TP 2 – API REST et Microservices en Java

Safa Ferchichi - 12408918- INFOA2

1 Introduction

L'objectif de ce TP est de mettre en pratique les **principes des API REST** et des **microservices** en utilisant le langage **Java** et le framework **Spring Boot**.

Le travail est découpé en deux parties :

— **Première partie** : implémenter une API REST simple dont les données sont stockées en mémoire (liste Java).

— **Deuxième partie** : reprendre le même principe, mais cette fois en **persistant les données dans une base de données relationnelle** (ici MySQL) à l'aide de **Spring Data JPA**.

Les manipulations réalisées s'appuient sur des vidéos fournies par l'enseignant, qui montrent comment :

— créer un projet Spring Boot avec Spring Initializr,

— définir des classes de modèle et des contrôleurs REST,

— configurer la persistance des données via JPA.

2 Première partie : API REST en mémoire

2.1 Objectif

La première partie consiste à :

— Créer une API REST basique avec Spring Boot.

— Manipuler une ressource `Etudiant` (identifiant, nom, moyenne).

— Exposer plusieurs endpoints REST (GET, POST, PUT, DELETE).

— Tester l'API à l'aide de Postman.

Les données sont conservées dans une simple `ArrayList` Java, jouant le rôle de "pseudo base de données" en mémoire.

2.2 Configuration de l'application

Le fichier `application.properties` de la première partie est le suivant :

```
spring.application.name=Tp2SAE
server.port=9999
```

Listing 1 – Configuration Spring Boot – Partie 1

L'application écoute donc sur le port `9999` : les URLs des services commencent par `http://localhost:99`

2.3 Classe principale et annotation `@SpringBootApplication`

La classe principale Spring Boot est :

```
package com.example.tp2sae;

import org.springframework.boot.SpringApplication;
```

```java
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Tp2SaeApplication {

    public static void main(String[] args) {
        SpringApplication.run(Tp2SaeApplication.class, args);
    }

}
```

Listing 2 – Classe principale Tp2SaeApplication

Explication de l'annotation `@SpringBootApplication`

`@SpringBootApplication` est une méta-annotation qui regroupe :

— `@Configuration` : indique que la classe contient des définitions de beans Spring.

— `@EnableAutoConfiguration` : demande à Spring Boot de configurer automatiquement l'application en fonction des dépendances présentes dans le classpath.

— `@ComponentScan` : permet de scanner le paquet courant (et ses sous-paquets) pour détecter automatiquement les composants Spring (`@RestController`, `@Service`, etc.).

Concrètement, cette annotation transforme la classe en point d'entrée Spring Boot, lance un serveur web embarqué (Tomcat) et permet à l'application de démarrer avec une configuration minimale.

2.4 Modèle de données : classe `Etudiant`

La ressource principale est représentée par la classe `Etudiant` :

```java
package com.example.tp2sae;

public class Etudiant {
    private int identifiant;
    private String nom;
    private double moyenne;

    public Etudiant() {
    }

    public Etudiant(int identifiant, String nom, double moyenne) {
        this.identifiant = identifiant;
        this.nom = nom;
        this.moyenne = moyenne;
    }
// Getters et Setters ...

}
```

Listing 3 – Classe Etudiant

Cette classe représente un étudiant avec :

— `identifiant` : identifiant (utilisé comme index dans la liste).

— `nom` : nom de l'étudiant.

— `moyenne` : moyenne.

2.5 Contrôleur REST : `MyApi` et annotations REST

Le contrôleur qui expose l'API REST est défini comme suit :

```
package com.example.tp2sae;

import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.Collection;

@RestController
public class MyApi {
    public static ArrayList<Etudiant> liste = new ArrayList<>();

    static {
        liste.add(new Etudiant(0, "A", 19));
        liste.add(new Etudiant(1, "A", 19));
        liste.add(new Etudiant(2, "A", 19));
        liste.add(new Etudiant(3, "A", 19));
    }

    @GetMapping(value = "/liste")
    public Collection<Etudiant> getAllEtudiant() {
        return liste;
    }

    @GetMapping(value = "/getEtudiant")
    public Etudiant getEtudiant(int identifiant) {
        return liste.get(identifiant);
    }

    @PostMapping(value = "addEtudiant")
    public Etudiant addEtudiant (Etudiant etudiant) {
        liste.add(etudiant);
        return etudiant;
    }

    @DeleteMapping(value = "/delete")
    public void supprimerEtudiant(int identifiant){
        liste.remove(identifiant);
    }

    @PutMapping(value =  "/modifier")
    public void modifierEtudiant(int identifiant, String nom) {
        liste.get(identifiant).setNom(nom);
    }

    @GetMapping(value="/b")
    public String bonjour(){
        return "bonjour";
    }

    @GetMapping(value="/bn")
    public String bonsoir(){
        return "bonsoir";
    }

    @GetMapping(value="/etudiant")
```

```
    public Etudiant getUnEtudiant(){
        return new Etudiant(1,"A",19);
    }

    @GetMapping(value="/somme")
    public double somme(double a , double b){
        return a + b;
    }

}
```

Listing 4 – Contrôleur REST MyApi – Partie 1

Explication des annotations de la partie 1

`@RestController`
- Combine `@Controller` et `@ResponseBody`.
- Indique que la classe expose des endpoints REST.
- Toutes les méthodes renvoient directement la réponse dans le corps HTTP, généralement en JSON (pas de vue HTML).

`@GetMapping`
- Spécifie qu'une méthode répond aux requêtes HTTP de type `GET`.
- L'attribut `value="/liste"` indique l'URL.
- Exemple : `@GetMapping("/liste")` expose l'URL `/liste` en GET.

`@PostMapping`
- Spécifie qu'une méthode répond aux requêtes HTTP `POST`.
- Utilisée pour **créer** ou **ajouter** une ressource (comme un nouvel étudiant).

`@PutMapping`
- Spécifie qu'une méthode répond aux requêtes HTTP `PUT`.
- Utilisée pour **mettre à jour** une ressource existante (par exemple, modifier le nom d'un étudiant).

`@DeleteMapping`
- Spécifie qu'une méthode répond aux requêtes HTTP `DELETE`.
- Utilisée pour **supprimer** une ressource (ici, supprimer un étudiant de la liste).

Endpoints principaux sur la ressource `Etudiant`

- **`GET /liste`** : renvoie la liste de tous les étudiants.
- **`GET /getEtudiant?identifiant=...`** : renvoie l'étudiant correspondant à l'index donné.
- **`POST /addEtudiant`** : ajoute un étudiant (paramètres dans la requête).
- **`DELETE /delete?identifiant=...`** : supprime l'étudiant à l'index donné.
- **`PUT /modifier?identifiant=...&nom=...`** : modifie le nom de l'étudiant.

2.6 Tests de la première partie avec Postman

L'application étant lancée sur le port `9999`, la base des URLs est :

`http://localhost:9999`

Les différents endpoints (`/liste`, `/getEtudiant`, `/addEtudiant`, etc.) ont été testés avec Postman en utilisant les méthodes HTTP correspondantes (GET, POST, PUT, DELETE). Les réponses sont renvoyées au format JSON ou sous forme de texte simple pour les méthodes de test (`/b`, `/bn`, `/somme`).

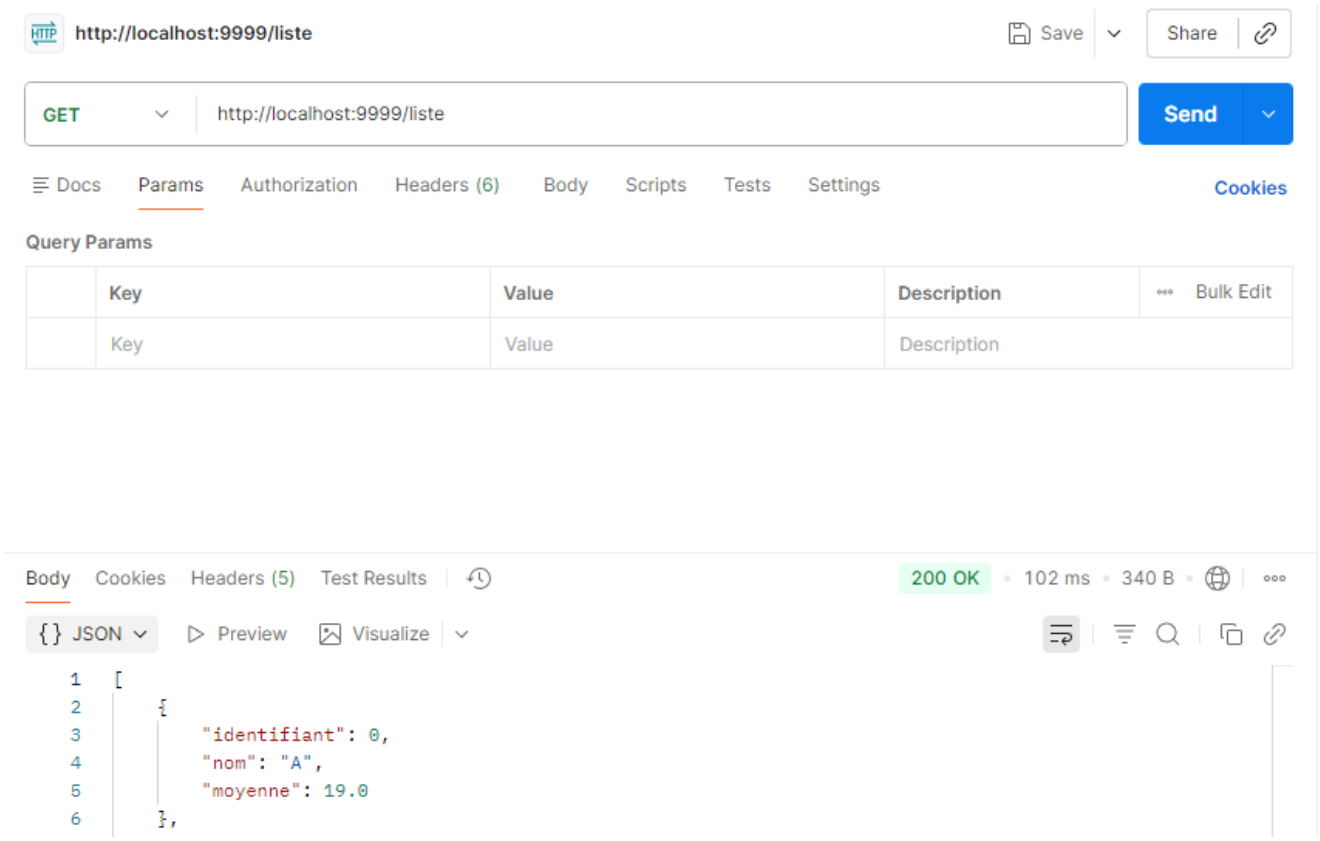

FIGURE 1 – Exemple de test avec postman

3 Deuxième partie : persistance avec JPA et MySQL

3.1 Objectif

La deuxième partie du TP consiste à remplacer la liste en mémoire par une **vraie base de données relationnelle**. Les principaux objectifs sont :

- Configurer une base MySQL (via XAMPP).
- Utiliser **Spring Data JPA** pour gérer la persistance.
- Définir une entité `Adherent` mappée sur une table.
- Initialiser quelques données au démarrage.

3.2 Entité `Adherent` et annotations JPA

```
package com.galille.tp2sae_h2.entities;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
```

```java
public class Adherent {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nom;
    private String ville;
    private int age;

    public Adherent(){

    }

    public Adherent(Long id, String nom, String ville, int age) {
        this.id = id;
        this.nom = nom;
        this.ville = ville;
        this.age = age;
    }
    // Getters et Setters ..

}
```

Listing 5 – Entité JPA Adherent

Explication des annotations de la partie 2 (JPA)

`@Entity`

- Indique que la classe `Adherent` est une entité JPA.
- Une table (par défaut `adherent`) sera créée dans la base de données pour stocker les instances de cette classe.

`@Id`

- Indique le champ qui joue le rôle de **clé primaire** dans la table.
- Ici, le champ `id` identifie de manière unique chaque adhérent.

`@GeneratedValue`

- Indique que la valeur de la clé primaire est générée automatiquement.
- `strategy = GenerationType.AUTO` laisse JPA choisir la stratégie adaptée (souvent auto-incrément côté base).
- Lorsqu'on sauvegarde un `Adherent` avec `id = null`, la base de données attribue automatiquement une nouvelle valeur.

3.3 Repository **`AdherentRepository`**

```java
package com.galille.tp2sae_h2.repository;

import com.galille.tp2sae_h2.entities.Adherent;
import org.springframework.data.jpa.repository.JpaRepository;

public interface AdherentRepository extends JpaRepository<Adherent, Long> {

}
```

Listing 6 – Interface AdherentRepository

Annotation et rôle de `JpaRepository`

Même s'il n'y a pas d'annotation directement dans cette interface, le fait d'étendre `JpaRepository<Adhere Long>` permet à Spring d'en faire un composant géré automatiquement. Spring détecte cette interface grâce au `@EnableJpaRepositories` activé par `@SpringBootApplication`.

Cette interface fournit :

- `findAll()` : récupérer tous les adhérents,
- `findById(Long id)` : récupérer un adhérent par identifiant,
- `save(Adherent a)` : insérer ou mettre à jour,
- `deleteById(Long id)` : supprimer par identifiant, etc.

3.4 Classe principale, `@SpringBootApplication` et `@Bean`

```
package com.galille.tp2sae_h2;

import com.galille.tp2sae_h2.entities.Adherent;
import com.galille.tp2sae_h2.repository.AdherentRepository;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class Tp2SaeH2Application {

    public static void main(String[] args) {
        SpringApplication.run(Tp2SaeH2Application.class, args);
    }

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

Listing 7 – Classe principale Tp2SaeH2Application

Explication de l'annotation `@Bean`

- `@Bean` indique que la méthode renvoie un objet qui doit être géré par le conteneur Spring.
- Ici, la méthode `runner` renvoie un `CommandLineRunner` qui sera exécuté automatiquement au démarrage de l'application.
- Ce `CommandLineRunner` utilise `AdherentRepository` pour insérer des données de test dans la base.

3.5 Configuration `application.properties`

```
spring.application.name=TP2SAE_H2
server.port=9191
spring.datasource.url=jdbc:mysql://localhost:3306/adherent
```

```
spring.datasource.username=root
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=create
```

Listing 8 – Configuration Spring Boot – Partie 2

Les propriétés principales sont :

- `server.port=9191` : l'application écoute sur le port 9191.
- `spring.datasource.url` : URL JDBC de la base MySQL `adherent`.
- `spring.datasource.username=root` : utilisateur MySQL.
- `spring.datasource.password=` : mot de passe (ici vide).
- `spring.jpa.hibernate.ddl-auto=create` : indique à Hibernate de recréer le schéma à chaque démarrage à partir des entités annotées avec `@Entity`.

3.6 Dépendances Maven et lien avec les annotations

Le fichier `pom.xml` contient notamment les dépendances suivantes :

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webmvc-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-h2console</artifactId>
    </dependency>

    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

Listing 9 – Extrait du pom.xml

Ces dépendances activent automatiquement, grâce à `@SpringBootApplication`, tout le support des annotations utilisées :

- `spring-boot-starter-web` : fournit les annotations REST (`@RestController`, `@GetMapping`, etc.).
- `spring-boot-starter-data-jpa` : fournit les annotations JPA (`@Entity`, `@Id`, `@GeneratedValue`) et la gestion des repositories.
- `mysql-connector-j` : permet à JPA/Hibernate de se connecter à MySQL via l'URL de la datasource.

4 Conclusion

Ce TP m'a permis de :

- Comprendre les principes de base des **API REST** (ressources, verbes HTTP, annotations Spring MVC).
- Mettre en place une **première API REST en mémoire** avec Spring Boot.
- **Intégrer une base MySQL** grâce à Spring Data JPA et aux annotations JPA (`@Entity`, `@Id`, `@GeneratedValue`).
- Voir concrètement la différence entre un stockage en mémoire (liste Java) et une **persistance réelle en base**.

Chaque annotation utilisée dans le code (`@SpringBootApplication`, `@RestController`, `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, `@Entity`, `@Id`, `@GeneratedValue`, `@Bean`) a un rôle précis dans la configuration de l'application, l'exposition des endpoints REST et la gestion de la persistance des données.