

# 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*.

### **1.2 Cycle de création**

Dans ***MovieLister***, implémenter les interfaces *BeanNameAware*, *BeanClassLoaderAware*, *BeanFactoryAware*, *ResourceLoaderAware*, *ApplicationEventPublisherAware*, *MessageSourceAware*, *ApplicationContextAware*, *BeanPostProcessor*, *InitializingBean* et afficher juste un message sur la console.

Visualisez la console de l'exécution du test

## 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

### 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 profil *file*
- l'autre dans le profil *jdbc*

## TP3 : Illustration de Spring AOP

### Objectifs du TP

Ce TP est une illustration des capacités de Spring AOP.

Il consiste à appliquer de façon transversale à toutes les classes du package *com.tp4.service* un greffon permettant de tracer les temps d'exécution de chaque méthode.

La technique des annotations sera préférée.

### Travail à effectuer

#### 1. Mise en place du projet

Ajouter les dépendances suivantes :

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>5.3.13</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.8.13</version>
</dependency>
```

#### 2. Configuration Spring

Activer l'AOP via `@AspectJ` dans la configuration

#### 3. Implémentation du greffon

Déclaration d'un nouveau bean « *performanceMonitor* »

Dans la classe du bean, implémenter un greffon traçant en mode INFO le temps d'exécution de toutes les méthodes du package *org.formation.service*

#### 4. Exécution de la classe de test

Réexécuter la classe de test et observer la console

## TP4 : Démarrage SpringBoot

- Création d'un projet Java Spring Boot (New → SpringStarter Project)
  - Dépendance sur Web
  - Exécution, (*Run As* → *Spring Boot App*)
  - Accéder à 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 et Configuration de Debug**

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

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

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

et l'activation de **devtools**

Tester le redémarrage automatique lors du changement du code Java

### **Configuration Externe**

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

Définir les propriétés de configuration suivantes :

- Valeur aléatoire dans le fichier *yml* affectée à une variable du contrôleur  
Afficher la valeur dans un point d'accès du contrôleur
- Dans *HelloProperties* (Ajouter des validations):
  - *hello.greeting* : Non vide
  - *hello.styleCase* : Upper, Lower ou Camel
  - *hello.position* : 0 ou 1

### **Profil**

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**

## TP5 : SpringBoot et SpringData

### 5.1 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 PostgreSQL

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

## **5.2 Ajout d'un Repository 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);
    }
}
```

Ajouter la dépendance permettant d'avoir un MongoDB embarqué

## TP6 : SpringBoot et SpringMVC

Cet TP permet de voir certains aspects du starter-web :

- Environnement Spring MVC
- Organisation
  - des ressources statiques
  - des gabarits HTML (Thymeleaf) et la stratégie de résolution de vues

### 6.1 Application Web et Spring MVC

#### *Ajout des dépendances*

Sur le projet précédent, ajouter les starters suivants :

- web
- thymeleaf
- dev-tools

Ajouter également les dépendances **webjar**

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>4.4.1-1</version>
</dependency>
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>jquery</artifactId>
  <version>3.4.1</version>
</dependency>
```

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

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

#### **Configuration**

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

## Contrôleur

Implémenter le cas d'usage de l'enregistrement d'un membre *MemberController* qui aura pour caractéristique :

- De répondre à une GET */web/register* qui routera vers la page de **register** présent dans templates
- De répondre à POST de */web/register* qui enregistre un membre en base et le positionne dans l'attribut de session *loggedUser* puis redirige vers la page **documents**

## 6.2 API REST

### 6.2.1 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

### 6.2.2 Configuration

Configurer le cors

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

### 6.2.3 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

## 6.3 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

## 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 Authentication custom

Mettre en place une classe implémentant *UserDetailsService*, 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.

### 7.3 Authentication OpenId / oAuth2

- 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

## TP8 : 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

### 8.1 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

### 8.2 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

## ***TP9 : 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

### ***9.1 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`

### ***9.2 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*

### ***9.3 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



# TP10 : Spring Data Réactive

## 10.1. Reactive Mongo DB

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

Récupérer la classe modèle fournie

## 10.2 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 *Account* dans la base, puis effectuant les requêtes définies par la classe *Repository*

## 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.

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

## TP11 : Spring Web Flux

### 11.1. 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

### 11.2 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 WebReactive 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.

## TP12: Mise en Production

### **12.1 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

### **12.2 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

Optionnellement, déployer chez Heroku