Cahier de Tps : Formation SpringBoot / Angular (Partie I)

IDE Recommandé : Spring Tools Suite 4.x

I-1 : Prise en main IDE

I-1.1 – Création de projet via Spring Intializr

Créer un nouveau projet SpringBoot avec comme starter le module web.

Visualisez les fichiers générés

Démarrer l'application (« Run As → Spring Boot App),

Accéder à l'application

Visualiser le « Boot DashBoard »

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

Dans les *Run Configurations*, activer la case à cocher debug et observer le rapport d'auto-configuration

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

I-1.2 – Propriétés de configuration et profils

Ajouter les 2 starters suivants :

- ***devtools***
- ***configuration-processor***

Convertir ***application.properties*** en ***application.yml***

Ajouter la classe ***HelloController*** suivante :

```
@RestController
public class HelloController {

	@Autowired
	HelloProperties props;

	@RequestMapping("/hello")
	public String hello(@RequestParam String name) {
	// En fonction de HelloProperties « Dire bonjour » à name
	}
}
```

Le contrôleur utilise la dépendance ***HelloProperties***. C'est une classe de

configuration applicative qui encapsule toutes les propriétés de configuration de la façon dont l'application « *dit bonjour* »)
Les propriétés consiste en :

- *hello.greeting* : Non vide (Ex : « Hello »)
- *hello.styleCase* : (Upper ou Lower). Le paramètre nom est passé en Majuscule ou Minuscule
- *hello.position* : (0 ou 1) Le nom est devant le greeting ou bien l'inverse

Implémenter la classe ***HelloProperties*** ; faire en sorte qu'elle soit validée au démarrage et tester l'auto-complétion dans l'éditeur de propriété.

Ensuite, faire une injection simple de valeur dans le contrôleur. Utiliser par exemple une valeur aléatoire du fichier properties.

Surcharge de propriétés :

Dans l'IDE, surcharger des propriétés de configuration via la commande en ligne

E commande en ligne, surcharger des propriétés de configurations :

- via la commande en ligne
- via une variable d'environnement

Configuration des traces

Modifier la configuration afin de générer un fichier de trace

Ajouter des traces de debug dans le controller

Modifier le niveau de trace du logger afin que le niveau de trace soit WARN sauf pour nos classes applicatives : niveau DEBUG

Profil

Activer la configuration des traces précédentes seulement pour le profil *prod*

Activer le profil :

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

I-2 : SpringBoot et SpringData

I-2.1 Base de données embarquées et Repository

Auto configuration par défaut de Spring JPA avec h2

Créer un projet avec

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

Ainsi que des dépendances habituelles sur ***devtools, configuration-processor, lombok, validation***

Récupérer les classes modèles fournies ainsi que le script import.sql.

Configurer Hibernate afin qu'il montre les instructions SQL exécutées

Démarrer l'application et vérifier que la base est créée automatiquement et que les insertions ont bien lieu

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

- CRUD sur Produit et Fournisseur
- Rechercher tous les produits
- Trouver les produits d'un fournisseur
- Trouver un produit par sa référence ET la référence fournisseur
- Tous les membres dont le nom contient une chaîne particulière
- Trouver tous les fournisseurs qui ont des produits dont la disponibilité est supérieur à un paramètre donné

Dans la classe de test générée par *SpringIntializr* , ajouter des méthodes permettant de vérifier que les méthodes effectuent les bonnes requêtes

Implémenter des méthodes de tests qui ajoutent des données en base

Optionnellement :

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

I-2.2 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

I-3 : API Rest avec SpringBoot

I-3.1 Starters

Ajouter la dépendance sur le module web

I-3.1 Contrôleurs

Créer un contrôleur REST *ProduitRestController* permettant de :

- De récupérer tous les produits
- D'effectuer toutes les méthodes CRUD sur un produit
- Afficher les produits d'un fournisseur particulier

Créer un contrôleur REST *FourniseurRestController* permettant de :

- Récupérer un fournisseur via sa référence
- Créer un fournisseur

Certaines méthodes pourront envoyer des exceptions métier
« *ProductNotFoundException* », « *FournisseurNotFoundException* »

Désactiver le pattern « *Open Session in View* »

Tester les URLs GET

I-3.2 Configuration

Configurer le cors

Ajouter un *ControllerAdvice* permettant de centraliser la gestion des exceptions *ProductNotFoundException* et *FournisseurNotFoundException*

I-3.3 OpenAPI et Swagger

Ajouter les dépendances SpringDoc 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

I-3.4 Client Rest

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

- Charger 1 produit
- Charger tous les produits
- Créer un produit

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

I-4 : *SpringSecurity*

I-4.1 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 REST

- Autorise l'accès à swagger et aux URLs GET
- Nécessite une authentification pour les toutes les autres méthodes

I-4.2 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

4.3 Authentification custom

Nous voulons baser nos utilisateurs sur un fichier ***users.csv***,

Récupérer les sources fournis et compléter la classe ***UserDetailServiceImpl***, configurer l'authentification afin qu'elle utilise cette classe.

Tester

I-4.4 JWT

Ajouter la dépendance suivante :

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
```

```
        <artifactId>jjwt</artifactId>
        <version>0.7.0</version>
    </dependency>
```

Récupérer le code fourni, le regarder, le comprendre.

Configurer la sécurité afin d'intégrer le filtre *JWTFilter* dans la chaîne de sécurité

Le test de fonctionnement peut s'effectuer via le script *jMeter* également fourni

I-5 : SpringBoot et Spring Test

5.1 Tests auto-configurés

@DataJpaTest

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

@JsonTest

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

@WebMVCTest

Utiliser ***@WebMVCTest*** pour tester ProduitRestController en utilisant un mockMVC

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