

EFREI 2023-24

A propos de moi

- Jérôme Senot js@stairwage.com
- Co-fondateur & CTO de Stairwage
 Digitalisation des acomptes sur
 salaire & bien-être financier des
 salariés
 - www.stairwage.com
- Plusieurs API GraphQL sur
 React/Node avec Apollo depuis 2017



Objectifs

Pouvoir choisir d'utiliser ou non GraphQL

Pouvoir utiliser une API GraphQL côté front ou back

• Pouvoir développer une API GraphQL sûre et performante

Partis pris

- Mettre immédiatement en pratique les notions abordées
- Vous faire profiter au maximum de mon expérience pratique
- Découvrir une des technologies/outils les plus utilisées à chaque niveau :
 - VS Code
 - TypeScript
 - Node.js + npm/yarn
 - Express + Apollo Server
 - BDD NoSQL: MongoDB
 - React.js + Apollo Client

Organisation des séances

• J1:

- Introduction à GraphQL : intérêt, schéma, scalar types, opérations, union & interface, documents, introspection, resolvers
- Backend requêté avec sandbox Apollo

• J2:

- Utilisation avancée : subscriptions, custom types, directives, erreurs, authentification, versioning, stitching
 & federation
- Front connecté au backend & amélioration du backend

• J3:

- Robustesse, sécurité & performance : types, complexité, batch, data loaders, persisted queries, cache
- Améliorations front & back

• J4 :

- Matin: adaptation du code sur un projet personnel
- Après-midi : présentations individuelles des projets 5min/personne

Problème : comment échanger des données avec un serveur (API backend) ?

- Couche réseau « standard » : HTTP (voire WebSocket)
- Quelle URI ? Quelles entêtes ? Quel body de la requête ? De la réponse ?
- Historiquement API REST:
 - POST /collection/ + data
 - GET /collection/:id => data
 - PUT /collection/:id + data
 - PATCH /collection/:id + data
 - DELETE /collection/:id
 - ...

- Inconvénients:
 - Si je ne veux pas toutes les données lors d'un GET ?
 - Si je veux récupérer les données après un PATCH ?
 - Si je veux récupérer plusieurs éléments de collections différentes ?
 - Si j'ai besoin de récupérer un élément pour savoir ensuite quels autres récupérer ?
 - •
- Et si le client pouvait envoyer une requête complexe au serveur pour demander plusieurs ressources et les données précises dont il a besoin ?

Histoire

- « Graph Q(uery) L(anguage) » créé en 2012 par Facebook pour un usage interne afin de minimiser l'utilisation de la connexion réseau sur les mobiles (nombre et taille des requêtes)
- Publié en open-source en 2015
- Utilisé aujourd'hui par les plus grandes sociétés tech :
 - Meta
 - X
 - Netflix
 - •

GraphQL



https://graphql.org

Avantages & inconvénients

Avantages :

- Efficience réseau & globale
- Utilisation de l'API plus rapide
- Robustesse à l'ajout de nouveaux champs
- Vérification possible côté backend des données attendues par le front
- Schéma de spécification de l'API facilement exploitable (typage strict du code, vérification des types en runtime, documentation, auto-complétion...)

Inconvénients :

- Sécurité : requêtes complexes donc abus plus difficiles à détecter ?
- Cache : requêtes variées donc difficiles à mettre en cache ?

Schéma

- Architecture « schema first » :
 - 1. On conçoit un schéma
 - 2. Utilisable dans les outils de développement
 - 3. On l'implémente côté backend
 - 4. On l'utilise côté front

• Documentation:

https://graphql.org/learn/schema/

Syntaxe de base : typage des données émises

```
type AnObject {
optionnalField: ID
requiredField: ID!
optionnalArrayOfOptionnalIds: [ID]
requiredArrayOfOptionnalIds: [ID]!
requiredArrayOfRequiredIds: [ID!]!
optionnalArrayOfRequiredIds: [ID!]
anOtherObject: OtherObject
# Commentaire pour les développeurs
 Documentation de l'API
```

Enums

```
enum Direction {
   NORTH # ⚠ sérialisé en JSON sous forme de chaîne
   EAST
   SOUTH
   WEST
}

type Trajectory {
   direction: Direction!
   distance: Float!
}
```

Unions & interfaces

```
type Human {
  name: String!
}

type Computer {
  os: String!
}

union User = Human | Computer
```

```
interface Animal {
 type: String
type Dog implements Animal {
 type: String # 🛕 les champs de l'interface implémentée sont
obligatoires
 tailLength: Float
type Bird implements Animal {
 type: String
 wingsLength: Float
```

Opérations : requêtes disponibles

```
type Query {
 datelso: String!
 allAnimals: [Animal!]!
type Mutation {
 # 1 nommer les mutations comme des
fonctions
 incrementCounter: Int!
type Subscription {
 newAnimal: Animal!
```

```
schema {
  query: RootQueries
}

type RootQueries {
  dateIso: String!
  allAnimals: [Animal!]!
}
```

Paramètres

```
type Query {
# <u>h</u> on peut nommer les queries complexes comme
des fonctions
                                                String!
getFormattedDate(timezone: String!):
type Mutation {
addSomeone(age: Int!, weight: Float, name: Name!):
Person!
# / mot clé différent des données émises
input Name {
first: String!
last: String!
```

```
type Mutation {
 # / pour l'évolutivité et simplifier l'écriture des requêtes, on
préfère utiliser des types spécifiques en entrée et sortie des
opérations complexes
 addSomeone(input: AddSomeoneInput!):
         AddSomeonePayload!
input AddSomeoneInput {
 age: Int!
 weight: Float
 name: Name!
type AddSomeonePayload {
 newPerson: Person!
 newPersonsCount: Int! # Nouveau champ
```

Exemple

https://studio.apollographql.com/public/SpaceX-pxxbxen

Google « apollo graphql spacex »

Documents

```
query {
 capsules {
  reuse_count
 type
query {
 capsule(id: "59188bfb3b266b") {
 id
  reuse_count
  type
 company {
  ceo
```

```
mutation {
 addSomeone(input: {
  age: 23,
  weight: 83.7,
  name: { first: "John", last: "Doe" }
 }) {
  newPerson: {
   age
```

Aliases & fragments

```
query {
  capsule1: capsule(id: "59188bfb3") {
    id
     reuse_count
    type
  }
  capsule2: capsule(id: "dfg654564") {
    id
     reuse_count
     type
  }
}
```

```
fragment capsuleData on Capsule {
 id
 reuse_count
query {
 capsule1: capsule(id: " 59188bfb3 ") {
  ...capsuleData
 capsule2: capsule(id: " dfg654564 ") {
  ...capsuleData
  reuse_count
  type
```

Imbrication

```
query {
  books {
    title
    author {
     name
     age
     books {
     title
     }
  }
  }
}
```

```
query {
 books {
  title
  author {
   name
   age
   books {
    title
    author {
     books {
      author {
       books {
```

Contenu de la requête HTTP de type POST

Query body :

```
{
  "query": "fragment capsuleData on Capsule
{\n id\n reuse_count\n}\n\nquery {\n capsule\
  (id: \"5e9e2c5bf35918ed873b2664\")
{\n ...capsuleData\n reuse_count\n type\n \
  missions {\n flight\n name\n }\n }\n}\n\n",
}
```

Response body :

```
{
  "data": {
    "capsule": {
        "id": "5e9e2c5bf35918ed873b2664",
        "reuse_count": 0,
        "type": "Dragon 1.0",
        "missions": null
     }
}
```

Variables

Document GraphQL (texte) :

"id": "5e9e2c5bf35918ed873b2664"

• Query body (JSON):

- Plus simple et sûr que l'injection de valeurs dans le corps de la requête GraphQL
- L'utilisation d'une API GraphQL par un client est finalement assez simple!

__typename & introspection

Document GraphQL (texte):

```
query {
 capsule (id: "5e9e2c5bf35918ed873b2664") {
  id
  type
    typename
  • Réponse (JSON) :
 "capsule": {
  "id": "5e9e2c5bf35918ed873b2664",
  "type": "Dragon 1.0",
  " typename": "Capsule"
```

```
# A à désactiver si API privée
{
   __schema {
     queryType {
        name
        fields {
            name
        }
     }
}
```

Requêtes sur des unions & interfaces

```
type Human {
 id: ID!
 name: String!
type Computer {
 id: ID!
 os: String!
union User = Human | Computer
query {
 user {
  typename # pour connaître le type de l'objet
  # 🛕 champ id commun mais pas requêtable dans l'union User
  ...on Human {
   id
   name
  ...on Computer {
   id
   os
```

```
interface User {
 id: ID!
type Human implements User {
 id: ID!
 name: String!
type Computer implements User {
id: ID!
 os: String!
query {
 user {
  typename # pour connaître le type de l'objet
  ...on User {
   id # 🛕 champ id requêtable dans l'interface User
  ...on Human {
   name
  ...on Computer {
   os
```

Implémentation d'un serveur GraphQL

- Chargement du schéma
- Traitement d'une requête :
 - 1. Parsing du document
 - 2. Vérification du type et de la présence des variables
 - 3. Vérification de la validité du document par rapport au schéma
 - 4. Remplacement des fragments par leur contenu
 - 5. Parcours arborescent de la racine jusqu'aux types scalar pour calculer les valeurs à renvoyer à l'aide de « resolvers »
 - nais les mutations doivent être exécutées dans l'ordre
 - 6. Sérialisation JSON

Resolvers : définition

 Fonction côté backend qui pour un champ d'un type calcule la valeur à renvoyer au front

• Paramètres :

- Valeurs des autres champs déjà calculée de l'objet
- Valeurs des paramètres du champ
- Contexte : objet initialisé au début du traitement de la requête et passé par référence à chaque resolver

Valeur de retour :

- Valeur du champ de l'objet à renvoyer (avant sérialisation)
- Si le champ est de type objet :
 - La valeur de retour peut-être « deep partial » et un resolver sera alors appelé sur chaque sous-valeur absente
 - <u>A</u> La valeur doit aussi contenir le champ __typename si pas déductible automatiquement du schéma ni de la valeur

Resolvers: exemples

```
{
  Query: {
    film: async (obj, args, context) =>
      await db.loadFilmByID(args.id),
  },
  Film: {
    director: async (obj, args, context) => {
      if (context.ip !== "123.456.789.0") throw new Error('Forbidden');
      return await db.loadDirectorByID(obj.directorId)
      },
  },
}
```

Serveurs GraphQL

- Implémentation complexe qui nécessite d'utiliser une implémentation déjà existante qu'il ne restera qu'à configurer : schéma, resolvers...
- Plusieurs implémentations existent pour Node.js : https://graphql.org/code/#javascript
- Apollo:
 - Solution back & front développée par le Meteor Development Group
 - Très complète, bien maintenue et parmi les plus utilisées et suivies sur GitHub
 - Pas forcément la plus « légère »
- Des solutions cloud (moins flexibles) existent pour mapper directement une BDD sur une API

https://aws.amazon.com/fr/graphql/graphql-dynamodb-data-modeling/

A vos claviers!

• API de films :

- Films ou séries:
 - Film : titre, année, durée en minutes, personnages (nom, acteur), notes, note moyenne si au moins 1 note, nb de notes
 - Série : titre, année du 1^{er} épisode, nb d'épisodes, personnages (nom, acteur), notes, note moyenne, nb de notes
- Acteurs : prénom, nom de famille, films, séries, rôles (film, personnage)
- Utilisateurs : pseudo, *notes*, *nb de notes*
- Notes anonyme ou pas : film, valeur 1-5, utilisateur si pas anonyme

Fonctionnalités :

- Récupérer les données stockées et *calculées*
- Ajouter/modifier/supprimer un élément (suppression en cascade)
- Pour de vraies données, cf https://www.imdb.com/chart/top/

Environnement

- VS Code avec extensions :
 - Biome
 - GraphQL: Language Feature
 - GraphQL: Syntax Highlighting
- Node.js en ESM avec yarn (ou npm)
- Modules:
 - @biomejs/biome
 - @graphql-codegen/cli
 - @graphql-codegen/typescript
 - @graphql-codegen/typescript-resolvers
 - @types/node
 - ts-node
 - tsx
 - typescript

- Configuration (cf doc de chaque outil) :
 - Biome:
 - .vscode/settings.json
 - biome.json
 - TypeScript: tsconfig.json dans chaque sous-projet (cf slide suivant)
 - Codegen : codegen.yml
 - package.json:

```
{
  "private": true,
  "type": "module",
  "scripts": {
    "generate": "graphql-codegen --config codegen.yml",
    "serve": "tsx watch back/src"
  },
    ...
}
```

Structure & tsconfig.json

```
/back/srcindex.ts...tsconfig.json/db
```

- /graphql
 - /schema
 - /src
 - tsconfig.json
- codegen.yml

```
•
```

```
"compilerOptions": {
 "lib": ["es2020"],
 "target": "es2020",
 "module": "NodeNext",
 "moduleResolution": "NodeNext",
 "esModuleInterop": false,
🛕 que pour back, pas [] pour graphql
 "types": ["node"],
 "noEmit": true,
 "allowImportingTsExtensions": true
```

Etapes

- Suivre doc Apollo Server avec leur exemple de données
- Lancer et tester avec l'URL locale qui sert une sandbox
- Extraire schéma dans /graphql/schema/schema.gql et générer resolvers.ts dans /graphql/src avec codegen et aussi schema.ts avec plugin custom réutilisant '@graphqlcodegen/schema-ast, puis utiliser les fichiers générés :
 - Importer typeDefs de schema.ts et l'utiliser pour démarrer le serveur Apollo
 - 2. Importer le type Resolvers de resolvers.ts et l'utiliser pour typer la variable resolvers const resolvers: Resolvers = ...
- 4. Lancer serveur MongoDB au démarrage avec mongodb-memory-server {instance: {dbPath: './db', storageEngine: 'wiredTiger' }}
- 5. Brancher les resolvers sur la BDD et ajouter mutation pour ajouter un objet :
 - 1. Installer le module mongodb
 - Se connecter au serveur local lancé par mongodb-memory-server (cf doc du driver mongodb) const client = await (new MongoClient(uri).connect());
 - 3. Récupérer la collection const books = client.db('main').collection<Book>('books')
 - 4. L'utiliser dans les resolvers await books.find({}).toArray()
 - 5. Injecter des données avec l'outil Compass de MongoDB ou implémenter une mutation pour insérer des données await books.insertOne({ title: 'Titre du livre', author: 'John Doe' })
- 6. Implémenter l'API cible :
 - Type TS des collections (sans _id)
 - 2. Queries données stockées
 - Mutations pour insérer des données
 - 4. Queries plus complexes
 - 5. Mutation pour modifier des données
 - 6. Mutation pour supprimer des données en cascade

Config codegen

```
schema: "./graphql/schema/schema.gql"
emitLegacyCommonJSImports: false
require:
 - ts-node/register
generates:
  ./graphql/src/resolvers.ts:
  plugins:
   - "typescript"
   - "typescript-resolvers"
  config:
   useTypeImports: true
   useIndexSignature: true
 ./graphql/src/schema.ts:
  plugins:
   - "./graphql/exportSchemaCodegenPlugin.cts"
```

```
// /graphql/exportSchemaCodegenPlugin.cts
import type { CodegenPlugin } from '@graphql-
codegen/plugin-helpers';
import { plugin } from '@graphql-codegen/schema-ast';

module.exports = {
  async plugin(schema, documents, config) {
    const res = await plugin(schema, documents, config);
    return 'export const typeDefs = `#graphql\n' + res + '`;';
  },
} satisfies CodegenPlugin;
```