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 le test 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* (non vide sans valeur par défaut) : 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.

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

Récupérer le script *import.sql* fourni et le placer correctement dans l'arborescence projet afin qu'il insère des données de test en base

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

Interfaces JpaRepository

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 ou le prénom 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

Tests

Utiliser la classe de test fournie pour valider les requêtes de l'entité *Member*.

Écrire sur le même modèle une classe de test validant les requêtes sur *Document*

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

TP4B (Optionnel) : MongoDB

Ajouter la dépendance sur MongoDB

Déclarer une classe modèle comme suit :

```
public class Customer {

    @Id
    public String id;

    public String firstName;
    public String lastName;

    public Customer() {}

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return String.format(
                "Customer[id=%s, firstName='%s', lastName='%s']",
                id, firstName, lastName);
    }
}
```

Définir une interface de type MongoRepository qui permet de recherche des classes Customer par les attributs firstName ou lastName

Au démarrage de l'application exécuter le code suivant :

```
private void _playWithMongo() {
            customerRepository.deleteAll();

            // save a couple of customers
            customerRepository.save(new Customer("Alice", "Smith"));
            customerRepository.save(new Customer("Bob", "Smith"));

            // fetch all customers
            System.out.println("Customers found with findAll():");
            System.out.println("-------------------------------");
            for (Customer customer : customerRepository.findAll()) {
                  System.out.println(customer);
            }
            System.out.println();

            // fetch an individual customer
            System.out.println("Customer found with
findByFirstName('Alice'):");
            System.out.println("--------------------------------");

      System.out.println(customerRepository.findByFirstName("Alice"));
```

```
            System.out.println("Customers found with
findByLastName('Smith'):");
            System.out.println("--------------------------------");
            for (Customer customer :
customerRepository.findByLastName("Smith")) {
                  System.out.println(customer);
            }

      }
```

Installer MongoDB pour tester ou ajouter la dépendance permettant d'avoir MongoDB en embarqué

TP5a (Optionnel) Application Web et Spring MVC

Ajout des dépendances

Sur le projet précédent, ajouter les starters suivants :

- web
- thymeleaf
- dev-tools

Récupérer les sources fournis (répertoire *static* et *templates* à placer dans /*src/main/resources*)

Configuration

Ajouter une configuration MVC déclarant les viewControleurs, permettant d'accéder aux pages présents dans le répertoire templates : *home.html* et *documents.html*

Contrôleur

Implémenter *MemberController* qui aura pour caractéristique :

- De répondre à une URL GET qui routera vers la page de ***login*** présent dans templates
- De répondre à l'URL POST de la page de login, et vérifier les données encapsulées dans la classe DTO User.
  - Si les données sont correctes, positionner un objet *Member* en session et rediriger vers la page ***documents***
  - Sinon, positionner une erreur

Webjar

Ajouter dans pom.xml la dépendance suivante :

```
<dependency>
     <groupId>org.webjars</groupId>
     <artifactId>bootstrap</artifactId>
     <version>3.3.7-1</version>
</dependency>
```

Effectuer un ***mvn install*** et recharger la page de login

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

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><last-version></version>
</dependency>
```

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

Ajouter des annotations OpenAPI pour parfaire la documentation

TP6 : *SpringSecurity*

Cet TP permet de voir différentes implémentations de la sécurité

6.1 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

6.2 Authentification Mémoire

Se définir des utilisateurs en mémoire permettant de vérifier les règles ACLs précédentes

Quels sont les filtres activés dans la chaîne de filtre ?

Activer le debug et effectuer des requêtes

6.3 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.

6.4 Authentification via OpenId/oAuth2

6.4.1 Authentification via Google, Github, Facebook

- Appliquer https://www.baeldung.com/spring-security-5-oauth2-login à notre projet
- Mettre en place une page spécifique
- En plus de la proposition de login, ajouter un formulaire d’authentification, permettant de s’authentifier avec la BD

TP6 (deprecated) : Appels REST

Créer un autre projet qui implémente un service implémentant 3 méthodes effectuant des appels REST vers l'application précédente :

- Charger 1 membre
- Charger tous les membres
- Créer un membre

Etapes :

- Créer un projet avec le starter web
  Désactiver le démarrage tomcat avec la propriété :
  `spring.main.web-application-type=none`
- Implémenter le bean Service en utilisant ***RestTemplateBuilder*** et ***RestTemplate***
- Écrire les tests permettant d'effectuer les appels

TP6 : Appels REST

Créer un autre projet qui implémente un service implémentant 3 méthodes effectuant des appels REST vers l'application précédente :

• Charger 1 membre
• Charger tous les membres
• Créer un membre

Etapes :
• Créer un projet avec le starter ***reactive-web***
Désactiver le démarrage tomcat avec la propriété :
spring.main.web-application-type=none

• Implémenter un bean Service en utilisant ***WebClient.Builder*** et ***WebClient***
• Écrire les tests permettant d'effectuer les appels

TP7 : Communication asynchrone

Nous mettons en place une communication de type Publish/Subscribe

ProductService publie l'événement TICKET_READY vers un topic

DeliveryService réagit en créant une livraison

Solution : tag 3

7.1 Démarrage du Message Broker

Récupérer le fichier ***docker-compose.yml*** permettant de démarrer le message broker Kafka et le processus ZooKeeper nécessaire

docker-compose up -d

Déclarer kafka dans le fichier host

```
127.0.0.1 kafka
```

7.2 Mise en place du producer

Ajouter les starters spring-kafka sur les projets ProductService

Implémenter une classe Service qui

- Crée charge un ticket
- Publie un événement *READY_TO_PICK* sur le topic Kafka ***tickets-status***, la clé du message et l'id du Ticket, le corps du message est une instance de la classe *ChangeStatusEvent*

7.3 Mise en place du consommateur

Ajouter les starters *spring-kafka* sur les projets *DeliveryService*

Implémenter une classe Service qui

- Écoute le topic ticket-status
- Si l'événement est du type READY_TO_PICK, créer une Livraison

Vous pouvez tester le tout avec le script JMeter fourni

TP9 : SpringBootTest et Test auto-configurés

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

@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

TP10: Mise en Production

10.1 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

10.2 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

TP11 (Optionnel) : Spring Data Réactive

11.1. Reactive Mongo DB

Démarrer un Spring Starter Project et choisir les starters ***reactive-mongodb*** et ***embedded Mongo***

Récupérer la classe modèle fournie

11.2 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*

Implémenter l'interface *CommandLineRunner* dans votre classe principale et exécuter du code ajoutant des classes *Acco unt* dans la base, puis effectuant les requêtes définies par la classe *Repository*

11.3 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.

Tester en appelant ce Service à partir de la classe principale.

TP12 : Spring Web Flux

12.1. Approche Contrôleur

L'objectif de cette partie est de comparer les performances et le scaling du modèle bloquant vis à vis du modèle non bloquant

Récupérer le projet Web fourni (modèle bloquant)

Créer un Spring Starter Project et choisir le starter ***web-reactive***

Implémenter un contrôleur équivalent à celui du modèle bloquant.

Utiliser le script JMeter fourni, effectuer des tirs avec les paramètres suivants : NB_USERS=100, PAUSE=1000

A la fin du résultat, notez :

- Le temps d'exécution du test
- Le débit

Effectuez plusieurs tirs en augmentant le nombre d'utilisateurs

12.2 Endpoint fonctionnels

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

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 déclararnt les endpoints de notre application.

Utiliser le script JMeter fourni pour tester votre implémentation.