Spring Boot : services web **GraphQL**

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Spring Boot

Service web (WS pour Web Service) ?

- Un programme (ensemble de fonctionnalités exposées en temps réel et sans intervention humaine)
- Accessible via internet, Intranet, réseaux privés...
- Indépendant de tout système d'exploitation
- Indépendant de tout langage de programmation
- Utilisant un système standard d'échange (**XML** ou **JSON**), ces messages sont généralement transportés par des protocoles internet connus **HTTP** (ou autres comme **FTP**, **SMTP**...)
- Pouvant communiquer avec d'autres WS

Spring Boot

Les WS peuvent utiliser les technologies web suivantes :

- **HTTP** (Hypertext Transfer Protocol) : le protocole, connu, utilisé par le World Wide Web et inventé par Roy Fiedling.
- **REST** (Representational State Transfer) : une architecture de services Web, créée aussi par Roy Fielding en 2000 dans sa thèse de doctorat.
- **SOAP** (Simple object Access Protocol) : un protocole, défini par Microsoft et IBM ensuite standarisé par **W3C**, permettant la transmission de messages entre objets distants (physiquement distribués).
- **WSDL** (Web Services Description Language) : est un langage de description de service web utilisant le format **XML** (standardisé par le **W3C** depuis 2007).
- **UDDI** (Universal Description, Discovery and Integration) : un annuaire de WS.

Spring Boot

GraphQL

- Langage de requête
- Créé par **Facebook** en 2012 puis rendu open-source depuis 2015
- Respectant l'architecture client/serveur
- Alternative aux **API REST**
- Permettant aux clients de spécifier exactement quelles données ils souhaitent recevoir dans une seule requête $\Rightarrow$ réduction de sur-fetching/sous-fetching de données

Spring Boot

Création de projet **Spring Boot**

- Aller dans `File > New > Other`
- Chercher `Spring`, dans `Spring Boot` sélectionner `Spring Starter Project` et cliquer sur `Next >`
- Saisir
  - `spring-graphql` dans `Name`,
  - `com.example` dans `Group`,
  - `spring-graphql` dans `Artifact`
  - `com.example.demo` dans `Package`
- Cliquer sur `Next`
- Chercher et cocher les cases correspondantes aux `Spring Data JPA`, `MySQL Driver`, `Lombok`, `Spring Web`, `Spring for GraphQL` et `Spring Boot DevTools`
- Cliquer sur `Next` puis sur `Finish`

Spring Boot

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Spring Boot

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Pour la suite, nous considérons

- une entité `Personne` à définir dans `com.example.demo.model`
- une interface DAO `PersonneRepository` à créer dans `com.example.demo.dao`
- un contrôleur `PersonneGraphQlController` à créer dans `com.example.demo.controller`

Spring Boot & REST

Créons une entité `Personne` **dans** `com.example.demo.model`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long num;
    @NonNull
    private String nom;
    @NonNull
    private String prenom;
    @NonNull
    @ManyToMany(cascade = CascadeType.ALL)
    private List<Adresse> adresses;
}
```

Spring Boot & REST

Et une entité `Adresse`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Adresse {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @NonNull
    private String rue;
    @NonNull
    private String codePostal;
    @NonNull
    private String ville;
    @ManyToMany(mappedBy = "adresses")
    private List<Personne> personnes = new ArrayList<Personne>();
}
```

Spring Boot & REST

Préparons notre interface DAO `PersonneRepository`

```
package com.example.demo.dao;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository<Personne, Long> {

}
```

Spring Boot & REST

Préparons notre interface DAO `PersonneRepository`

```
package com.example.demo.dao;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository<Personne, Long> {

}
```

Et une interface `AdresseRepositoty`

```
package com.example.demo.dao;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Adresse;

public interface AdresseRepositoty extends JpaRepository<Adresse, Long> {

}
```

Spring Boot

Dans `application.properties`, ajoutons les données permettant la connexion à la base de données et la configuration de `Hibernate`

```
spring.datasource.url=jdbc:mysql://localhost:3306/cours_graphql?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Spring Boot

Dans `application.properties`, ajoutons les données permettant la connexion à la base de données et la configuration de `Hibernate`

```
spring.datasource.url=jdbc:mysql://localhost:3306/cours_graphql?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```


Ajoutons aussi une propriété pour activé l'interface GraphQL qui nous permettra d'envoyer les requêtes au serveur

```
spring.graphql.graphiql.enabled=true
```

Spring Boot

Préparons la classe de démarrage pour alimenter la base de données avec de données de test

```
@AllArgsConstructor
@SpringBootApplication
public class SpringQraphqlApplication implements ApplicationRunner {

    private PersonneRepository personneRepository;

    public static void main(String[] args) {
        SpringApplication.run(SpringQraphqlApplication.class, args);
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {

        Adresse a1 = new Adresse("paradis", "Marseille", "13000");
        Adresse a2 = new Adresse("plantes", "Paris", "75000");
        Adresse a3 = new Adresse("Salengro", "Marseille", "13000");

        Personne personne1 = new Personne("wick", "john", List.of(a1));
        Personne personne2 = new Personne("dalton", "jack", List.of(a2, a3));

        personneRepository.save(personne1);
        personneRepository.save(personne2);
    }
}
```

Spring Boot

Créons le contrôleur `PersonneGraphQlController` **dans lequel nous injectons** `PersonneRepository`

```
@Controller
@AllArgsConstructor
public class PersonneGraphQlController {

    private PersonneRepository personneRepository;

}
```

Spring Boot

Préparons la méthode suivante qui permet de retourner la liste des personnes stockées dans la base de données

```
@Controller
@AllArgsConstructor
public class PersonneGraphQlController {

    private PersonneRepository personneRepository;

    @QueryMapping
    List<Personne> personnes() {
        return personneRepository.findAll();
    }
}
```

Spring Boot

Explication

- `@QueryMapping` : utilisée pour mapper une méthode qui gère une opération de lecture (query) dans **GraphQL**
- `@MutationMapping` : utilisée pour mapper une méthode qui gère une opération d'écriture (mutation) dans **GraphQL**

Spring Boot

Créons un fichier `schema.gtaphqls` **dans** `graphql` **de** `src/main/resources`

```
type Query {
        personnes: [Personne]
}


type Personne {
        num: Int,
        nom: String,
        prenom: String
}
```

Spring Boot

Explication

- `type Query` : permet de définir une requête **GraphQL**.
- `personnes: [Personne]` : signifie que la requête `personnes` retournera une liste ([]) d'objets de type `Personne`.
- `type Personne` : décrit le type d'objet `Personne` à retourner.
- En **GraphQL**, les types commencent généralement par une majuscule `Int`, `String`, `Float`...

Spring Boot

Pour tester le Web Service, il faut

1. lancer l'application,
2. aller à `http://localhost:8080/graphiql?path=/graphql`,
3. saisir la requête suivante :

```
query {
  personnes { nom, prenom }
}
```

4. envoyer la requête et vérifier qu'on récupère les données relatives aux colonnes demandées (`nom` et `prenom`) au format **JSON**.

Spring Boot

Pour tester depuis **Postman**, il faut

- saisir l'URL `http://localhost:8080/graphql`,
- choisir la méthode `POST`,
- cocher `raw` et sélectionner `JSON`,
- ajouter l'objet **JSON** suivant

```
{
  "query": "{ personnes {  nom prenom } }"
}
```

- envoyer la requête et vérifier qu'on récupère les données relatives aux colonnes demandées (`nom` et `prenom`) au format **JSON**.

Spring Boot

Modifions le fichier `schema.gtaphqls` pour récupérer les adresses

```
type Query {
        personnes: [Personne]
}

type Personne {
        num: Int,
        nom: String,
        prenom: String,
        adresses : [Adresse]
}

 type Adresse {
    id: Int,
    rue: String,
    ville: String,
    codePostal: String
 }
```

Spring Boot

Pour tester le Web Service, il faut

1. lancer l'application,
2. aller à `http://localhost:8080/graphiql?path=/graphql`,
3. saisir la requête suivante :

```
query {
  personnes {nom, prenom, adresses  {rue}}
}
```

4. envoyer la requête et vérifier qu'on récupère les données relatives aux colonnes demandées au format **JSON**.

Spring Boot

Ajoutons une méthode permettant de retourner une seule Personne selon l'identifiant (num)

```
@Controller
@AllArgsConstructor
public class PersonneGraphQlController {

    private PersonneRepository personneRepository;

    @QueryMapping
    List<Personne> personnes() {
        return personneRepository.findAll();
    }

    @QueryMapping
    Personne personneById(@Argument Long id) {
        return personneRepository.findById(id).orElseThrow(
                () -> new NotFoundException(id, "Personne"));
    }
}
```

Spring Boot

Définissons `NotFoundException`

```
package com.example.demo.exception;

public class NotFoundException extends RuntimeException {

    public NotFoundException(Long id, String type) {
        super(String.format("%s avec l'identifiant %s n'existe pas", type, id));
    }

}
```

Spring Boot

Mettons à jour le fichier `schema.gtaphqls`

```
type Query {
        personnes: [Personne]
        personneById(id: Int): Personne
}

type Personne {
        num: Int,
        nom: String,
        prenom: String,
        adresses : [Adresse]
}

 type Adresse {
    id: Int,
    rue: String,
    ville: String,
    codePostal: String
 }
```

Spring Boot

Pour tester le Web Service, il faut

1. lancer l'application,
2. aller à `http://localhost:8080/graphiql?path=/graphql`,
3. saisir la requête suivante :

```
query {
  personneById(id: 2) { nom, prenom }
}
```

4. envoyer la requête et vérifier qu'on récupère les données relatives aux colonnes demandées (`nom` et `prenom`) au format **JSON**.

Spring Boot

Pour tester le Web Service, il faut

1. lancer l'application,
2. aller à `http://localhost:8080/graphiql?path=/graphql`,
3. saisir la requête suivante :

```
query {
  personneById(id: 5) { nom, prenom }
}
```

4. envoyer la requête et vérifier que le message d'erreur qu'on a définie pour les personnes inexistantes ne s'affiche pas.

Spring Boot

Créons la classe `GraphQLExceptionHandler` pour permettre d'afficher les messages d'erreur personnalisés

```
package com.example.demo.config;

import org.springframework.graphql.execution.DataFetcherExceptionResolverAdapter;

public class GraphQLExceptionHandler extends DataFetcherExceptionResolverAdapter {


}
```

Spring Boot

Implémentons la méthode `resolveToSingleError` **de la classe** `DataFetcherExceptionResolverAdapter`

```
package com.example.demo.config;

import org.springframework.graphql.execution.DataFetcherExceptionResolverAdapter;
import org.springframework.stereotype.Component;

import graphql.GraphQLError;
import graphql.schema.DataFetchingEnvironment;

@Component
public class GraphQLExceptionHandler extends DataFetcherExceptionResolverAdapter {

    @Override
    protected GraphQLError resolveToSingleError(Throwable ex, DataFetchingEnvironment env) {
        return super.resolveToSingleError(ex, env);
    }

}
```

Spring Boot

Modifions le code de la méthode `resolveToSingleError`

```
package com.example.demo.config;

import org.springframework.graphql.execution.DataFetcherExceptionResolverAdapter;
import org.springframework.stereotype.Component;

import com.example.demo.exception.NotFoundException;

import graphql.ErrorType;
import graphql.GraphQLError;
import graphql.GraphqlErrorBuilder;
import graphql.schema.DataFetchingEnvironment;

@Component
public class GraphQLExceptionHandler extends DataFetcherExceptionResolverAdapter {

    @Override
    protected GraphQLError resolveToSingleError(Throwable ex, DataFetchingEnvironment env) {
        if (ex instanceof NotFoundException) {
            return toGraphQLError(ex);
        } else {
            super.resolveToSingleError(ex, env);
        }
    }

    private GraphQLError toGraphQLError(Throwable ex) {
        return GraphqlErrorBuilder.newError().message(ex.getMessage()).errorType(ErrorType.
          DataFetchingException).build();
    }
}
```

Spring Boot

Pour tester le Web Service, il faut

1. lancer l'application,
2. aller à `http://localhost:8080/graphiql?path=/graphql`,
3. saisir la requête suivante :

```
query {
  personneById(id: 5) { nom, prenom }
}
```

4. envoyer la requête et vérifier que le message `"Personne ayant l'identifiant 5 n'existe pas"` s'affiche dans la réponse.

Spring Boot

Préparons la méthode suivante qui permet d'ajouter une nouvelle `Personne` dans la base de données

```
@Controller
@AllArgsConstructor
public class PersonneGraphQlController {

    private PersonneRepository personneRepository;

    @QueryMapping
    List<Personne> personnes() {
        return personneRepository.findAll();
    }

    @QueryMapping
    Personne personneById(@Argument Long id) {
        return personneRepository.findById(id).orElseThrow(
                () -> new NotFoundException(id, "Personne"));
    }

    @MutationMapping
    Personne addPersonne(@Argument Personne personne) {
        return personneRepository.save(personne);
    }
}
```

Spring Boot

Déclarons `addPersonne` **dans** `schema.gtaphqls` **: Le paramètre d'entrée doit être déclaré avec le mot-clé** `input`

```
type Mutation {
        addPersonne(personne: PersonneRequest): Personne
}

type Personne {
        num: Int,
        nom: String,
        prenom: String,
        adresses : [Adresse]
}

input PersonneRequest {
        nom: String,
        prenom: String,
}
```

Spring Boot

Pour tester le Web Service, il faut

1. lancer l'application,
2. aller à `http://localhost:8080/graphiql?path=/graphql`,
3. saisir la requête suivante :

```
mutation {
  addPersonne(personne: {
    nom: "Maggio",
    prenom: "Sophie"
  }) {
    num, nom, prenom
  }
}
```

4. envoyer la requête et vérifier qu'on ajoute et récupère les données relatives aux colonnes demandées (`num`, `nom` et `prenom`) au format **JSON**.

Spring Boot

Notion de variable dans **GraphQL**

- Utilisée pour passer des valeurs dynamiques aux requêtes **GraphQL**,
- Préfixée par le symbole `$` dans la définition d'une **Query** ou d'une **Mutation**,
- Permettant de séparer les valeurs des arguments de la requête elle-même.

Spring Boot

Pour tester le Web Service, il faut

1. saisir la requête suivante :

```
mutation($nom: String, $prenom: String) {
  addPersonne(personne: {
    nom: $nom,
    prenom: $prenom
  }) {
    num, nom, prenom
  }
}
```

2. saisir les variables :

```
{"nom": "Linus", "prenom": "Benjamin" }
```

3. envoyer la requête et vérifier qu'on ajoute et récupère les données relatives aux colonnes demandées (`num`, `nom` et `prenom`) au format **JSON**.

Spring Boot

Pour tester le Web Service, il faut

1. saisir la requête suivante :

```
query($id: Int) {
  personneById(id: $id) { nom, prenom }
}
```

2. saisir les variables :

```
{"id": 2}
```

3. envoyer la requête et vérifier qu'on récupère les données relatives aux colonnes demandées (`nom` et `prenom`) de la personne ayant l'identifiant 2 au format **JSON**.

Spring Boot

Exercice 1

Écrire puis tester les deux méthodes qui permettront de modifier ou supprimer une personne.

Spring Boot

Exercice 2

Développer une application utilisant les frameworks **Angular**, **React.js** ou **Vue.js**, offrant à l'utilisateur des interfaces graphiques pour la gestion des données relatives aux personnes. Ces fonctionnalités incluent l'ajout, la modification, la suppression, la consultation et la recherche des informations, en se basant sur le service web existant.