Ateliers Spring Essentials

Pré-requis :

- JDK 11+
- IDE (IntelliJIDEA, Eclipse, STS, VSCode)
- Librairie lombok : https://projectlombok.org/downloads/lombok.jar
- Docker
- Git

TP1 : Première configuration Spring

Le premier TP reprend l'exemple développé dans la présentation.
Nous voulons donc développer un objet métier capable de lister tous les films d'un réalisateur.
Les films étant stockés dans une base (Un simple fichier texte pour le moment).

L'objet métier offre donc une méthode :

```
public List<Movie> moviesDirectedBy(String director)
```

Une interface Java définit les méthodes que devront implémenter les objets DAO liés à un type de repository particulier.

```
public interface MovieDAO {
	public List<Movie> findAll();
}
```

Une implémentation de cette interface pour un fichier tabulé respectant un format particulier est fournie (*org.formation.dao.FileDAO*).

Enfin, une classe de test permet également de tester l'implémentation de votre objet métier.

1.1 Configuration XML

1. Mise en place du projet
Importer le projet Maven fourni

2. Implémenter org.formation.service.MovieLister
Implémenter la fonction métier et les constructeurs ou setters permettant l'injection de dépendance.

3. Effectuer la configuration de Spring
Dans le fichier ***src/test/resources/test.xml***, déclarer les beans nécessaires et positionner leurs attributs.

4. Exécuter la classe de test

Exécuter la classe de test *org.formation.service.MovieListerTest.*

TP2 Configuration via les annotations

2.1 Configuration via annotations

Reprendre le TP précédent :

Ajouter la dépendance sur ***javax.annotation :javax.annotation-api :1.3.2*** afin d'avoir à disposition les annotations *@PostConstruct* et *@PreDestroy*

Effectuer une configuration équivalent au TP précédent uniquement via des annotations.

En particulier, on utilisera :

- Les annotations @Configuration et les annotations de stéréotypes
- On externalisera le nom du fichier contenant les films dans un fichier *application.properties*

2.2 Profils

Nous voulons exécuter le test dans 2 profils distinct « file » et « jdbc »

Ajouter les dépendances suivantes dans le pom.xml :

```
<dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>5.3.13</version>
</dependency>
<dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>42.3.1</version>
</dependency>
```

Démarrer la base de données avec :
docker-compose -f postgres-docker-compose.yml up -d

Se connecter sur *pgAdmin* (localhost:81) avec ***admin@admin.com/admin***
déclarer le serveur avec comme propriétés de connexion :
movies-postgresql postgres/postgres
Créer une BD nommée ***movies*** et exécuter le script d'initialisation fourni

Récupérer les classes JDBC fournies et les annoter correctement.

Réécrire la classe de test en définissant 2 méthodes :

- 1 effectuant let est dans le profile *file*
- l'autre dans le profil *jdbc*

TP3 : Démarrage SpringBoot

- Création d'un projet Java Spring Boot (*New → SpringStarter Project)*
  - Dépendance sur Web
  - Exécution, (*Run As → Spring Boot App*)
  - Accéder à ***http://localhost:8080***
  - Aller dans les *Run → Configurations*, surcharger la propriété *server.port*
    - Accéder à la nouvelle URL

- Créer une classe contenant le code suivant :

```
@Component
@ConfigurationProperties("hello")
public class HelloProperties {

	/**
	 * Greeting message returned by the Hello Rest service.
	 */
	private String greeting = "Welcome ";

	public String getGreeting() {
		return greeting;
	}

	public void setGreeting(String greeting) {
		this.greeting = greeting;
	}
}
```

- Une autre classe implémentant un service REST :

```
@RestController
public class HelloController {

	@Autowired
	HelloProperties props;

	@RequestMapping("/hello")
	public String hello(@RequestParam String name) {
		return props.getGreeting()+name;
	}
}
```

Ajouter dans le *pom.xml* une dépendance permettant de traiter l'annotation *@ConfigurationProperties* pendant la phase de build et créer les fichiers nécessaires afin que Spring Tool prennent en compte les nouvelles propriétés :

```
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-configuration-
processor</artifactId>
		</dependency>
```

- Tester ensuite la complétion dans l'éditeur de propriétés

- Tester l'application

Auto configuration

Dans la classe Main, utiliser la valeur de retour de `SpringApplication.run(Application.class, args);` pour afficher les beans Spring configurés.

Plug-in Maven

Démarrer l'application en ligne de commande après avoir fait un build (*mvn package* ou *gradlew bootJar*)

Starters de développement

Ajouter la dépendance sur `spring-boot-configuration-processor`

et l'activation de ***devtools***

Observer les conséquences de devtools : Redémarrage automatique lors du changement du code Java

Configuration propriétés applicatives via le format .yml

Renommer le fichier *application.properties* en *application.yml*

Définir les propriétés applicatives suivantes, ces propriétés sont encapsulées dans la classe de propriétés de configuration *HelloProperties* :

- *hello.greeting* (on vide) : La façon de dire *bonjour*
- *hello.styleCase* (Upper, Lower ou Camel) : La façon d'écrire le nom
- *hello.position* (0 ou 1) : Le nom est en premier ou en seconde position

Ajouter des contraintes de validation dans *HelloProperties*

Profils

Définir un port différent pour le profil *prod*

Activer le profil :

- Via votre IDE
- Via la ligne de commande après avoir généré le *fat jar*

Mode DEBUG et Configuration des traces

Activer l'option ***-debug*** au démarrage

Modifier la configuration afin de générer un fichier de trace

Modifier le niveau de trace du logger *org.springframework.boot* à DEBUG sans l'option ***-debug***

TP4 : SpringData

4.1 Spring Data JPA

Auto configuration par défaut de Spring JPA

Créer un projet avec

- une dépendance sur le starter ***Spring JPA***
- une dépendance sur ***hsqldb***

Récupérer les classes modèles fournies ainsi que le script sql.

Les classes *Member* et *Document* sont utilisées pour la partie JPA, la classe *Customer* pour la partie NoSQL

Si nécessaire ajouter une dépendance Maven ou Gradle

Configurer Hibernate afin qu'il montre les instructions SQL exécutées

Mettre au point un fichier *import.sql* qui insère des données de test en base

Démarrer l'application et vérifier que les insertions ont bien lieu

Faire un premier démarrage et observer la console

Définissez des interfaces *Repository* 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

Optionnellement :

Injecter un *EntityManager* ou un *Datasource* pour travailler directement au niveau de JPA ou JDBC

Configuration Datasource et pool de connexions

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

Implémentation Service

Implémenter une méthode métier qui permet d’ajouter un document à l’ensemble des utilisateurs de la base

Tester la méthode

TP5 : API Rest avec SpringBoot et SpringMVC

5.1 Contrôleurs

Créer un contrôleur REST *MemberRestController* permettant de :

- D'afficher tous les membres
- D'afficher les membres dont le nom complet (prenom + nom) contenant un chaîne particulière
- D'effectuer toutes les méthodes CRUD sur un membre

Lors du retour d'une liste de membres le json retourné ne contiendra que les attributs simples de *Member*

Lors du retour d'un unique membre le json contiendra également les documents associés.

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* »

Désactiver le pattern « *Open Session in View* »

Tester les URLs GET

5.2 CORS et gestion des Exceptions

Configurer le cors, pour autoriser toutes les requêtes

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

5.3 OpenAPI et Swagger

Ajouter la dépendance suivante pour obtenir la documentation OpenApi 3.0

```
<dependency>
	<groupId>org.springdoc</groupId>
	<artifactId>springdoc-openapi-ui</artifactId>
	<version>1.5.9</version>
</dependency>
```

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

Ajouter des annotations OpenAPI pour parfaire la documentation

TP7 : *SpringSecurity*

Cet TP permet de voir différentes implémentations de la sécurité

7.1 Configuration

Ajouter Spring Security dans les dépendances du projet Web précédent

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

Ajouter une classe de Configuration de type `WebSecurityConfigurerAdapter qui :`

Sur l'application MVC Back-end

- Autorise l'accès à la page home
- Oblige une authentification pour toutes les autres pages
- Sur un logout réussi, retourner à la page home

Sur l'application REST

- Autorise l'accès à la documentation swagger
- Nécessite une authentification pour les méthodes GET de l'api
- Nécessite le rôle ADMIN pour toutes les autres méthodes

7.2 Authentification custom

Mettre en place une classe implémentant *UserDetailService*, configurer l'authentification afin qu'elle utilise cette classe.

La classe s'appuiera sur le bean *MemberRepository* développé dans les Tps précédents

Faire également en sorte que les mots de passe soient cryptés dans la base.

TP6 : SpringBoot et 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

6.1 Tests auto-configurés

@DataJpaTest

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

@JsonTest

Écrire 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

TP7: Mise en Production

7.1 Actuator

Ajouter la dépendance vers *Actuator*,

Visualiser les endpoints par défaut

Configurer l'application afin que tous les endpoints soient activés et accessible

Visualiser les endpoints

Mettre en place un profil de production :

- Utilisation base Postgres
- Choix pour le profil de sécurité
- Désactivation de swagger-ui

7.2 Artefact pour la production

Modifier le build afin que les informations de build soient ajoutées

Configurer le build afin que l'artefact généré soit exécutable et ait le nom du projet

Démarrer l'application et accéder aux endpoints d'actuator

Optionnellement, construire une image Docker et la démarrer par Docker