✓ Terminé

title: Neo4J + GraphQL

Neo4J utilise GraphQL pour réaliser très simplement des API CRUD avec un minimum de code à écrire (voir également Dgraph).

Rappels sur `GraphQL`

`GraphQL` est une spécification qui sert à deux choses :

- préciser le schéma d'un système d'information

```
type Todo{
  id: ID!
  content: String
  belongsTo: TodoList
  done: Boolean
}

type TodoList{
  title: String
  owner: User
}

type User{
  id: ID @id
  username: String!
  password: String!
  roles: [String!]
}
```

- formaliser les notions requêtes : `query` et `mutation` :

```
{
  todoLists {
    title
    owner {
      username
    }
  }
}
```

```
mutation {
  createUsers(
    input: { username: "admin", password: "rootroot", roles: ["admin"] }
  ) {
    users {
      id
      username
      roles
    }
  }
}
```

Pour être plus précis, `GraphQL` propose d'enrichir ses requêtes à l'aide

- de variables

?

```
{
  "query": "mutation($json:JSON){
         saveToDev(serial:$json)}",
  "variables":{
    "json":{
      "id":"toti"
    }
  }
}
```

- et d'un entête HTTP.

```
{
  "authorization": "Bearer eyJ0eXAiOi..."
}
```

Il est important de bien noter que `GraphQL` n'est qu'une spécification et en aucun cas une implémentation. Pour des implémentations dans les différents langages, on trouvera son bonheur dans https://graphql.org/code/.

Notion de résolveur

Une notion centrale est celle de résolveur, qui réalise l'implémentation concrète entre la requête `GraphQL` et le point d'accès à la base de données.

Par exemple, si on utilise Apollo GraphQL pour produire un serveur en Javascript, on aura des résolveurs comme suit :

```
const resolvers = {
    Mutation:{
    insertClient: (root, args, context) => {
        return client.query("INSERT INTO client(nom) VALUES('" + args.nom + "') RETURNING ID")
        .then(res => res.rows[0].id)
        .catch(err => {
            console.log(err.stack);
        });
    }
    },
    Query: {
    clients: (root, args, context) => {
            return client.query('SELECT * from client')
        .then(res => res.rows)
        .catch(err => {
                    console.log(err.stack);
        })
    }
    }
};
```

Ces résolveurs sont équivalents aux *repository* de `Doctrine`, l'ORM de Symfony. L'écriture de résolveurs est une tâche classique du développement d'application : pour gérer la partie Modèle du MVC, le développeur prépare à l'avance toutes les requêtes nécessaires. Il s'agit d'un travail d'anticipation fastidieux.

Limites de GraphQL

Écrire tous les résolveurs est fastidieux : il faut prévoir tous les cas d'usages du CRUD (create, read, update, delete) et les paramètres des requêtes :

- passage de paramètres pour les mutations ou les recherches, entre autres pour suivre les relations
- tri des résultats
- fournir toutes les mutations utiles au CRUD
- gestion de l'authentification avec le jeton
- pagination
- gestion des droits d'accès

Le couple `Neo4j` + `graphQL`

`Neo4j` est un gestionnaire de données graphiques, c'est-à-dire de données représentées par des graphes. Les noeuds sont les entités, les arcs sont les relations. Il s'agit d'une façon moderne d'envisager la gestion d'un système d'information, qui gèrera nativement la sémantique.

Associé à `graphQL`, `neo4j` devient particulièrement puissant puisqu'il permet de s'affranchir de tous les éléments fastidieux liés à la rédaction des résolveurs. Grâce à quelques annotations dans le schéma (du type des annotations `Doctrine`), toutes les limitations évoquées ci-dessus sont levées.

?

```
type Todo{
  id: ID! @id
  content: String
  belongsTo: TodoList @relationship(type: "BELONGS_TO", direction: OUT)
  done: Boolean @default(value: false)
}

type TodoList{
  id: ID @id
  title: String
  owner: User @relationship(type: "OWNED_BY", direction: OUT)
}

type User{
  id: ID @id
  username: String!
  password: String! @private
  roles: [String!]
}

type Mutation {
  signUp(username: String!, password: String!): String! ### JWT
  signIn(username: String!, password: String!): String! ### JWT
}
```

Le code du serveur est particulièrement léger :

```
const { Neo4jGraphQL } = require("@neo4j/graphql");
const { ApolloServer } = require("apollo-server");
const neo4j = require("neo4j-driver");
const fs = require("fs");
const dotenv = require("dotenv");
const path = require("path");

// Load contents of .env as environment variables
dotenv.config({path: __dirname + '/.env'});

// Load GraphQL type definitions from schema.graphql file
const typeDefs = fs
  .readFileSync(path.join(__dirname, "todo.graphql"))
  .toString("utf-8");

// Create Neo4j driver instance
const driver = neo4j.driver(
  process.env.NEO4J_URI,
  neo4j.auth.basic(process.env.NEO4J_USER, process.env.NEO4J_PASSWORD)
);

// Create executable GraphQL schema from GraphQL type definitions,
// using @neo4j/graphql to autogenerate resolvers
const neoSchema = new Neo4jGraphQL({
    typeDefs,
});

// Create ApolloServer instance that will serve GraphQL schema created above
// Inject Neo4j driver instance into the context object, which will be passed
//  into each (autogenerated) resolver
const server = new ApolloServer({
  context:({req}) => ({ driver, req }),
  schema: neoSchema.schema
});

// Start ApolloServer
server.listen().then(({ url }) => {
  console.log(`GraphQL server ready at ${url}`);
});
```

Lorsque l'on accède avec un navigateur à l'URL du serveur (voir par exemple http://graphql.unicaen.fr:4000), on constate en examinant la documentation que tous les résolveurs nécessaires au CRUD pour les entités du schéma sont directement disponibles sans développement additionnel.

Gestion de l'authentification

Lors de la réalisation d'une API CRUD, il n'est pas toujours possible de compter sur la disponibilité d'un *coockie* d'authentification. Il faut alo ? tourner vers une autre technologie : les *jetons* d'authentification.

Ces jetons sont fournis par le serveur, qui va hacher un ensemble d'informations à l'aide d'une clé secrète. L'utilisateur récupère ce jeton lors de son inscription ou de sa connexion, et doit le transmettre lors de chaque opération pour s'authentifier. Lorsque le serveur reçoit un jeton, il lui est facile de le dé-hacher pour obtenir les informations sur l'utilisateur.

Nous utilisons ici une technologie particulière de jeton : JWT (JSON web token). Lors de la connexion, on récupère le jeton :

```
mutation {
  signIn(username: "admin", password: "rootroot")
}
```

```
{
  "data": {
    "signIn":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIxZmNlODk2Yy0zNmJiLTQyZjQtYWFiMi0xNGZiMjMxZjkwMzEiLCJyb2xlcyI6WyJhZG1pbiJdLCJqdGkiOiIyODVhNjZiYS1mOGE5LTQwYTUtOWU3ZC1iMjg5ZmE4YWVjYmQiLCJpYXQiOjE2MzQ4MzA2Mjd9.3O3gDRu5FUqOKclwsSpEtWoyP0PBm3t2YbkKZ6oN5j8"
  }
}
```

Pour utiliser le jeton, il faut ajouter un entête HTTP à la requête `GraphQL`, dont la clé est `authorization` et la valeur est `"Bearer " + <jeton>`\ :

```
{
  "authorization": "Bearer eyJ0eXAi...j8"
}
```

Noter que le jeton est en trois parties, séparées par des points. Les deux premières consistent en un encodage `base64` des données et ne sont pas cryptées (il ne faut donc pas y mettre de données sensibles), la dernière est une signature par la clé, qui certifie que le jeton n'a pu être fourni que par le serveur. On peut décoder le token à l'aide de https://jwt.io/ (secret: dFt8QaYykR6PauvxcyKVXKauxvQuWQTc)

- header (type de token + algorithme de hachage)

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

- payload (la donnée véhiculée)
  - sub (l'id du souscripteur)
  - roles
  - jti (un id tiré au hasard)
  - iat (la date à partir de laquelle le jeton est valide)
  - exp (la date jusqu'à laquelle le jeton est valide)

```
{
  "sub": "1fce896c-36bb-42f4-aab2-14fb231f9031",
  "roles": [
    "admin"
  ],
  "jti": "285a66ba-f8a9-40a5-9e7d-b289fa8aecbd",
  "iat": 1634830627
}
```

TP

L'objectif principal de ce TP est de devenir à l'aise avec l'API GraphQL qui expose la base de données de Neo4J. Cette base de données sera réutilisée comme mémoire permanente de votre projet en développement d'application mobile. Il vous faut donc arriver à effectuer toutes les requêtes du CRUD sur les entités de la base. Les mutations doivent utiliser des paramètres pour être utilisées par la librairie `fetch` en Javascript.
Des détails ici

- à l'aide du navigateur, se rendre sur le serveur <graphql.unicaen.fr:4000>. Ce serveur fournit un connecteur `graphql` sur une base de données `Neo4j` avec le schéma fourni précédemment.
- découvrir la documentation et le schéma complet généré par l'application
- tester quelques requêtes du CRUD. En particulier vous devez être suffisamment agile en `graphQL` pour effectuer des mutations d'insertions de toutes les entités. N'hésitez pas à utiliser l'interface en mode *cliquodrome*
- entre autres, vous devez arriver à effectuer des mutations avec paramètres, où les arguments sont passés dans la section `Query Variables`.
- notez bien qu'une mutation est censée renvoyer quelque chose, ce qui explique qu'elle finit par une partie analogue à une requête.
- cet entraînement que vous pratiquez ici vous sera utile pour le cours `Techno web, développement d'applications client` où vous aurez à développer une application `React` exécutant des requêtes `graphQL`.

Création d'utilisateur

- tester les mutation `SignUp` et `SignIn` en vérifiant les tokens obtenus (en utilisant https://jwt.io/)
- créer deux utilisateurs (signup) et tester le bon fonctionnement de l'authentification
- à l'aide de `Postman`, déclencher une série de tests convaincants. Pour cela, organiser les tests de façon hiérarchique à l'aide de dossiers, qui

?

permettent de définir une autorisation par token dont hériteront les requêtes du dossier.

Programmation des requêtes en Javascript

En utilisant le serveur http://graphql.unicaen.fr:4000, définissez en Javascript *toutes* les requêtes du CRUD. Un canevas est disponible à https://ecampus.unicaen.fr/mod/resource/view.php?id=1006308. Votre programme principal doit être comme suit\ :

```
import { signIn } from "./sign.js"

import { createTodoList } from "./todoList.js";
import { createTodo } from "./todo.js";

const username = "toto";
const password = "rootroot";

async function run() {
    let token = '';
    await signIn(username, password)
        .then(t => { token = t })
        .catch(console.error)

    const title = "liste pour toto";
    let todoListId = '';
    await createTodoList(username, title, token)
        .then(todoList => { todoListId = todoList.id })
        .catch(console.error)

    const content = "todo item 1";
    let todoId = '';
    await createTodo(content, todoListId, token)
        .then(todo => {todoId = todo.id})
        .catch(console.error)

    console.log("token", token)
    console.log("todoListId", todoListId)
    console.log("todoId", todoId)
}

run()
```

Utilisation de `postman`

- lancez `postman` et organisez un série de tests de l'API
- créer une collection et lui ajouter des requêtes
- utiliser l'onglet `body` pour insérer des requêtes GraphQL
- les `query` peuvent être en GET si vous mettez l'instruction `graphQL` dans l'URL mais doivent être en POST si vous paramétrez un body `graphQL`. Les mutations doivent être en POST (utilisez la section `graphQL` du body et la section *variables*)
- testez à l'unité vos requêtes dans le navigateur à l'adresse http://graphql.unicaen.fr:4000 et déposez les dans `postman` une fois mises au point
- pour les requêtes dont le résultat doit être testé, utiliser "Response body is equal to a string". N'oubliez pas d'ajouter un `\n` à la fin de la chaîne.
- prêtez attention à l'ordre dans les requêtes
- quand vous éditez une requête, n'oubliez pas de la sauvegarder.
- il est possible d'exécuter l'intégralité d'une collection (*run*)

Évitez d'utiliser l'utilisateur `toto` pour vos tests. Il est réservé pour mes tests. Réservez un utilisateur de votre choix pour les tests.

Une correction est disponible sous forme de fichier JSON.

La collection des tests est également disponible ici, il s'agit d'un lien de partage d'API importable dans une collection `Postman`.

Installation d'un serveur `neo4j + graphQL` 4.4 sur Linux Debian/Ubuntu

Cette section relate les opérations que j'ai effectuées pour installer le serveur http://graphql.unicaen.fr:4000 et la machine virtuelle. Vous n'êtes pas censé effectuer vous-même ces opérations, sauf si vous souhaitez installer le serveur chez vous ou sur une machine virtuelle.

Inspiré de https://neo4j.com/docs/operations-manual/4.4/installation/linux/debian/

Attention :

- il faut une version 11 du java runtime
- `neo4j` est la base de données, `cypher-shell` est le shell de connexion. Les deux doivent être compatibles, ici en version 4.4

La procédure décrite dans la documentation est OK mais il faut préciser la version correcte de `cypher-shell`\ :

?

```
# télécharger la clé apt de neo4j
wget -O - https://debian.neo4j.com/neotechnology.gpg.key | sudo apt-key add -
# ajouter la source neo4j
echo 'deb https://debian.neo4j.com stable latest' | sudo tee -a /etc/apt/sources.list.d/neo4j.list
sudo apt-get update

# vérifier la disponibilité de cypher-shell et neo4j
apt list -a cypher-shell neo4j
# lancer l'installation des paquets
sudo apt-get install cypher-shell=1:4.4.24 neo4j=1:4.4.24
# Les NOUVEAUX paquets suivants seront installés :
#  ca-certificates-java cypher-shell daemon fontconfig-config fonts-dejavu-core java-common libasound2
#  libasound2-data libavahi-client3 libavahi-common-data libavahi-common3 libcups2 libfontconfig1 libgraphite2-3
#  libharfbuzz0b libjpeg62-turbo liblcms2-2 libnspr4 libnss3 libpcsclite1 neo4j openjdk-11-jre-headless
```

Il faut maintenant activer le plug-in `apoc` (qui doit avoir le même numéro de version que `neo4j`), permettant de gérer les droits d'accès\ :

```
cd /var/lib/neo4j
sudo cp labs/apoc-4.4.0.20-core.jar plugins/
```

Tout est en place, on peut lancer la base de données et s'y connecter une première fois, ce qui déclenche la modification du mot de passe initial\ :

```
sudo neo4j start
# on vérifie que la base est lancée
neo4j status
# connexion initiale. login: neo4j, password: neo4j
cypher-shell
username: neo4j
password:
Password change required
new password:
confirm password:
Connected to Neo4j using Bolt protocol version 4.4 at neo4j://localhost:7687 as user neo4j.
Type :help for a list of available commands or :exit to exit the shell.
Note that Cypher queries must end with a semicolon.
```

Noter que `cypher-shell` se connecte par le biais du protocole `Bolt` à l'adresse <neo4j://localhost:7687>. Une interface d'administration est disponible à .

Gestion du service **neo4j**

- consultation des logs du service : `sudo journalctl -e -u neo4j`
- activation du service : `sudo systemctl enable neo4j`

Conception d'un service pour le serveur **node**

D'après https://nodesource.com/blog/running-your-node-js-app-with-systemd-part-1/

- éditer le fichier de configuration du service

  ```
  sudo jed /lib/systemd/system/todo.service
  ```

- le fichier est le suivant\ :

```
[Unit]
Description=serveur Neo4j / GraphQL
Documentation=https://example.com
After=neo4j.service

[Service]
# Environment=NODE_PORT=4000
Type=simple
User=rioultf
ExecStart=/usr/bin/node /home/rioultf/neo4j/todoServer/index.js
Restart=always

[Install]
WantedBy=multi-user.target
```

- noter la dépendance au service `neo4j`
- pour que `dotenv` charge correctement le `.env`, il aura fallu le paramétrer correctement\ :

  ```
  dotenv.config({path: __dirname + '/.env'});
  ```

  ?

- redémarrer le démon des services

```
sudo systemctl daemon-reload
```

- lancer le service

  sudo systemctl start todo

- activer le service

```
sudo systemctl enable todo.service
```

- obtenir le statut du service

  sudo systemctl status todo.service

- consulter les logs

  sudo journalctl -e -u neo4j

Liens

- refcard cypher
- https://github.com/neo4j/graphql/blob/master/examples/neo-push/server/src/gql/User.ts
- https://neo4j.com/docs/graphql-manual/current/ogm/examples/custom-resolvers/
- https://neo4j.com/docs/graphql-manual/current/guides/v4-migration/authorization/

Exemples de requêtes cypher

```
match (this) detach delete this
match (this) return this
MATCH (t:Todo) DETACH DELETE t
match (u:User) return u.id;
```

Modifié le: lundi 30 octobre 2023, 16:11

◄ TP 2.5 - Test d'une API avec Postman

Choisir un élément

Aller à...

Canevas pour API GraphQL/Neo4J en Javascript ►