# Ateliers Spring Essentials

Pré-requis :

- JDK 17+
- IDE (IntelliJIDEA, Eclipse, STS, VSCode)
- Librairie lombok : https://projectlombok.org/downloads/lombok.jar
- Docker
- Git

Solutions :

https://github.com/dthibau/springessentials-light-solutions

## Table des matières

# Atelier 1 : 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.*

# Atelier 2 : Configuration via les annotations

## *2.1 Configuration via annotations*

Reprendre le TP précédent :

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*

# Atelier 3 : Démarrage SpringBoot

## *3.1 Spring Initializr*

Aller sur https://start.spring.io

Créer un projet :

- Choix par défaut pour : ***Project, Language, SpringBoot***
- Group: ***org.formation***
- artificat : ***first-project***
- Package Name : ***org.formation***
- Choix par défaut : ***Packaging, Java***

Choisir dépendance ***Spring Reactive Web***

Générer le projet et copier le dans un répertoire de travail

Visualiser les fichiers générés

Construire le fat jar et l'exécuter

Générer une image Docker et l'exécuter

## *3.2 Support IDE*

- 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

## *3.3 Auto configuration*

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

## *3.4 Starters de développement*

Ajouter la dépendance sur ***devtools***

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

## *3.6 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*

## *3.7 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*

## *3.8 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***

# Atelier 4 : SpringData

## *4.1 Spring Data JPA*

### 4.1.1 Auto configuration par défaut de Spring JPA

Créer un projet avec

- une dépendance sur le starter ***Spring JPA***
- une dépendance sur ***H2***

Récupérer les classes modèles fournies et visualiser les annotations des classes *Member* et *Document*

Si nécessaire ajouter une dépendance Maven ou Gradle

Configurer Hibernate afin qu'il montre les instructions SQL exécutées

Visualiser le script *import.sql* fourni placé dans ***src/main/resources***

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

### 4.1.2 Interfaces JpaRepository

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

- CRUD sur Member et documents
- Rechercher tous les documents
- Trouver le membre 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

### 4.1.3 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

### 4.1.4 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

### 4.1.5 Implémentation Service

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

Tester la méthode

## *4.2 : MongoDB Réactif (Optionnel)*

Créer un nouveau projet avec les dépendances suivates :

- MongoReactif
- les starters de développement

Ajouter également une dépendance pour bénéficier d'un serveur Mongo Embarqué :

```
<dependency>
  <groupId>de.flapdoodle.embed</groupId>
  <artifactId>de.flapdoodle.embed.mongo.spring30x</artifactId>
  <version>4.5.2</version>
  <scope>test</scope>
</dependency>
```

Ajouter également le numéro de version de serveur Mongo dans le fichier de propriétés :

```
de:
  flapdoodle:
    mongodb:
      embedded:
        version: 4.0.2
```

Récupérer la classe modèle fournie et regarder l'import de l'annotation @Id

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

Dans une classe de tests, définir la méthode de test suivante

```
@Test
    public void _playWithMongo() {
        customerRepository.deleteAll();
        // save a couple of customers
        customerRepository.save(new Customer("Alice", "Smith")).block();
        customerRepository.save(new Customer("Bob", "Smith")).block();

        // fetch all customers
        System.out.println("Customers found with findAll():");
        System.out.println("-------------------------------");

        List<Customer> customers = customerRepository.findAll().collectList().block();
        for (Customer customer : customers) {
            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").collectList().block());
        System.out.println("Customers found with findByLastName('Smith'):");
        System.out.println("-------------------------------");
        customers = customerRepository.findByLastName("Smith").collectList().block();

        for (Customer customer : customers) {
            System.out.println(customer);
        }
    }
```

Assurer vous que le test passe et vérifier les messages sur la console.

Améliorer le code afin qu'il soit plus réactif

# Atelier 5 : Spring Web

## *5.1 : API Rest avec SpringBoot et SpringMVC*

### *5*.1.1 Contrôleurs

Créer une couche contrôleur 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« *EntityNotFoundException* » « *MemberNotFoundException* »

Désactiver le pattern « *Open Session in View* »

Tester les URLs GET

### *5*.1.2 C*ORS et gestion des Exceptions*

Configurer le cors, pour autoriser toutes les requêtes

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

### *5*.1.3 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

# Atelier 6 : Client REST

## *6.1 Spring MVC*

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

Étapes :

- 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

## 6.2 Spring WebFlux

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

# Atelier 7 : Messaging

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

## *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 le projet ***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

# Atelier 8: *SpringSecurity*

Cet TP permet de voir différentes implémentations de la sécurité

## *8.1 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 :`

Si Vous avez une application MVC.

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

## *8.2 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

## *8.3 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.

## *8.4 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

## *8.5 Intégration KeyCloak*

### *1. Mise en place du fournisseur de jeton (KeyCloak)*

Démarrer un serveur KeyCloak via Docker :

```
docker run -p 8080:8080 -e KEYCLOAK_ADMIN=admin -e
KEYCLOAK_ADMIN_PASSWORD=admin quay.io/keycloak/keycloak:20.0.2
start-dev
```

- Connecter vous à sa console d'administration avec admin/admin
- Créer un Realm ***MemberRealm***
- Sous ce realm, créer 2 clients
  - ***member-app***
  - Positionner ***access-type*** à ***confidential*** et Valid Redirect Uri à ***r**
- Créer ensuite un utilisateur ***user/secret***

Obtention des jeton :

Grant type *password* pour le client *store-app*

```
curl -XPOST http://localhost:8089/auth/realms/MemberRealm/protocol/openid-
connect/token -d grant_type=password -d client_id=member-app -d
client_secret=<client_secret> -d username=user -d password=secret
```

Obtenir un jeton de rafraîchissement en utilisant une requête POST avec :

```
client_id:member-app
client_secret:<client_secret>
'refresh_token': refresh_token_requete_précédente,
grant_type:refresh_token
```

### *2. OAuth2 login et sécurité stateful*

Ajouter les starters ***`security`*** ***et*** ***`oauth2-client`*** au projet Members

Configurer a sécurité de la gateway (Bean étendant *SecurityFilterChain*) comme suit (Code Spring WebFlux) :

```
@Bean
public SecurityWebFilterChain securitygWebFilterChain(
  ServerHttpSecurity http) {
    return http.authorizeExchange()
            .pathMatchers("/actuator/**").permitAll()
            .pathMatchers("/auth/**").permitAll()
      .anyExchange().authenticated()
      .and().
      oauth2Login().csrf().disable().build();
}
```

Configurer le client oauth2 de Keycloak comme suit :

```
spring :
  security:
    oauth2:
      client:
        provider:
          keycloak:
            token-uri: http://localhost:8089/auth/realms/StoreRealm/protocol/openid-
connect/token
            authorization-uri:
http://localhost:8089/auth/realms/StoreRealm/protocol/openid-connect/auth
            user-info-uri:
http://localhost:8089/auth/realms/StoreRealm/protocol/openid-connect/userinfo
            user-name-attribute: preferred_username
        registration:
          store-app:
            provider: keycloak
            client-id: store-app
            client-secret: 57abb4f6-5130-4c73-9545-6d377dd947cf
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/keycloak"
```

Accéder ensuite à l'API via un navigateur

Pour mieux comprendre, vous pouvez voir le filtre configuré dans la console et également augmenter les traces via :

```
logging:
  level:
    org.springframework.security: debug
```

### *3. ACL et resource server*

Ajouter les starters ***`oauth2-resource-server`*** au projet .

Indiquer la propriété ***`spring.security.oauth2.resourceserver.jwr.issuer-uri`***

Modifier la configuration de la sécurité pour s'adapter au resource server

```
@Bean
public SecurityWebFilterChain securitygWebFilterChain(
  ServerHttpSecurity http) {
    return http.authorizeExchange()
            .pathMatchers("/actuator/**").permitAll()
            .pathMatchers("/auth/**").permitAll()
      .anyExchange().authenticated()
      .and().
      oauth2ResourceServer(v -> v.jwt()).csrf().disable().build();
}
```

Configurer les routes de la gateway de telle façon que le serveur KeyCloak soit accessible via la gateway.

# Atelier 9 : 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

## *9.1 @DataJpaTest*

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

## *9.2 @JsonTest*

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

## *9.3 @WebMVCTest*

Utiliser ***@WebMVCTest*** pour :

- Tester MemberRestController en utilisant un mockMVC

# Atelier 10: 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.

## *5.1 Application Web et Spring MVC (Optionnel)*

### 5.1.1 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*)

### 5.1.2 Configuration

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

### 5.1.3 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

### 5.1.4 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