Cahier de TP
«Spring Cloud et GraphQL»

Outils utilisés :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux, MacOs, Windows 10
- JDK11+
- Spring Tools Suite 4.0 avec Lombok
- Git
- Postman

Atelier 1: SpringData

1.1 Spring Data JPA

Créer un projet avec

- Type : **Maven**
- Nom : ***member-documents***
- Groupe : ***org.formation***
- package : ***org.formation***

et les dépendances suivantes

- une dépendance sur le starter ***Spring Data JPA***
- sur le starter ***validation***
- sur le starter ***lombok***
- une dépendance sur ***H2 Database***

Récupérer les classes modèles fournies ainsi que le script sql.

Configurer Hibernate afin qu'il montre les instructions SQL exécutées

Démarrer l'application et vérifier le script *import.sql* est bien exécuté

Définissez des interfaces *JpaRepository* qui implémentent les fonctionnalités suivantes :

- CRUD sur Member et documents
- Rechercher tous les documents
- Trouver les membres ayant un email particulier
- Trouver le membre pour un email et un mot de passe donné
- Tous les membres dont le nom contient une chaîne particulière
- Rechercher tous les documents d'un membre à partir de son nom (Penser à utiliser

l'annotation *@Query*)

- Compter les membres
- Compter les documents
- Trouver un membre à partir de son ID avec tous les documents associés pré-chargés

Modifier la classe de test générée par *SpringIntializr* pour vérifier que les méthodes effectuent les bonnes requêtes

Profil de prod

Ajouter une dépendance sur le driver PostgresSQL

Définir une base *postgres* utilisant un pool de connexions (maximum 10 connexions) dans un profil de production.

Créer une configuration d'exécution qui active ce profil

1.2. Reactive Mongo DB

Démarrer un autre Projet Spring avec

- Type : **Gradle**
- Name : **account**
- Group : **org.formation**
- package : **org.formation**

et comme dépendances

- ***reactive-mongodb***
- ***embedded Mongo*** (enlever le scope test)

Récupérer la classe modèle fournie

1.2.1 Classe Repository

Créer une interface Repository héritant de `ReactiveCrudRepository<Account, String>`

Définir 2 méthodes réactives :

- Une méthode permettant de récupérer toutes les classes *Account* via leur attribut *value*
- Une méthode permettant de récupérer la première classe *Account* via l'attribut *owner*

Utiliser la classe de test pour exécuter du code ajoutant des classes *Account* dans la base, puis effectuant les requêtes définies par la classe *Repository*

1.2.2 ReactiveMongoTemplate

Créer une classe de configuration créant un bean de type ***ReactiveMongoTemplate*** comme suit :

```
@Configuration
public class ReactiveMongoConfig {

    @Autowired
    MongoClient mongoClient;

    @Bean
    public ReactiveMongoTemplate reactiveMongoTemplate() {
        return new ReactiveMongoTemplate(mongoClient, "test");
    }
}
```

Créer ensuite une classe Service exposant une interface métier de gestion des classes *Account* et utilisant le template.

Appeler ce Service dans la classe de test.

Atelier 2: Spring MVC, API REST

Continuation du projet SpringData JPA

2.1 Contrôleurs

Créer un contrôleur REST *MemberRestController* permettant de :

- D'effectuer toutes les méthodes CRUD sur un membre
- Afficher les membres contenant un chaîne particulière

Créer un contrôleur REST *DocumentRestController* permettant de :

- Récupérer tous les documents d'un membre donné
- D'ajouter un document à un membre

Certaines méthodes pourront envoyer des exceptions métier « *MemberNotFoundException* », « *DocumentNotFoundException* »

Désactiver le pattern « *Open Session in View* »

Tester les URLs GET

2.2 Configuration

Configurer le cors

Ajouter un *ControllerAdvice* permettant de centraliser la gestion des exceptions *MemberNotFoundException*

2.3 OpenAPI et Swagger

Ajouter les dépendances Swagger dans *pom.xml*

Accéder à la description de notre api REST (*http://server:port/swagger-ui.html*)

Ajouter des annotations OpenAPI pour parfaire la documentation

Atelier 3 : Spring Webflux

L’objectif est d’offrir une API Rest pour la gestion de la base Mongo du TP précédent

3.1 Endpoint fonctionnels

Reprendre le TP précédent et y ajouter le starter WebReactif et supprimer l’ancienne classe principale

Créer une classe *Handler* regroupant les méthodes permettant de définir les *HandlerFunctions* suivantes :

- « *GET /accounts* » : Récupérer tous les *accounts*
- « *GET /accounts/{id}* » : Récupérer un *account* par un id
- « *POST /accounts* » : Créer un account

Créer la classe de configuration WebFlux déclarant les endpoints de notre application.

Utiliser le script JMeter fourni pour tester votre implémentation.

Atelier 4. Spring Security

Ajouter *Spring Security* dans les dépendances du projet Web ***members-document***

Tester l'accès à l'application

Activer les traces de debug pour la sécurité

Visualiser le filtre ***springSecurityFilterChain***, effectuer la séquence d’authentification et observer les messages sur la console

Éventuellement, personnaliser le login et le mot de passe de l’utilisateur par défaut.

Atelier 5. Spring Test

Ce TP continue le projet précédent et ajoute différents types de tests à notre projet.

Nous travaillons donc dans l'arborescence ***src/test***

A chaque mise en place des tests, il est conseillé de se faire injecter le contexte applicatif Spring (*ApplicationContext*) afin d'inspecter les beans configurés

5.1 Tests auto-configurés

@DataJpaTest

Ecrire une classe de test vérifiant le bon fonctionnement de la méthode *findByOwner*

@JsonTest

Ecrire une classe de test vérifiant le bon fonctionnement de la sérialisation/désérialisation de la classe Member

@WebMVCTest

Utiliser ***@WebMVCTest*** pour :

- Tester MemberRestController en utilisant un mockMVC
- Tester l'accès à la page d'accueil avec HtmlUnit

5.2 Tests complets (avec la sécurité)

Utiliser l'annotation ***SpringBootTest*** pour se créer un environnement web démarrant sur un port aléatoire

Utiliser les annotations ***@WithMockUser*** pour simuler un utilisateur authentifié

Atelier 6 : GraphQL

Explorer l’API : https://countries.trevorblades.com/

Via des requêtes d’introspection, déc ouvrer l’API

Effectuer des requêtes :

- La liste des continents et leurs pays associés

- Le continent dont le code est ***‘AN’*** avec pour chaque pays associé son nom, sa capitale et sa monnaie.

- Variabiliser la requête

- Quel est le sens de l’écriture du Pashto parlé en Afghanistan ?

Atelier 7 : GraphQL Java

Dans le projet members-document, ajouter la dépendance sur graphql-java :

```
<dependency>
	<groupId>com.graphql-java</groupId>
	<artifactId>graphql-java</artifactId>
	<version>16.2</version>
</dependency>
```

Dans ***src/main/resources/graphql***, définir le schéma ***member.graphqlds*** qui permettra de récupérer les Membres présents dans la base de données

Dans une classe de test annotée *@SpringBootTest(webEnvironment = WebEnvironment.**NONE**)*

- Écriture une méthode *@BeforeEach* qui effectue les initialisations :
  - Lecture du schéma
  - Définition d'au minimum un DataFetcher capable de récupérer tous les membres de la base
  - Câblage
  - Création du GraphQLSchema
  - Instanciation du moteur
- Ecrire une méthode de test effectuant une requête *GraphQL* simple, par exemple : *"query { members { nom } }"*

Atelier 8: Spring GraphQL

8.1 Mise en place SpringBoot

Toujours sur le projet member-documents, remplacer la dépendance sur graphql-java par le starter Spring GraphQL

Désactiver la sécurité en enlevant le starter spring-security

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

Configurer SpringBoot pour pouvoir afficher le schéma via http :
spring.graphql.schema.printer.enabled=true

Démarrer l'application et accèder ensuite aux URL
http://localhost:8080/graphql/schema
Ainsi que
http://localhost:8080/graphiql

Effectuer des requêtes d'introspection

8.2 Requêtes de Query

Développer un contrôleur et les méthodes nécessaires pour répondre à :

```
{
  members {
    nom
    documents {
      name
    }
  }
```

8.3 Requête de mutation

Modifier le schéma afin d’ajouter une mutation permettant d’insérer un membre en base, utiliser un type Input pour encapsuler les données

Implémenter la méthode correspondante dans le contrôleur

8.4 Modèle réactif

Appliquer les mêmes étapes au modèle réactif

Atelier 9 : Test GraphQL

Sur le projet réactif, ajouter la dépendance suivante :

```
testImplementation 'org.springframework.graphql:spring-graphql-test:1.0.0-SNAPSHOT'
```

Modifier le schéma pour définir un subscrition

Faire le mapping dans la classe controller et retourner un Flux<Account>

Ecrire ensuite une classe de test qui configure automatiquement GraphlTester

Effectuer la requête de subscription et utiliser StepVerifier pour vérifier le nombre d'instance Account

Atelier 10 : *ApolloClient*

5.1 Récupérer le schéma du micro-service cible

Récupérer la projet Gradle fournie

Démarrer le projet réactif ***account***

Exécuter la commande suivante

```
./gradlew downloadApolloSchema \
  --endpoint="http://localhost:8080/graphql" \
  --schema="src/main/graphql/org/formation/schema.graphqls"
```

5.2 Définir un fichier de requête

Dans un fichier ***src/main/graphql/org/formation/acounts.graphql***, définir une query GraphQL

Générer les classes correspondantes via

```
./gradlew build
```

5.3 Exécution de la requête

Écrire une classe principale comme suit :

```
// Créer un ApolloClient
ApolloClient apolloClient = ApolloClient.builder()
       .serverUrl("http://localhost:8080/graphql")
       .build();

// Ensuite enqueue la requête
apolloClient.query(new AccountsQuery())
       .enqueue(new ApolloCall.Callback<AccountsQuery .Data>() {
          @Override
          public void onResponse(@NotNull Response<AccountsQuery
.Data> response) {
              // Afficher response.getData
          }

          @Override
          public void onFailure(@NotNull ApolloException e) {
              // Afficher l'erreur
          }
       });
```

Atelier 11 : Micro-services

11.1 Mise en place serveurs de discovery et de config

Importer les projets ***eureka*** et ***config*** fournis

Les démarrer dans l'IDE, démarrer *config* puis *eureka*

Vérifier le serveur eureka sur http://localhost:1111

Sur le projet ***account***

Ajouter les dépendances suivantes :

```
implementation 'org.springframework.cloud:spring-cloud-starter-bootstrap'
implementation 'org.springframework.boot:spring-boot-starter-actuator'
implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
```

```
implementation 'org.springframework.cloud:spring-cloud-starter-config'
```

Récupérer les fichiers de configuration :

- ***bootstrap.yml*** : à mettre dans ***src/main/resources*** du projet et supprimer le fichier application.yml ou properties présent dans le répertoire
- ***account-service.yml*** : à mettre dans le projet config dans ***src/main/resources/shared***

Faire de même pour le projet ***member-documents***

Redémarrer ***config*** avec les nouvelles ressources

Vérifier les bonnes inscriptions des 2 services dans Eureka

11.2 Interactions micro-services et Load-balancing

Dans le projet account, définir une nouvelle requête GraphQL :

```
accountByOwner(owner: String): Account
```

L'implémenter dans la classe Controller

Dans le projet *member-documents*, récupérer les classes Dto fournies et définir une nouvelle requête dans le schéma GraphQL :

```
memberWithAccount(id :Int): MemberAccount
```

```
type MemberAccount {
    id: Int!
    nom: String!
    prenom: String
    email: String!
    value: Float  // Provient du service account

}
```

MemberAccount et donc une agrégation de données stockées dans la BD relationnelle et dans Mongo

Implémenter une classe ***AccountService*** qui encapsulera les appels *GraphQL* vers le service *account* :

- S'injecter un *DiscoveryClient*
- Instancier un REST Template
- Interroger Eureka pour obtenir l'URL d'une instance ACCOUNT-SERVICE disponible, implémenter un load balancing basique
- Effectuer la requête JSON correcte :
  ```
  String q ="{\"query\":\"{\\n
  accountByOwner(owner: \\\"dthibau@wmmod.com\\\") {\\n    id\\n    value\\n
  }\\n}\",\"variables\":null}";
  ```
- Décoder la réponse pour obtenir une instance de AccountDto

Utiliser le service dans la méthode du contrôleur mappée sur la nouvelle requête.

Tester le tout avec 1 puis plusieurs instances de *account-service*.

(Pour démarrer plusieurs instances de account-service sur localhost, vous pouvez utiliser la notion de profil de Spring)

11.3 Circuit Breaker Pattern

Dans le projet ***member-documents***, ajouter la dépendance sur *Resilience4j*

Encapsuler le code de l'appel du service *account* dans un circuit breaker.