conçu pour fonctionner sur un seul terminal via HTTP.
- Son contenu de réponse est en JSON. C’est le seul format supporté.

On remarque **une souplesse de l’API REST pour l'affichage du contenu de réponse**.

POINT D’ENTRÉE - ENDPOINT

REST :

Un ‘endpoint’ différent pour chaque ressource, avec des méthodes de requête HTTP différentes :

- http://www.example.com:3000/hives : GET, POST
- http://www.example.com:3000/hives/1 : PUT, PATCH, DELETE
- http://www.example.com:3000/bees/hive/1 : GET

GraphQL :

- http://www.example.com:4000/graphql : GET, (POST ?)

Via ces méthodes, la notion de *query*, *mutation* entre en jeu.

Inconvénient : Il ne permet pas de mutualiser le cache HTTP et de servir la même réponse.
C’est un réel problématique pour des sites à fort trafic.

DATABASE & DATAS

REST et GraphQL peuvent se plugguer à n'importe quel type de base de données (MySQL, PostgresQL…).
Pour notre exemple, nous allons réaliser **les mêmes requêtes SQL** avec la base de donnée **MySQL**.

REST :

Pour faire un appel à un point d'entrée, on utilisera un client HTTP basé sur la promesse (**Axios**).

GraphQL :

Apollo Client est conçu pour nous aider à créer rapidement une interface utilisateur qui récupère les données avec GraphQL et qui peut être utilisée avec n'importe quel serveur JavaScript.

Les données utilisées seront affichées sont sous **format JSON**.

DATABASE ET DATAS (suite)

Pour illustrer nos exemples, nous allons créer des tables :

Hive	Bee	Role
• ID • name • number • location	• ID • name • birthday_date • hive_id • role_id	• ID • name • position

DATABASE ET DATAS (suite)

Nous allons réaliser la même requête SQL pour les 2 API, afin de pouvoir les comparer.

Nous cherchons à sélectionner l'ensemble des abeilles associées à leur rôle et vivant dans la même Ruche.

```
SELECT

b.name AS bee_name, b.birthday_date AS bee_birthday, r.name AS role, h.name AS hive_name
h.number AS hive_number, h.location AS hive_location

FROM bee b

INNER JOIN hive h ON b.hive_id = h.id
INNER JOIN role r ON b.role_id = r.id

WHERE b.hive_id = ?
```

DATABASE ET DATAS (suite)

REST :

```
Bee.getAllBeeByHive = function getAllBeeByHive(hiveId, result)
{
    let sqlRequest = "SELECT b.name AS bee_name, b.birthday_date AS bee_birthday, r.name AS role, h.name AS hive_name, h.number AS hive_number, h.location AS hive_location FROM bee b INNER JOIN hive h ON b.hive_id = h.id INNER JOIN role r

    sql.query(sqlRequest, hiveId,  function (err, res) {
        if (err) {
            result(null, err);
        } else {
            let bees = res.map(obj => {
                return {
                    "bee_name" : obj.hive_name,
                    "bee_birthday": obj.hive_number,
                    "role": obj.role,
                };
            });
            let arr = [
                {
                    "hive_name" : res[0].hive_name,
                    "hive_number": res[0].hive_number,
                    "hive_location": res[0].hive_location,
                    "bees": bees
                }
            ];
            result(null, arr);
        }
    });
};
```

DATABASE ET DATAS (suite)

L'idée est d'avoir le même format de données (objet).

REST :

Après réception des requêtes SQL, la structure et le format ont été travaillés. Les données n'ont pas été formatées.

GraphQL :

On définit 2 types d'objet (*Object types*) et leurs champs (*fields*) : Bee et Organisation. Les données seront dispatchées à l'aide de *resolver*.

```
const typeDefs = gql`
    scalar DateTime

    type Bee {
        bee_id: ID
        bee_name: String
        bee_birthday: DateTime
        role: String
    }

    type Organisation{
        hive_id: ID
        hive_name: String
        hive_number: Int
        hive_location: String
        bees:[Bee]
    }

    type Query {
        beesByHive(hive_id: Int!): Organisation
    }
`;
```

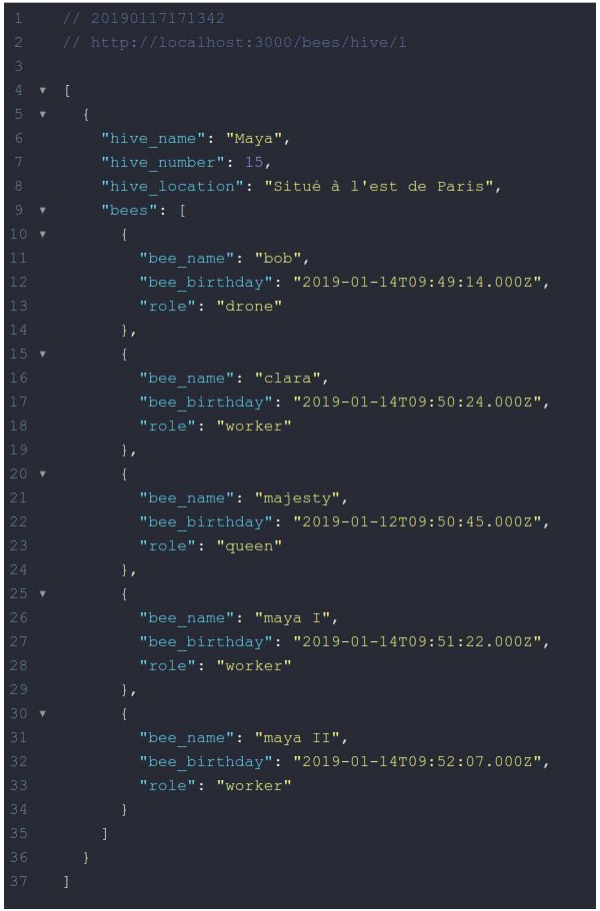
DATABASE ET DATAS (suite)

GraphQL (suite):

```
const schema = {
    typeDefs: typeDefs,
    resolvers: {
        DateTime: GraphQLDateTime,
        Query: {
            beesByHive: (_, data) => { return
beeQuery.getAllBeeByHive(data) },
        },
```

```
        Organisation: {
            hive_name: async (data) => {
                return data[0].hive_name;
            },
            hive_number: async (data) => {
                return data[0].hive_number;
            },
            hive_location: async (data) => {
                return data[0].hive_location;
            },
            bees: async (bee) => {
                const arr = await
Promise.all(Object.values(bee).map(async (data) => {
                    return {
                        'bee_name': data.bee_name,
                        'bee_birthday': data.bee_birthday,
                        'role' : data.role
                    };
                }));
                return arr;
            },
        },
    }
};
```

AFFICHAGE DE TABLEAU ET D'OBJET (REST)



```
// 20190117171342
// http://localhost:3000/bees/hive/1

[
  {
    "hive_name": "Maya",
    "hive_number": 15,
    "hive_location": "Situé à l'est de Paris",
    "bees": [
      {
        "bee_name": "bob",
        "bee_birthday": "2019-01-14T09:49:14.000Z",
        "role": "drone"
      },
      {
        "bee_name": "clara",
        "bee_birthday": "2019-01-14T09:50:24.000Z",
        "role": "worker"
      },
      {
        "bee_name": "majesty",
        "bee_birthday": "2019-01-12T09:50:45.000Z",
        "role": "queen"
      },
      {
        "bee_name": "maya I",
        "bee_birthday": "2019-01-14T09:51:22.000Z",
        "role": "worker"
      },
      {
        "bee_name": "maya II",
        "bee_birthday": "2019-01-14T09:52:07.000Z",
        "role": "worker"
      }
    ]
  }
]
```

http://www.example.com:3000/bees/hive/1

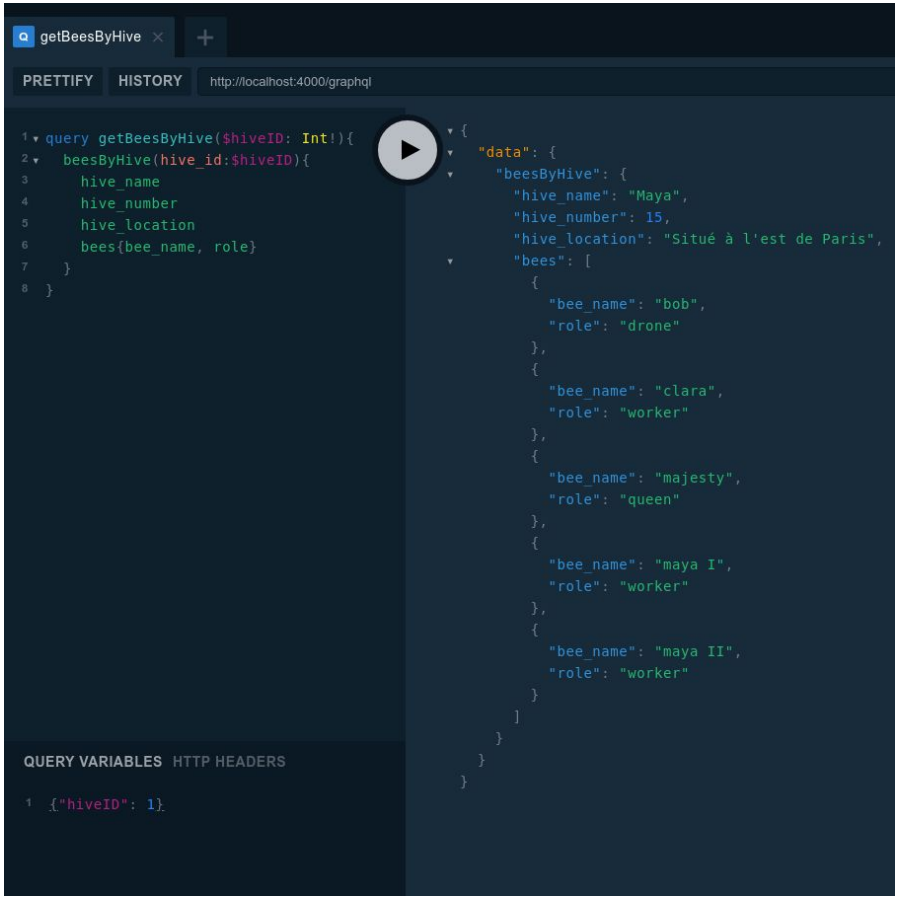
Si je veux n'afficher qu'un seul champ comme 'bee_name' associé à 'hive', je vais devoir modifier ma requête SQL et créer un autre 'endpoint'.

Je suis limité par les contraintes de l'API REST. J'ai donc moins de liberté…

Je peux très bien garder cet 'endpoint' et n'utiliser qu'une partie pour afficher mes données, côté Front.

➡ Donc, est-ce vraiment pertinent ?

AFFICHAGE D'OBJET (GraphQL)



http://www.example.com:4000/graphql

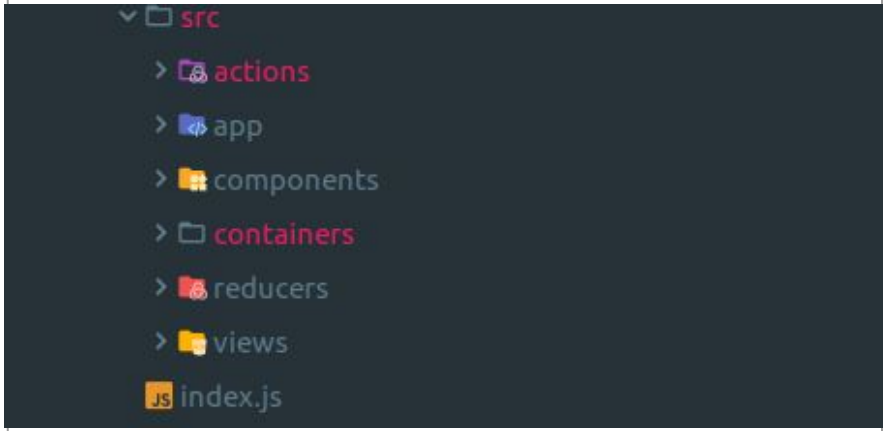
Selon ses besoins, le client a la liberté d'afficher les données associées à leur champ.

Il peut très bien choisir un ou plusieurs champs, sans modification de requête SQL.

L'avantage de cette souplesse est de nous d'éviter de refaire des requêtes SQL, pour un besoin précis.

DÉVELOPPEMENT AVEC REACT

REST :



La couche Redux est fortement recommandée voire obligatoire, dans le cas des sites à fort trafic.
On évite de faire de la programmation spaghetti.

Le temps de développement est allongé.

DÉVELOPPEMENT AVEC REACT (suite)

GraphQL :

```
import React, { Component } from 'react';
import { graphql, compose } from "react-apollo";
import getAllBeeByHiveQuery from "../queries/getAllBeeByHive.gql";

class Organisation extends Component {
    render() {
        if(this.props.beesByHive.loading){
            return <div>Chargement...</div>
        }

        return (
            <div>
                <h1>Détails d'une ruche : {this.props.beesByHive.hive_name} /
{this.props.beesByHive.hive_number}</h1>
                .....
            </div>
        )
    }
}
```


```
export default compose(
    graphql(getAllBeeByHiveQuery,{
        name:"beesByHive",
        options : (props) => {
            return {
                variables : {
                    hive_id : props.params.hive_id
                }
            }
        }
    }),
)(Organisation);
```

DÉVELOPPEMENT AVEC REACT (suite)

GraphQL (suite):

Quelques mots sur les bonnes pratiques à mettre en place :

- Environnement : Désactiver GraphiQL dans l'environnement de production (nécessité de créer d'autres environnements (test, dev)),
- Gestion de réponse : afficher, à minima, des informations (*data*, *errors*),
- Nullabilité : utilisation de préférence des types *non-null* (!), pour éviter des confusions avec des *null* des données,
- Base de donnée : utilisation de *DataLoader* (couche d'extraction des données) pour la gestion de cache et le traitement par lot des données.

En utilisant ces pratiques recommandées par GraphQL, (https://graphql.org/learn/best-practices/)
On gagne en productivité en terme de développement.

Nous pouvons utiliser GraphQL pour des sites vitrines, institutionnelles.

CONCLUSION : Retour d’expérience sur GraphQL

- Nécessité d’apprentissage
- Souci de configuration avec *resolver,* avec d’autres langages (hors JS)
- Nécessité d’adhésion collégiale (avec l’équipe)
- Problématique avec des sites à fort trafic et nécessité de revoir l’architecture des applications

➡ **Pour des sites à fort trafic, l’API REST a encore de beaux jours devant lui.**

➡ **GraphQL, malgré ses défauts et sa jeunesse, est une bonne alternative face au format REST, en terme de productivité (développement). Je vous encourage vivement à l’utiliser pour des sites institutionnelles, vitrines...**

MERCI !

DES QUESTIONS ? :)