



Spring Cloud et GraphQL

David THIBAU – 2021

david.thibau@gmail.com



GraphQL

Rappels *GraphQL* *GraphQL Java* *Spring GraphQL* *Test*

Références :

<https://graphql.org/>

<https://github.com/graphql-java/graphql-java>

<https://docs.spring.io/spring-graphql/docs/1.0.0-SNAPSHOT/reference/html/>

<https://www.slideshare.net/Pivotal/spring-graphql>



Introduction

- ❖ ***GraphQL (Graph Query Language)*** :
Spécification d'un langage de requête et
d'un environnement d'exécution
Dernière release spéc : 2018, Draft en cours
- ❖ Développé par Facebook en 2012, puis
OpenSource en 2015
- ❖ API Web : alternative à REST et SOAP
- ❖ C'est le client qui détermine la réponse de
l'appel Web plutôt que le serveur



Exemple

Requête POST :

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

Réponse :

```
{
  "data": {
    "orders": [
      {
        "id": 1,
        "productsList": [
          {
            "product": {
              "name": "orange",
              "price": 1.5
            },
            "quantity": 100
          }
        ],
        "totalAmount": 150
      }
    ]
  }
}
```



Principe

Un service *GraphQL* est créé

- En définissant un schéma : Types de données et leurs champs
- En fournissant les fonctions permettant d'accéder à chaque champ

A l'exécution, lors de la réception d'une requête¹, le service

- Vérifie si la requête correspond au schéma
- Exécute toutes les fonctions nécessaires pour récupérer les données

1. Typiquement via HTTP(S), mais la spéc. ne le précise pas



Arguments

Il est possible de passer des **arguments** aux champs d'une requête. Ils peuvent servir à de la sélection ou à des transformations

```
human(id: "1000") {  
  name  
  height(unit: METER)  
}
```

```
"data": {  
  "human": {  
    "name": "Luke Skywalker",  
    "height": 1.72  
  }  
}
```



Alias

Les **alias** permettent de fournir des noms alternatifs pour éviter toute confusion dans la réponse

```
{
  empireHero: hero(episode: EMPIRE) {
    name
  }
  jediHero: hero(episode: JEDI) {
    name
  }
}
```

```
{
  "data": {
    "empireHero": {
      "name": "Luke Skywalker"
    },
    "jediHero": {
      "name": "R2-D2"
    }
  }
}
```



Fragments

Les **fragments** sont des sous-ensembles de requêtes réutilisables

```
{
  first: hero(episode: EMPIRE) {
    ...comparisonFields
  }
  second: hero(episode: JEDI) {
    ...comparisonFields
  }
}

fragment comparisonFields on Character
{
  name
  appearsIn
  friends {
    name
  }
}

{
  "data": {
    "first": {
      "name": "Luke Skywalker",
      "appearsIn": ["NEWHOPE", "EMPIRE", "JEDI"],
      "friends": [
        {"name": "Han Solo"},
        {"name": "R2-D2"}
      ]
    },
    "second": {
      "name": "R2-D2",
      "appearsIn": ["NEWHOPE", "EMPIRE", "JEDI"],
      "friends": [
        {"name": "Luke Skywalker"},
        {"name": "Han Solo"},
        {"name": "Leia Organa"}
      ]
    }
  }
}
```




Variables

Les **variables** permettent de variabiliser des requêtes.

Il suffit alors de changer la valeur de la variable pour avoir une réponse différente

```
query HeroNameAndFriends($episode: Episode)
{
  hero(episode: $episode) {
    name
    friends {
      name
    }
  }
}

{
  "episode": "NEWHOPE"
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```



Directives

Une **directive** permet d'inclure ou d'exclure un champ ou un fragment (*@include* et *@skip*)

```
query Hero($episode: Episode,  
           $withFriends: Boolean!) {  
  hero(episode: $episode) {  
    name  
    friends @include(if: $withFriends) {  
      name  
    }  
  }  
}
```

```
{  
  "episode": "JEDI",  
  "withFriends": false  
}
```

```
{  
  "data": {  
    "hero": {  
      "name": "R2-D2"  
    }  
  }  
}
```



Mutations

Il est possible d'écrire des données sur le serveur avec GraphQL

Par convention, les écritures sont effectuées à l'aide de l'opération **mutation**

```
mutation CreateReviewForEpisode(
  $ep: Episode!, $review: ReviewInput!) {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}
```

```
{
  "data": {
    "createReview": {
      "stars": 5,
      "commentary": "This is a great movie!"
    }
  }
}
```

```
{
  "ep": "JEDI",
  "review": {
    "stars": 5,
    "commentary": "This is a great movie!"
  }
}
```



Subscription

Une autre opération a été ajoutée dans la spécification : ***subscription***

Elle s'applique au modèle réactif et fournit donc un stream d'évènements en réponse

```
subscription sub {  
  newMessage {  
    body  
    sender  
  }  
}
```



Schémas et types

Le schéma conditionne les requêtes valides

Il définit les structures de données (Objets) et leur types ?

Exemple :

```
type Character {  
  name: String!  
  appearsIn: [Episode!]!  
}
```

- › Un *Character* est constitué des 2 champs *name* et *appearsIn*
- › *name* est une chaîne de caractères non-null
- › *appearsIn* est un tableau non-null d'objet *Episode* défini autre part dans le schéma

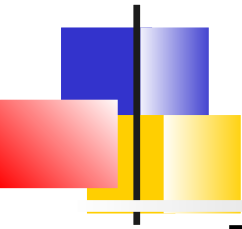


Arguments

Chaque champ peut avoir zéro ou plusieurs arguments avec éventuellement des valeurs par défaut.

Ce sont les arguments qui seront passées à la fonction responsable de récupérer la valeur du champ

```
type Starship {  
  id: ID!  
  name: String!  
  length(unit: LengthUnit = METER): Float  
}
```



Types de base

Types spéciaux (opérations) :

query, mutation, subscription

Scalaire :

Int, Float, String, Boolean, ID

Enumeration :

enum

List :

[]



Interface

Une **interface** est un type abstrait qui définit un ensemble de champs qu'un type doit inclure pour l'implémenter

```
interface Character {  
    id: ID!  
    name: String!  
    friends: [Character]  
    appearsIn: [Episode]!  
}  
type Droid implements Character {  
    id: ID!  
    name: String!  
    friends: [Character]  
    appearsIn: [Episode]!  
    primaryFunction: String  
}
```




Interface et fragments en ligne

Les interfaces sont utiles lorsque la réponse d'une requête contient des objets de type différents

```
query HeroForEpisode($ep: Episode!) {  
  hero(episode: $ep) {  
    name  
    ... on Droid {  
      primaryFunction  
    }  
  }  
}
```

```
{  
  "ep": "JEDI"  
}
```

```
{  
  "data": {  
    "hero": {  
      "name": "R2-D2",  
      "primaryFunction": "Astromech"  
    }  
  }  
}
```



Types Union

Les types ***union*** sont très similaires aux interfaces, mais ils ne spécifient aucun champ commun entre les types.

```
union SearchResult = Human | Droid | Starship
```

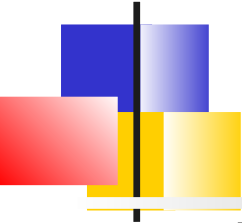
De la même façon, il faudra utiliser un fragment en ligne pour afficher les champs d'un type spécifique



Exemple Union

```
{
  search(text: "an") {
    __typename
    ... on Human {
      name
      height
    }
    ... on Droid {
      name
      primaryFunction
    }
    ... on Starship {
      name
      length
    }
  }
}
```

```
{
  "data": {
    "search": [
      {
        "__typename": "Human",
        "name": "Han Solo",
        "height": 1.8
      },
      {
        "__typename": "Human",
        "name": "Leia Organa",
        "height": 1.5
      },
      {
        "__typename": "Starship",
        "name": "TIE Advanced
x1",
        "length": 9.2
      }
    ]
  }
}
```



Type d'entrée

Les types d'entrée servent à passer des structures d'objet comme arguments

Très utile pour les mutations

```
input ReviewInput {  
  stars: Int!  
  commentary: String  
}
```



Exemple : mutation, types d'entrée

```
mutation CreateReviewForEpisode(
  $ep: Episode!, $review: ReviewInput!) {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}
```

```
{
  "data": {
    "createReview": {
      "stars": 5,
      "commentary": "This is a great movie!"
    }
  }
}
```

```
{
  "ep": "JEDI",
  "review": {
    "stars": 5,
    "commentary": "This is a great movie!"
  }
}
```



Exécution

Après avoir été validée, une requête GraphQL est exécutée par un serveur qui renvoie un résultat qui reflète la forme de la requête demandée, généralement au format JSON.

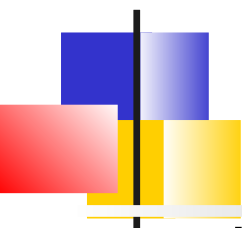


Resolver

A chaque champ d'une requête GraphQL est associée une fonction ou une méthode du type englobant qui renvoie le type défini par le champ.

La fonction, nommée ***resolver*** ou ***DataFetcher***, est fournie par le développeur

Lorsqu'un champ est exécuté, le résolveur correspondant est appelé pour produire la valeur suivante.



Arguments d'un resolver

Une fonction *resolver* reçoit 4 arguments :

- **obj** : L'objet précédent.
- **args** Les arguments fournis au champ dans la requête GraphQL.
- **context** (optionel) Valeur fournie à chaque résolveur et contenant des informations contextuelles importantes telles que l'utilisateur actuellement connecté ou l'accès à une base de données.
- **info** (optionel) Une valeur qui contient des informations spécifiques au champ relatives à la requête actuelle ainsi que les détails du schéma, voir *GraphQLResolveInfo* .



Points d'entrée

Au niveau racine de chaque serveur GraphQL se trouve un type qui représente tous les points d'entrée possibles dans l'API GraphQL, il est souvent appelé le type **Root** ou le type **Query**

```
Query: {  
  human(obj, args, context, info) {  
    return context.db.loadHumanByID(args.id).then(  
      userData => new Human(userData)  
    )  
  }  
}
```



Types de resolver

Certains *resolvers* peuvent effectuer des opérations asynchrones, i.e. chargement via une BD.

D'autres *resolvers* sont triviaux, i.e. accès à une propriété d'un objet.

D'autres font des transformations, i.e. transformer un entier en énumération

Cela reste transparent pour l'API



Exemples Javascript : Liste

```
human: {
```

```
  starships(obj, args, context, info) {
```

```
    // Liste de Promises
```

```
    return obj.starshipIDs.map(
```

```
      id => context.db.loadStarshipByID(id).then(
        shipData => new Starship(shipData)
```

```
      )
```

```
    )
```

```
  }
```

```
}
```



Introspection

Il est toujours possible de récupérer le schéma via l'API via le champ **`__schema`**

Par exemple, pour récupérer tous les types disponibles

```
{  
  __schema {  
    types {  
      name  
    }  
  }  
}
```

Pour connaître le type représentant les *query*

```
{  
  __schema {  
    queryType {  
      name  
    }  
  }  
}
```



Introspection (2)

Pour connaître les champs d'un type particulier :

```
{
  __type(name: "Droid") {
    name
    fields {
      name
      type {
        name
        kind // NON-NULL, INTERFACE, LIST, ...
      }
    }
  }
}
```



GraphQL

Rappels *GraphQL*
GraphQL Java
Spring GraphQL
Test



Implémentation Java / Spring

GraphQL Java : Démarré en 2015, plus de 40 releases

<https://github.com/graphql-java/graphql-java>

GraphQL Java Tools : Facilite la gestion des schémas GraphQL

<https://github.com/graphql-java-kickstart/graphql-java-tools>

GraphQL Java Spring : Démarré en 2019, 2 releases. Intégration avec Spring/SpringBoot. Utilisation de HTTP comme protocole de transport

<https://github.com/graphql-java/graphql-java-spring>

Spring GraphQL : Successeur du précédent, Démarré en 2020, 0 releases, Intégration avec Spring/SpringBoot (Officiel Pivotal).

GA prévue en Mai 2022 avec Spring Boot 2.7

<https://github.com/spring-projects/spring-graphql>

Netflix DGS : Intégration avec SpringBoot, Nombreuses releases, Idées reprises dans *Spring GraphQL*

<https://github.com/netflix/dgs-framework/>



Introduction

GraphQL Java est l'implémentation Java (serveur) de GraphQL.

Le moteur de *GraphQL Java* ne s'occupe que de l'exécution de requêtes. Il ne traite aucun sujet lié à HTTP ou à JSON

Les principales étapes de création d'un serveur *Java GraphQL* sont :

- Définir un schéma GraphQL.
- Implémenter les *Resolvers/DataFetcher* des champs et les configurer.



Hello world

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
        String schema = "type Query{hello: String}";  
        // Parsing du schéma  
        SchemaParser schemaParser = new SchemaParser();  
        TypeDefinitionRegistry typeDefinitionRegistry = schemaParser.parse(schema);  
        // Définition des resolvers  
        RuntimeWiring runtimeWiring = new RuntimeWiring()  
            .type("Query", builder -> builder.dataFetcher("hello", new  
StaticDataFetcher("world")))  
            .build();  
        // Génération du schéma  
        SchemaGenerator schemaGenerator = new SchemaGenerator();  
        GraphQLSchema graphQLSchema = schemaGenerator.makeExecutableSchema(typeDefinitionRegistry,  
runtimeWiring);  
        // Instanciation du moteur  
        GraphQL build = GraphQL.newGraphQL(graphQLSchema).build();  
        ExecutionResult executionResult = build.execute("{hello}");  
  
        System.out.println(executionResult.getData().toString());  
        // Prints: {hello=world}  
    }  
}
```



Définition de schéma

Soit via une String

```
SchemaParser.newParser()  
    .schemaString("Query { }")
```

Soit via un fichier du classpath

```
SchemaParser.newParser()  
    .file("my-schema.graphqls")
```

Possibilité de séparer en plusieurs fichiers



Association GraphQL / Java

GraphQL Java associe les champs des objets *GraphQL* aux méthodes et propriétés des objets Java.

- Pour la plupart des champs scalaires, un POJO avec des propriétés est suffisant. (*PropertyDataFetcher*)
- Les champs plus complexes nécessitent des méthodes plus complexes avec un état non fourni par le contexte GraphQL (dépôts, connexions, etc.).



Resolvers et classes de données

Le développeur fournit donc des **classes de données** (simple POJO) et des **resolver**

- Par défaut, à chaque champ est associé le *PropertyDataFetcher*
- Les autres Resolver doivent être configurés explicitement

Lors de l'exécution, le moteur utilise

- soit le *Resolver* spécifique si il existe,
- soit *PropertyDataFetcher*



PropertyDataFetcher

Le DataFetcher par défaut supporte les Map et Java beans

- Si la source est une Map, appel de *source.get(propertyName)*
- Sinon il est capable d'utiliser les getters ou les propriétés

Dans la définition du schéma, il est possible de définir des types qui sont mappés vers des propriétés différentes du bean :

```
directive @fetch(from : String!) on FIELD_DEFINITION
```

```
type Product {  
  ...  
  description : String @fetch(from:"desc")  
}
```



Exemple : Schéma

```
type Query {  
  books(match : String): [Book!]  
}
```

```
type Book {  
  id: Int!  
  name: String!  
  author: Author!  
}
```

```
type Author {  
  id: Int!  
  name: String!  
}
```



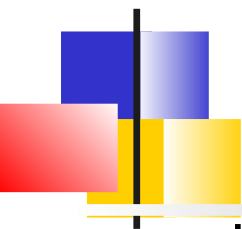
Exemple : Classes de données

```
@Data
class BookDto {
    private int id;
    private String name;
    private LocalDate launchDate ;
    private int authorId;

}
```

```
@Data
class AuthorDto {
    private int id;
    private String name;

}
```



DataFetchingEnvironment

Les Datafetcher reçoivent en argument un ***DataFetchingEnvironment*** qui encapsule :

- **Source** : L'objet qui résulte du fetching du champ parent
- **Root** : Le champ racine
- **Arguments** : Les arguments
- **Context** : Un *ThreadLocal* disponible pendant tout le traitement de la requête. On peut y stocker et récupérer les valeurs que l'on veut (Typiquement, une connexion BD)
- **ExecutionStepInfo** : Le graphe d'exécution sous forme de chemin
- **SelectionSet** : L'ensemble des champs enfants
- **ExecutionId** : Un identifiant unique de la requête. (Utile pour les traces par exemple)



Exemple DataFetcher spécifique

```
DataFetcher booksDataFetcher = new DataFetcher<List<BookDTO>>() {  
    @Override  
    public List<BookDTO> get(DataFetchingEnvironment environment) {  
        DatabaseSecurityCtx ctx = environment.getContext();  
  
        List<BookDTO> books;  
        String match = environment.getArgument("match");  
        if (match != null) {  
            books = fetchBooksFromDatabaseWithMatching(ctx, match);  
        } else {  
            books = fetchAllBooksFromDatabase(ctx);  
        }  
        return books;  
    }  
};
```



Example (2)

```
DataFetcher authorDataFetcher = new DataFetcher<AuthorDTO>() {  
    @Override  
    public List<BookDTO> get(DataFetchingEnvironment environment) {  
        DatabaseSecurityCtx ctx = environment.getContext();  
        BookDTO bookDto = (BookDTO)environment.getSource();  
  
        AuthorDTO author = fetchAuthorFromDatabase(ctx, book);  
  
        return author;  
    }  
};
```



JavaBeans et DataFetchEnvironment

Les POJO peuvent également profiter de l'environnement.

```
@Data
class BookDTO {
    private int id;
    private String name;
    private LocalDate launchDate ;
    private int authorId;

    // ...

    public String getLaunchDate(DataFetchingEnvironment environment) {
        String dateFormat = environment.getArgument("dateFormat");
        return yodaTimeFormatter(launchDate,dateFormat);
    }
}
```



Cablage

Avant pouvoir exécuter une requête, il faut câbler le schéma (*Runtime wiring*).

Cela consiste à :

- Associer les DataFetchers aux types du schéma
- Éventuellement, définir de nouveaux types scalaires¹.
- Éventuellement, associer des *TypeResolver* aux interfaces ou union

1. Voir <https://github.com/graphql-java/graphql-java-extended-scalars>



Exemple Wiring

```
RuntimeWiring buildRuntimeWiring() {  
    return RuntimeWiring.newRuntimeWiring()  
        .scalar(ExtendedScalars.DateTime)  
        // Syntax avec lambda  
        .type("QueryType", typeWiring -> typeWiring  
            .dataFetcher("hero", new StaticDataFetcher(StarWarsData.getArtoo()))  
            .dataFetcher("human", StarWarsData.getHumanDataFetcher())  
            .dataFetcher("droid", StarWarsData.getDroidDataFetcher())  
        )  
        .type("Human", typeWiring -> typeWiring  
            .dataFetcher("friends", StarWarsData.getFriendsDataFetcher())  
        )  
  
        // Ou builder  
        .type(  
            newTypeWiring("Character")  
                .typeResolver(StarWarsData.getCharacterTypeResolver())  
                .build()  
        )  
        .build();  
}
```



Exécution

Pour exécuter une requête sur un schéma, il faut appeler ***execute()*** sur une instance de ***GraphQL*** créé avec le schéma et le wiring.

La requête est fournie sous forme de ***ExecutionInput***

Le résultat d'une requête est un ***ExecutionResult*** qui contient les données et/ou une liste d'erreurs.



GraphQLSchema

```
SchemaParser schemaParser = new SchemaParser();  
SchemaGenerator schemaGenerator = new SchemaGenerator();  
File schemaFile = loadSchema("starWarsSchema.graphqls");
```

```
TypeDefinitionRegistry typeRegistry = schemaParser.parse(schemaFile);  
RuntimeWiring wiring = buildRuntimeWiring();
```

```
GraphQLSchema graphqlSchema =  
    schemaGenerator.makeExecutableSchema(typeRegistry, wiring);
```



Exécution

```
GraphQL graphql = GraphQL.newGraphQL(schema)
    .build();

ExecutionInput executionInput = ExecutionInput.newExecutionInput()
    .query("query { hero { name } }")
    .build();

ExecutionResult executionResult = graphql.execute(executionInput);

Object data = executionResult.getData();
List<GraphQLError> errors = executionResult.getErrors();
```




GraphQL

Rappels GraphQL
GraphQL Java
Spring GraphQL
Test



Introduction

Profite de l'IoC et des auto-configurations

- Starter SpringBoot prévu pour la 2.6

Prend en charge les requêtes GraphQL
sur HTTP et sur WebSocket

Mécanisme de propagation de contexte et
de résolution d'exception

Intégration avec Spring Security

Apports Spring



Boot Starter

Initialization

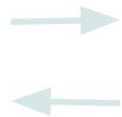
Context propagation

Security

Exception resolution

@Controller

Querydsl repositories



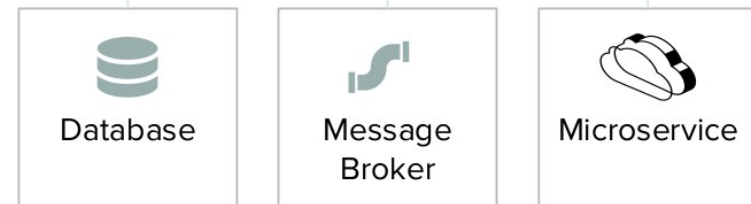
HTTP,
WebSocket

GraphQL Java
Engine

Data Fetchers

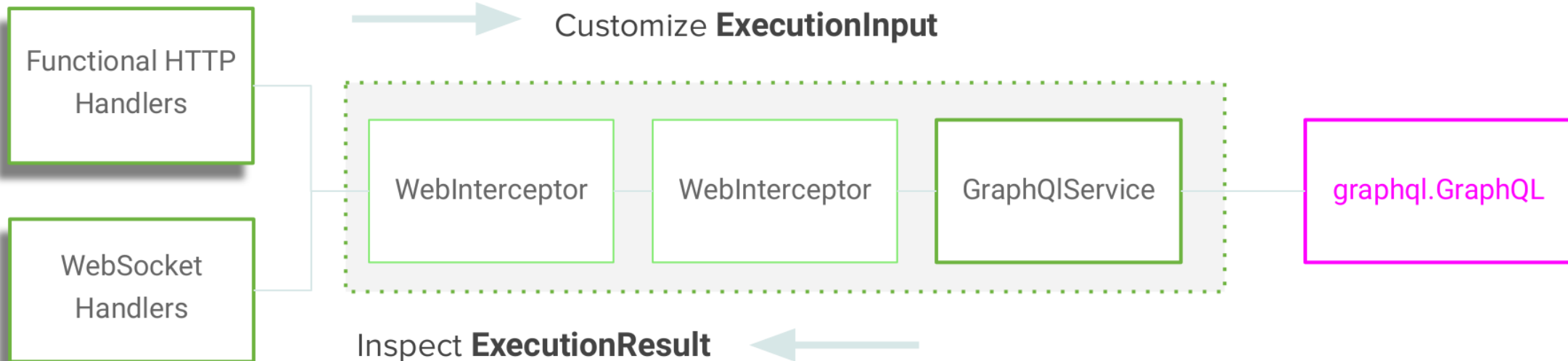
Spring MVC
Spring WebFlux

Testing



Big Picture

WebMvc + WebFlux





HTTP

Requêtes POST en JSON

Si requête correct, réponse toujours 200

- En cas d'erreur, un corps de message décrivant l'erreur

Compatible avec *starter-web* et *starter-webflux*

- Fonctionnalités équivalentes et traitement asynchrone de la requête
- Différence sur les écritures bloquantes ou non bloquantes de la réponse HTTP

Point d'accès exposés via *RouterFunction*



WebSocket

Support *WebSocket* basé sur *graphql-ws*¹

Utile pour traiter les *subscriptions*

Compatible avec *starter-web* et *starter-webflux*

- Fonctionnalités équivalentes et traitement asynchrone de la requête
- *Webflux* utilise de I/O non bloquant et supporte le *back pressure* pour contrôler le débit du flux

1. <https://github.com/enisdenjo/graphql-ws>



WebInterceptor

Les gestionnaires de requêtes (HTTP ou WebSocket) délègue le traitement à une chaîne d'interception : séquence de ***WebInterceptor***

Ils peuvent être utilisés pour inspecter les entêtes HTTP ou enregistrer des transformateurs de *graphql.ExecutionInput*

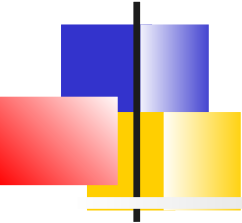


Exemple *WebInterceptor*

Transformation de la réponse

```
class MyInterceptor implements WebInterceptor {

    @Override
    public Mono<WebOutput> intercept(WebInput webInput,
    WebGraphQLHandler next) {
        return next.handle(webInput)
            .map(webOutput -> {
                Object data = webOutput.getData();
                Object updatedData = ... ;
                return webOutput.transform(builder ->
                    builder.data(updatedData));
            });
    }
}
```

GraphQLService

GraphQLService : Abstraction pour appeler le moteur *GraphQL Java* pour l'exécution des requêtes

Il localise le moteur via ***GraphQLSource*** qui contient un *builder* pour initialiser le moteur GraphQL avec son schéma



Propagation de contexte

2 mécanismes différents pour la propagation de données tout au long du traitement de la requêtes (DataFetcher, Repository, etc..) :

- MVC : Implémentation d'un ***ThreadLocalProcessor***
- WebFlux : Utilisation du *Reactor Context*



Exemple WebMVC

```
public class RequestAttributesAccessor implements ThreadLocalAccessor {  
  
    private static final String KEY = RequestAttributesAccessor.class.getName();  
  
    @Override  
    public void extractValues(Map<String, Object> container) {  
        container.put(KEY, RequestContextHolder.getRequestAttributes());  
    }  
  
    @Override  
    public void restoreValues(Map<String, Object> values) {  
        if (values.containsKey(KEY)) {  
            RequestContextHolder.setRequestAttributes((RequestAttributes) values.get(KEY));  
        }  
    }  
  
    @Override  
    public void resetValues(Map<String, Object> values) {  
        RequestContextHolder.resetRequestAttributes();  
    }  
}
```



Exemple WebFlux

```
public class ContextWebFilter implements WebFilter {

    // Le contexte et ces données sont disponibles tout au long de la requête
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {
        return chain.filter(exchange).contextWrite(context -> context.put("name", "007"));
    }
}

...
@Repository
public class DataRepository {

    public Mono<String> getGreeting() {
        return Mono.deferContextual(context -> {
            Object name = context.get("name");
            return Mono.delay(Duration.ofMillis(50)).map(aLong -> "Hello " + name);
        });
    }
}
```



Exception

GraphQL Java permet d'enregistrer un unique ***DataFetcherExceptionHandler***

Spring GraphQL permet d'enregistrer un chaîne de

DataFetcherExceptionResolver

=> Ils sont appelés séquentiellement jusqu'à ce que l'on résolve l'exception en une liste d'objets *graphql.GraphQLError*



Example

```
@Component
public class MyExceptionResolver extends DataFetcherExceptionHandlerAdapter {

    @Override
    protected GraphQLError resolveToSingleError(Throwable ex, DataFetchingEnvironment env) {
        return GraphQLErrorBuilder.newError(env)
            .message("Resolved error: " + ex.getMessage())
            .errorType(ErrorType.INTERNAL_ERROR).build();
    }
}
```



Batching

Étant donné un livre et son auteur, nous pouvons créer un *DataFetcher* pour charger les livres et un autre pour charger l'auteur de chaque livre.

- Cela permet aux requêtes de sélectionner uniquement les données dont elles ont besoin
- mais lors du chargement de plusieurs livres, nous finissons par charger l'auteur pour chaque livre individuellement,
Problème du N+1 select

Spring GraphQL permet de traiter cela en mode batch et de charger tous les auteurs des livres en 1 seul *select*



Fonctionnement

Au moment de l'exécution d'une requête, une application peut enregistrer une fonction de chargement par lots, i.e. un ***DataLoader*** dans le *DataLoaderRegistry*

Un *DataFetcher* peut accéder au *DataLoader* pour charger une entité via sa clé unique.

Le *DataLoader* ne charge pas l'entité immédiatement, mais renvoie plutôt une promesse et diffère jusqu'à ce qu'il puisse utiliser la fonction de chargement par lots pour charger toutes les entités associées ensemble.

Le *DataLoader* conserve également un cache des entités précédemment chargées.



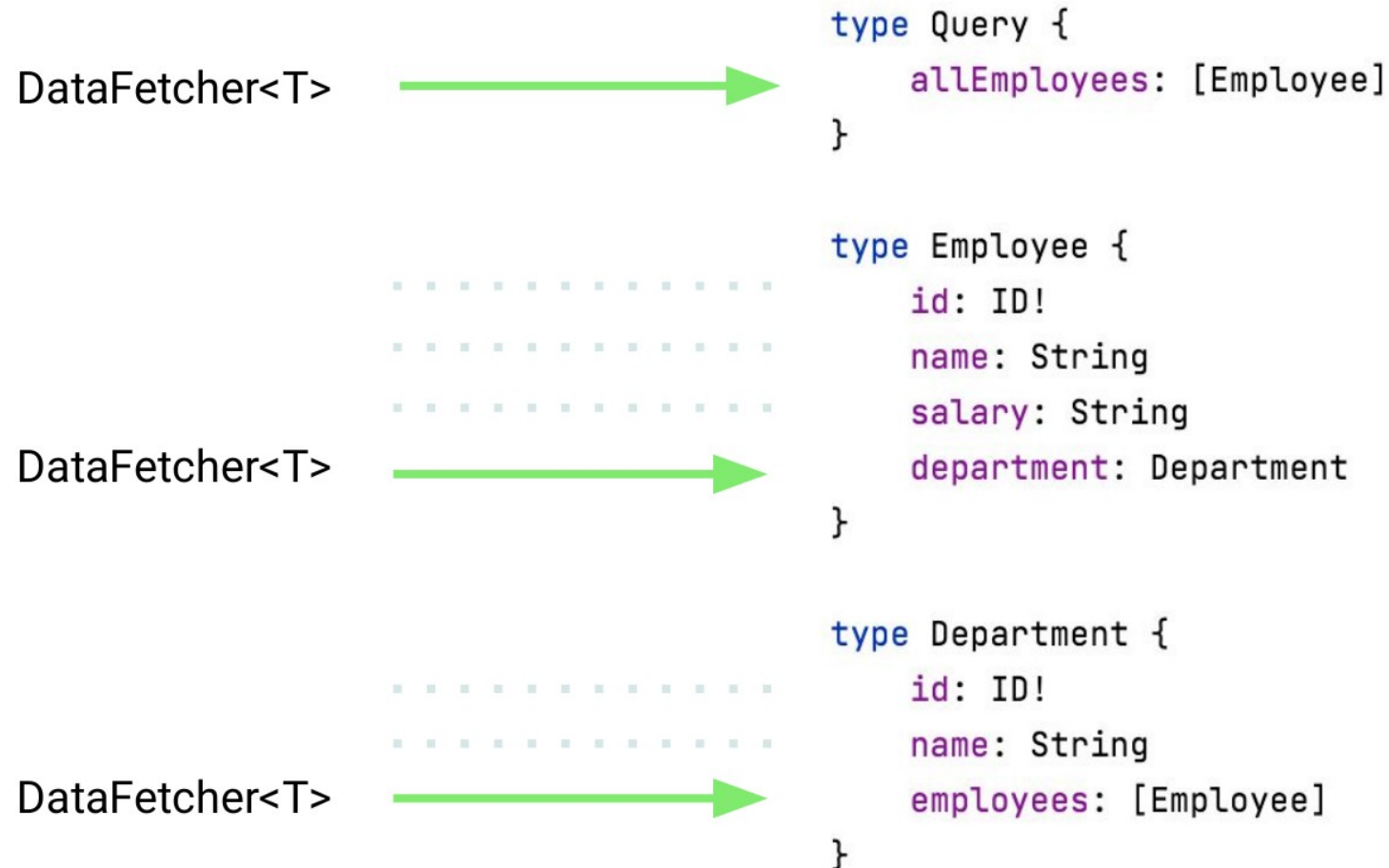
Données

Contrat GraphQL Java : ***DataFetcher***

```
public interface DataFetcher<T> {  
    T get(DataFetchingEnvironment environment)  
        throws java.lang.Exception  
}
```



Example





Configuration programmatische

Si l'on veut faire un câblage manuel, on peut définir un bean ***RuntimeWiringConfigurer***

```
public void configure(RuntimeWiring.Builder wiringBuilder) {  
  
    wiringBuilder.type("Query", builder -> builder.dataFetcher(  
        "allEmployees", environment -> this.employeeService.getAllEmployees()));  
  
    wiringBuilder.type("Department", builder -> builder.dataFetcher(  
        "employees", environment -> {  
            Department department = environment.getSource();  
            return this.employeeService.getEmployeesForDepartment(department.getId());  
        }));  
  
    wiringBuilder.type("Mutation", builder -> builder.dataFetcher(  
        "updateSalary", environment -> {  
            Map<String, Object> inputMap = environment.getArgument("input");  
            String employeeId = (String) inputMap.get("employeeId");  
            BigDecimal salary = new BigDecimal((String) inputMap.get("newSalary"));  
            this.employeeService.updateSalary(employeeId, salary);  
            return null;  
        }));  
}
```



Annotations

Spring GraphQL fournit un modèle de programmation basé sur des annotations.

Les composants **@Controller** utilisent des annotations pour déclarer des méthodes de traitement des requêtes

Elles ont des signatures flexibles pour récupérer les données des champs GraphQL.



@QueryMapping

```
@Controller
public class EmployeeController {

    @QueryMapping { type: Query, field: allEmployees }
    public List<Employee> allEmployees() {
        return this.employeeService.getAllEmployees();
    }
}
```

```
type Query {
    ➡ allEmployees: [Employee]
}

type Employee {
    id: ID!
    name: String
    salary: String
    department: Department
}
```



@SchemaMapping

@Controller

```
public class EmployeeController {
```

@QueryMapping

```
public List<Employee> allEmployees() {  
    return this.employeeService.getAllEmployees();  
}
```

type = **Department**, field = **employees**

@SchemaMapping

```
public List<Employee> employees(Department department) {  
    return employeeService.getEmployeesForDepartment(department.getId());  
}
```

```
type Employee {  
    id: ID!  
    name: String  
    salary: String  
    department: Department  
}  
  
type Department {  
    id: ID!  
    name: String  
    → employees: [Employee]  
}
```




@MutationMapping

@Controller

public class EmployeeController

```
type Mutation {
```

```
  ➡ updateSalary(input: UpdateSalaryInput!): UpdateSalaryPayload
}
```

@QueryMapping

```
public List<Employee> allEmployees() {
    return this.employeeService.getAllEmployees();
}
```

@SchemaMapping

```
public List<Employee> employees(Department department) {
    return employeeService.getEmployeesForDepartment(department.getId());
}
```

type = **Mutation**, field = **updateSalary**

@MutationMapping

```
public void updateSalary(@Argument SalaryInput input) {
    String employeeId = input.getEmployeeId();
    BigDecimal salary = input.getNewSalary();
    this.employeeService.updateSalary(employeeId, salary);
}
```



Complément

@SchemaMapping peut être déclarée au niveau de la classe

Cela permet de spécifier un type par défaut pour toutes les méthodes *handler*

```
@Controller
@SchemaMapping(typeName="Book")
public class BookController {

    // Méthodes pour les champs de type "Book"
}
```




Méta-annotations

@QueryMapping, *@MutationMapping* et *@SubscriptionMapping* sont des méta-annotations qui sont elles-mêmes annotées avec *@SchemaMapping*

Le type est alors prédéfini à *Query*, *Mutation* ou *Subscription*

```
@Controller
```

```
public class BookController {
```

```
    @QueryMapping
```

```
    public Book bookById(@Argument Long id) { // ... }
```

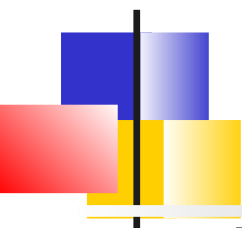
```
    @MutationMapping
```

```
    public Book addBook(@Argument BookInput bookInput) { // ... }
```

```
    @SubscriptionMapping
```

```
    public Flux<Book> newPublications() { // ... }
```

```
}
```



Arguments des méthodes

Les arguments des méthodes handler peuvent être :

- **@Argument** : Pour accéder aux arguments du champ
- **Source** : Pour accéder au champ parent
- **DataLoader** : Pour accéder à un *DataLoader* du *DataLoaderRegistry*
- **GraphQLContext** : Pour accéder au contexte à partir de *DataFetchingEnvironment*
- **DataFetchingEnvironment** : Pour un accès direct à *DataFetchingEnvironment*



@Argument

Le nom de l'argument peut être spécifié :
`@Argument("bookInput")`

Sinon le nom du paramètre de méthode est utilisé
(nécessite l'indicateur de compilateur `-parameters` avec Java 8+ ou des informations de débogage du compilateur).

Par défaut, un `@Argument` est requis, on peut le rendre facultatif avec `required=false` ou en le déclarant avec `java.util.Optional`.

On peut utiliser `@Argument` sur une `Map<String, Object>` pour obtenir toutes les valeurs d'argument. L'attribut `name` ne doit alors pas être défini.



Source

Pour accéder à l'instance du parent, il suffit de déclarer un paramètre de méthode du type attendu.

```
@Controller
public class BookController {

    @SchemaMapping
    public Author author(Book book) {
        // ...
    }
}
```

L'argument permet également de déterminer le type pour le mapping



DataLoader

Lorsque l'on enregistre une fonction de chargement par lots pour une entité, on peut accéder au *DataLoader* en déclarant un argument de méthode de type *DataLoader*

```
@Controller
public class BookController {

    public BookController(BatchLoaderRegistry registry) {
        registry.forTypePair(Long.class, Author.class).registerBatchLoader((authorIds, env) -> {
            // load authors
        });
    }

    @SchemaMapping
    public CompletableFuture<Author> author(Book book, DataLoader<Long, Author> loader) {
        return loader.load(book.getAuthorId());
    }
}
```



GraphQLContext et DataFetchingEnvironment

GraphQLContext : peut être utilisé pour contenir des valeurs clés utiles lors du fetching des données

DataFetchingEnvironment : contexte d'exécution où on peut trouver des informations relative au champ *graphql*



Starter

```
<dependencies>
  <dependency>
    <groupId>org.springframework.experimental</groupId>
    <artifactId>graphql-spring-boot-starter</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </dependency>
</dependencies>
```

```
<!-- For Spring project milestones or snapshot releases -->
```

```
<repositories>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
  </repository>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>
```



Schéma

Par défaut les schémas présents dans ***src/main/resources/graphql*** avec les extensions `".graphqls"`, `".graphql"`, `".gql"`, or `".gqls"` sont chargés

- Peut être customisé via la propriété *spring.graphql.schema.locations*

Le schéma peut être accédé via http si *spring.graphql.schema.printer.enabled=true*



GraphQL

Rappels GraphQL
GraphQL Java
Spring GraphQL
Test



GraphQLTester

Spring GraphQL fournit **GraphQLTester** qui permet de :

- Vérifier que les réponses GraphQL sont 200 (OK).
- Vérifier qu'il n'y a pas d'erreurs inattendues sous la clé "errors" dans la réponse.
- Décoder le contenu de "data" dans la réponse.
- Utiliser JsonPath pour décoder différentes parties de la réponse.
- Tester les Subscription.



Dépendances

```
<dependency>  
  <groupId>org.springframework.graphql</groupId>  
  <artifactId>spring-graphql-test</artifactId>  
  <version>1.0.0-SNAPSHOT</version>  
  <scope>test</scope>  
</dependency>
```



Mise en place

L'annotation **@AutoConfigureGraphQLTester** permet de configurer automatiquement un bean de type *GraphQLTester*

```
@SpringBootTest
```

```
@AutoConfigureGraphQLTester
```

```
public class MockMvcGraphQLTests {
```

```
@Autowired
```

```
private GraphQLTester graphQLTester;
```



Exemples

// jsonPath

```
this.graphQlTester.query(query)
    .execute()
    .path("project.releases[*].version")
    .entityList(String.class)
    .hasSizeGreaterThan(1);
```

// Contenu json

```
this.graphQlTester.query(query)
    .execute()
    .path("project")
    .matchesJson("{\"repositoryUrl\":\"http://github.com/spring-projects/spring-framework\"}");
```

// Décodage json

```
this.graphQlTester.query(query)
    .execute()
    .path("project")
    .entity(Project.class)
    .satisfies(project -> assertThat(project.getReleases()).hasSizeGreaterThan(1));
```



Exemple Subscription

```
Flux<String> result =  
    graphqlTester.query("subscription { greetings }")  
        .executeSubscription()  
        .toFlux("greetings", String.class);
```

```
StepVerifier.create(result)  
    .expectNext("Hi")  
    .expectNext("Bonjour")  
    .expectNext("Hola")  
    .verifyComplete();
```



Couche transport

Si il faut valider la couche transport lors du test, on peut utiliser

WebGraphQLTester

Il est également auto-configuré via l'annotation

@AutoConfigureGraphQLTester

On peut alors utiliser mockMvc,
WebTestClient ou un vrai serveur



Exemple

```
@SpringBootTest
@AutoConfigureWebTestClient
@AutoConfigureGraphQLTester
class SampleApplicationTests {

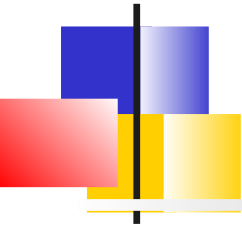
    @Autowired
    private WebGraphQLTester graphQlTester;

    @Test
    void anonymousThenUnauthorized() {

        this.graphQlTester.query(query)
            .execute()
            .errors()
            .satisfy(errors -> {
                assertThat(errors).hasSize(1);
                assertThat(errors.get(0).getErrorType()).isEqualTo(ErrorType.UNAUTHORIZED);
            });
    }
}
```




Annexe



GraphQL

QueryDSL



Introduction

QueryDsl : Un moyen sûr (type safe) d'exprimer des requêtes en Java qui fonctionne sur plusieurs stockage de données

Spring Data prend en charge *Querydsl*

Spring GraphQL fournit un

QuerydslDataFetcher :

- Adapte un repository *Spring Data* à un *DataFetcher*
- Traduit les paramètres des requêtes *GraphQL* en des prédicats de *Querydsl*



QueryDsl

```
List<Person> persons =  
    queryFactory.selectFrom(person)  
        .where(  
            person.firstName.eq("John"),  
            person.lastName.eq("Doe"))  
        .fetch();
```



Exemple

[https://github.com/spring-projects/
spring-graphql/blob/main/samples/
webmvc-http/src/main/java/io/spring/
sample/graphql/repository/
ArtifactRepositories.java](https://github.com/spring-projects/spring-graphql/blob/main/samples/webmvc-http/src/main/java/io/spring/sample/graphql/repository/ArtifactRepositories.java)