Rapport TP1 - API REST et Microservices avec Spring Boot

1 Initialisation et Configuration du Projet Spring Boot

1.1 Création du Projet

Le projet Spring Boot est créé via **Spring Initializr**, un outil en ligne qui génère la structure de base du projet. Il est important d'utiliser Java 17 (et non Java 8) pour bénéficier des dernières fonctionnalités et du support à long terme.

Les fichiers sont organisés dans un package (par exemple `com.example.spring1`) qui structure l'application et évite les conflits de noms entre classes.

1.2 Rôle de Maven

Maven est un outil de gestion de projet dont le but principal est de **libérer le développeur de certaines tâches de configuration**, notamment l'installation et la gestion des dépendances. Maven télécharge automatiquement les bibliothèques nécessaires et gère les versions, simplifiant considérablement le processus de développement.

1.3 Configuration du Port d'Écoute

Par défaut, Spring Boot démarre sur le port 8080. Pour changer le port d'écoute, on modifie le fichier de configuration :

— **Avec application.properties :** `src/main/resources/application.properties`

```
server.port=9191
```

— **Avec application.yaml :** `src/main/resources/application.yaml`

```
server:
  port: 9191
```

Note : En YAML, l'indentation est cruciale et fait partie de la syntaxe.

2 Création d'une API REST Simple

2.1 Annotations Fondamentales

Pour créer une API REST avec Spring Boot, plusieurs annotations sont essentielles :

— **@RestController :** Placée avant la déclaration de la classe, cette annotation indique que la classe est un contrôleur REST. Elle combine `@Controller` et `@ResponseBody`, permettant de renvoyer directement des objets sérialisés en JSON.

— **@GetMapping(value = "/url") :** Mappe une méthode HTTP GET sur l'URL spécifiée. L'endpoint devient accessible via `http://localhost:<port>/url`.

— **@PostMapping :** Mappe une méthode HTTP POST, utilisée pour créer de nouvelles ressources.

— **@PutMapping :** Mappe une méthode HTTP PUT, utilisée pour modifier des ressources existantes.

— **@DeleteMapping :** Mappe une méthode HTTP DELETE, utilisée pour supprimer des ressources.

2.2 Les Méthodes HTTP et Leur Usage

— **GET :** Récupérer des données (lecture seule, idempotente)
— **POST :** Créer une nouvelle ressource
— **PUT :** Modifier une ressource existante (remplacement complet)
— **DELETE :** Supprimer une ressource

Ces méthodes forment la base de l'architecture REST et permettent d'effectuer toutes les opérations CRUD (Create, Read, Update, Delete).

2.3 Stockage Temporaire sans Base de Données

Pour simplifier le développement initial, on peut déclarer les objets en mémoire avec le mot-clé **static**. Cela permet de tester l'API sans avoir besoin de configurer une base de données. Les données sont stockées dans une structure comme `ArrayList` et persistent tant que l'application est en cours d'exécution.

Important : Utiliser `static` est nécessaire pour que les données soient partagées entre toutes les requêtes, car Spring crée de nouvelles instances du contrôleur pour chaque requête.

2.4 Tests de l'API

Deux méthodes principales permettent de tester l'API :

1. **Navigateur web :** Pour les requêtes GET simples, on peut directement accéder à l'URL en ajoutant les paramètres :

   ```
   http://localhost:9999/somme?a=1&b=1
   ```

   La syntaxe est : `?param1=val1&param2=val2&...&paramN=valN`

2. **Postman :** Outil plus complet permettant de tester toutes les méthodes HTTP (GET, POST, PUT, DELETE). Postman offre une interface graphique pour passer les paramètres, définir les en-têtes et visualiser les réponses JSON.

3 Persistance des Données avec Spring Data JPA

3.1 Architecture en Packages

Pour une meilleure organisation du code, le projet est structuré en trois packages :
— **entities :** Contient les classes qui représentent les entités de la base de données
— **repository :** Contient les interfaces d'accès aux données
— **web :** Contient les contrôleurs REST

3.2 JPA et Hibernate

JPA (Java Persistence API) est une spécification Java qui facilite le **mapping objet-relationnel** (ORM), c'est-à-dire la correspondance entre les objets Java et les tables d'une base de données. **Hibernate** est l'une des implémentations les plus populaires de JPA.

L'avantage principal : **Hibernate nous épargne beaucoup de travail** en générant automatiquement les requêtes SQL, en gérant les transactions et en synchronisant les objets avec la base de données.

3.3 Annotations JPA Essentielles

Pour qu'une classe Java devienne une entité persistante, deux annotations sont indispensables :

— **@Entity :** Placée avant la déclaration de la classe, elle indique que cette classe sera mappée vers une table de la base de données.

— **@Id :** Placée avant l'attribut servant de clé primaire, elle identifie le champ qui sera l'identifiant unique de chaque enregistrement.

@GeneratedValue(strategy = GenerationType.AUTO) : Placée sous `@Id`, cette annotation permet la génération automatique des clés primaires par la base de données, mais elle n'est pas obligatoire.

3.4 Interface Repository

On crée une interface qui hérite de `JpaRepository` :

```
public interface AdherentRepository extends JpaRepository<Adherent, Long
    >
```

Cette interface prend deux paramètres génériques :

— Le nom de l'entité (ici `Adherent`)

— Le type de la clé primaire (ici `Long`)

Spring Data JPA génère automatiquement l'implémentation de cette interface, fournissant des méthodes prêtes à l'emploi comme `save()`, `findAll()`, `findById()`, `delete()`, etc. C'est le principe d'**injection de dépendances**.

3.5 Initialisation des Données avec CommandLineRunner

Pour insérer des données de test au démarrage de l'application, on utilise un `CommandLineRunner` dans la classe principale :

```
@Bean
CommandLineRunner runner(AdherentRepository repository) {
    return args -> {
        repository.save(new Adherent(null, "A", "B", 29));
        repository.save(new Adherent(null, "C", "D", 25));
    };
}
```

L'annotation **@Bean** indique à Spring que cette méthode sera exécutée automatiquement au démarrage de l'application.

4 Configuration des Bases de Données

4.1 Base de Données H2 (En Mémoire)

H2 est une base de données relationnelle en mémoire, idéale pour le développement et les tests. Configuration dans `application.properties` :

```
server.port=9191
spring.datasource.url=jdbc:h2:mem:adherent
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

— `jdbc:h2:mem:adherent` : Crée une base en mémoire nommée "adherent"

— `spring.h2.console.enabled=true` : Active la console web H2

— `spring.h2.console.path=/h2-console` : Définit le chemin d'accès à la console

La console H2 est accessible via `http://localhost:<port>/h2-console`, permettant de visualiser et manipuler les données stockées.

4.2 Migration vers MySQL

Pour utiliser MySQL au lieu de H2, on modifie la configuration :

```
server.port=9191
spring.datasource.url=jdbc:mysql://localhost:3306/adherent
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=create
```

- `jdbc:mysql://localhost:3306/adherent` : URL de connexion MySQL (port 3306 par défaut)
- `username` et `password` : Identifiants de connexion à MySQL
- `ddl-auto=create` : Hibernate crée automatiquement les tables au démarrage (écrase les données existantes)

Important : Il faut également ajouter la dépendance MySQL dans le fichier `pom.xml` pour que Maven télécharge le driver JDBC MySQL.

Valeurs possibles pour `ddl-auto` :

- `create` : Crée les tables à chaque démarrage (détruit les données)
- `update` : Met à jour le schéma sans perdre les données
- `validate` : Valide le schéma sans le modifier
- `none` : Aucune action automatique

5 Principe REST et Microservices

Une API REST (Representational State Transfer) repose sur plusieurs principes :

- **Sans état (Stateless) :** Chaque requête contient toutes les informations nécessaires, le serveur ne conserve pas d'état entre les requêtes
- **Ressources identifiées par URL :** Chaque ressource (étudiant, adhérent) est accessible via une URL unique
- **Méthodes HTTP standardisées :** Utilisation cohérente de GET, POST, PUT, DELETE
- **Format JSON :** Les données sont échangées au format JSON, léger et universel

Les microservices sont des applications REST autonomes, légères et déployables indépendamment. Spring Boot facilite grandement leur création en fournissant un environnement d'exécution embarqué et une configuration automatique.