GraphQL, Retour d'expérience

Dans le cadre du développement du backend du site web du LJK

Caroline Bligny, le 10 décembre 2020

Contexte
Pourquoi GraphQL?

- Développement d’un backend classique pour un site web
  - Base de données simple - actualités, thèses, séminaires, offres d’emploi
  - API pour la distribution des données, consommée par le site public
  - Partie admin pour la saisie des données
- Le graal : trouver un (des) outil(s) qui permette de faire cela:
  - en un minimum d’effort,
  - sans redondance du modèle,
  - adaptable en cas de modification du modèle

L’heure des choix

- GraphQL à la place d’une API REST classique pour avoir un seul endpoint.

- Base de données MongoDB car plus souple qu’une base de donnée relationnelle.

- Framework web Python flask + flask-admin pour faire comme AMIES.

Pile logicielle complete

- Base de données mongoDB
- Backend python
  - ORM : mongoengine
  - Framework web flask
  - Interface de saisie : flask-admin
  - API GraphQL : graphene
- Frontend spip, consomme l'API GraphQL pour afficher les données

GraphQL c’est quoi

- C’est un langage de requête (Graph Query Langage) ainsi que la définition d’un environnement d’exécution. C’est un standard.

- Défini les interactions client-serveur pour répondre aux demandes de données du client. Utilisé surtout pour le web. « Backend for frontend pattern »

- Dernières spécifications là : https://spec.graphql.org/draft/

- Créé par Facebook en 2012, code ouvert en 2015

- Créé dans le contexte des applications mobiles pour optimiser les transferts de données. Evite les problèmes de données insuffisantes (under-fetching) ou surnuméraires (over-fetching)

- Issu de l’écosystème javascript, react

Coté client - l'objectif

- Le client défini l'objet qu'il veut recevoir
- La structure de la requête est identique à la structure de la réponse.
- Documentation automatique
- Interface graphiql pour tester les requêtes

```
{
  orders {
    id
    productsList {
      product {
        id
        name
        weight
        price
      }
      quantity
    }
    totalAmount
  }
}
```

```
{
  "data": {
    "orders": [
      {
        "id": 0,
        "productsList": [
          {
            "product": {
              "id": 1,
              "name": "orange",
              "weight": 0.3,
              "price": 1.5
            },
            "quantity": 100
          }
        ],
        "totalAmount": 250
      }
    ]
  }
}
```

Les grand principes

- **Une seule route** - ou endpoint.

- Rôle central du **schéma**. Il défini la structure des données et les opérations possibles sur ces données. C'est un ensemble d'objets ou « **types** » avec une liste de champs. Il doit suivre la spec GraphQL.

- Fonction d'introspection du schéma —> documentation automatique, vérification de la syntaxe par le client

- **Structure arborescente**. Attention le nom Graph(QL) est usurpé

- Le serveur fourni le schéma et implémente les ***resolvers*** pour chaque champ *(Il faut quand même un peu coder …)*

Exemple de Schéma graphql

```
// Définition des types de données

type Person {
  id: ID!
  firstName: String! // ! pour obligatoire
  lastName: String!
  these: These
}

type News {
  id: ID!
  title: String!
  newsType: NewsType!
  publishDate: Date!
  content: String
  validated: Boolean!
}

type These {
  id: ID!
  speaker: Person!
  eventDate: Date!
  eventPlace: String!
  title: String!
  description: String
  directors: [Person] // liste de Person
}

scalar Date

enum NewsType {
  ACTUALITE
  FAIT MARQUANT
  MANITESTATION
}

// Définition des requêtes sur les données
type Query {
  allNews(title: String, publishDate: Date, newsType: NewsType): [News]
  oneNews( id: ID! ): News
  theses( directorName: String, year:Int ): [These]
  these( id: ID, speakerName: String ): These
}

//Définition des modifications de données possibles
type Mutation {
  addNews( content: String, author: ID! ): News
  updateNews( id: ID!, content: String, author: ID! ): News
  deleteNews( id: ID! ): News
  validateNews( id: ID! ): News
}

// Subscription : le server pousse les modifs de données au client
type Subscription { ... }
```

Pour développer

- Pile logicielle d’origine : node, express, graphql-js, relay, et react pour le client - Apollo platform

- Relay est une surcouche à graphql qui gère entre autre la pagination. Structure des requêtes/réponses spécifique

- Plein d’autres librairies, dans plein d’autres langages : https://graphql.org/code/ - JavaScript, Go, PHP, Python, Java,C# / .NET, Ruby, Elixir, Rust, Swift, Scala, Flutter, Clojure, C / C++, Elm, OCaml, Haskell, Erlang, Groovy, R, Julia, Perl, D

- En python, Graphene vs Ariadne, code-first approach vs schema-first approaches

- Graphene, le schéma est déduis de l’ORM, utilisation de resolvers par défaut

graphene - extrait schema.py - query

```python
class News(MongoengineObjectType):

    # champ additionnel calculé dans models.py
    publish_year = graphene.Int(source='publish_year')

    image_url=graphene.String() # Champ additionnel calculé ici
    def resolve_image_url(parent, info):
        url = ""
        if parent.image:
            url = urllib.parse.urljoin(host, parent.image)
        return url

    class Meta:   # Adaptation du modele de models.py
        model = NewsModel
        exclude_fields =('str_event_date',)

class Query(graphene.ObjectType):

    all_news = graphene.List(News, event_type=graphene.String()) # Toutes les news
    def resolve_all_news(self, info, **kwargs):
       return NewsModel.objects.get(event_type=event_type)

    one_news = graphene.Field(News, id=graphene.String())       # Une seule news
    def resolve_one_news(self, info, id):
        return NewsModel.objects.get(id=id)

schema = graphene.Schema(query=Query, mutation=Mutation, types=[News,])
```

graphene - extrait schema.py - mutation

```
class NewsInput(graphene.InputObjectType):
    event_type = graphene.String()
    title = graphene.String()
    english_title = graphene.String()
    description = graphene.String()
    english_description = graphene.String()

class CreateNews(graphene.Mutation):
    news = graphene.Field(News)

    class Arguments:
        news_data = NewsInput()

    def mutate(root, info, news_data):
        news = NewsModel(
            event_type = news_data.event_type,
            title = news_data.title,
            english_title = news_data.english_title,
            description = news_data.description,
            english_description = news_data.english_description,
        )
        news.save()
        return CreateNews(news=news)

class Mutation(graphene.ObjectType):
    create_news = CreateNews.Field()
```

Url pour une requête
tests dans le navigateur

1) Url en « clair »

```
http://127.0.0.1:5000/graphql?query=query { allNews(eventType: "ACTUALITE") { id
eventType title description publishDate url image imageUrl } }&raw
```

Ajouter **&raw** pour avoir du json, sinon on a le html de l'interface graphiql

Les caractères spéciaux peuvent poser problème (surtout le &)

2) Url encodée

```
http://127.0.0.1:5000/graphql?query=query%20%7B%20allNews(eventType:
%22ACTUALITE%22)%20%7B%20id%20eventType%20title%20description%20publishDate%20url%20
image%20imageUrl%20%7D%20%7D&raw
```

3) Url récupérée de graphiql

```
http://127.0.0.1:5000/graphql?
query=query%20%7B%0A%20%20allNews(eventType%3A%22ACTUALITE%22)%20%7B%0A%20%20%20%20i
d%0A%20%20%20%20eventType%0A%20%20%20%20title%0A%20%20%20%20description%0A%20%20%20%
20publishDate%0A%20%20%20%20url%0A%20%20%20%20image%0A%20%20%20%20imageUrl%0A%20%20%
7D%20%0A%7D&variables=null&operationName=undefined
```

Récupérer les données
tests en ligne de commande

```
query='query {
  allNews(eventType:"ACTUALITE") {
    id
    eventType
    title
    description
    publishDate
    url
    imageUrl
  }
}'
# remplace retour chariot par espace :
query="$(echo $query)
# Ecrit l'url en clair
echo http://127.0.0.1:5000/graphql?query="$query"\&raw

curl -H Content-Type: application/json" -X POST \
-d "{ \"query\": \"$curl_query\"}"
http://127.0.0.1:5000/graphql
# POST ou GET marchent. Attention il faut que les quotes soient echappées :
mettre /"ACTUALITE/"
# Pour utiliser une simple quote dans la requête : '"'"'

wget http://127.0.0.1:5000/graphql?query="$query" -O result.json
# Ecrit l'url encodée au passage
```

Bilan

- Graphene mongoengine : **Pas de filtres** sans l'utilisation de Relay, c'est relou. Pour avoir plus de fonctionnalités, d'exemples, de doc, viser django + postgres - ou basculer du coté obscur du code, en javascript

- Il faut se donner la peine de comprendre le principe du schema graphql pour surcharger le code - ça fait un modèle en plus.

- Les mutations ne me paraissent pas intéressantes pour ce projet

- Graphql ne remplace pas une API REST, c'est un autre outil

- Intéressant de tester MongoDB, mais difficile de se passer du SQL (ex : le UNION). OK pour des données peu liées.

Bilan GraphQL & Cie

- 😍 Principe de requête arborescente où on donne la structure qu’on veut
- 👍 GraphiQL
- 👍 Fonction auto descriptive
- 😡 Doc graphene-mongoengine
- 😕 Fonctionnalités graphene-mongoengine
- Pas abordé : les questions de cache et d’authentification

Conclusion

La quête du graal continue