



BASE DE DONNÉES AVANCÉE

Rapport TP3 - Développement d'un micro-services

Élèves :

Aïssa PANSAN

11933936

Enseignant :

Samir YOUSSEF

30 Mars 2025

1 Initialisation

Pour réaliser une micro-service en SpringBoot, on peut utiliser plusieurs IDE comme **IntelliJ Idea**. Cela dit, cela n'a pas marché pour (malgré plusieurs essais, je vous le promets). **Dans la suite de ce rapport, l'ensemble des opérations seront effectuées dans l'IDE Eclipse.**

Les premières étapes sont les mêmes que celle de la vidéo, on peut les faire sur **IntelliJ** à la différence du choix des versions.

- On choisit la version 21 de Java.
- On utilise le package Jar, pour déployer sans un serveur d'application. C'est comme un zip.
- On doit utiliser la dépendance **Spring Web**, on peut enfin valider.

Remarque : Un projet Spring est un projet Maven

Ensuite, en ligne de commande. On va initialiser le projet Maven via la commande **mvn install**

C'est à partir de cette étape, que l'on utilisera Eclipse.

2 Écriture d'une première API REST

SpringBoot intègre directement Tomcat. On n'a pas besoin de paramétrer l'IDE pour lancer Tomcat, ni même l'installer. On va lancer notre application **src/main/java/com/example/**. On accède à l'application via localhost sur le port 8080

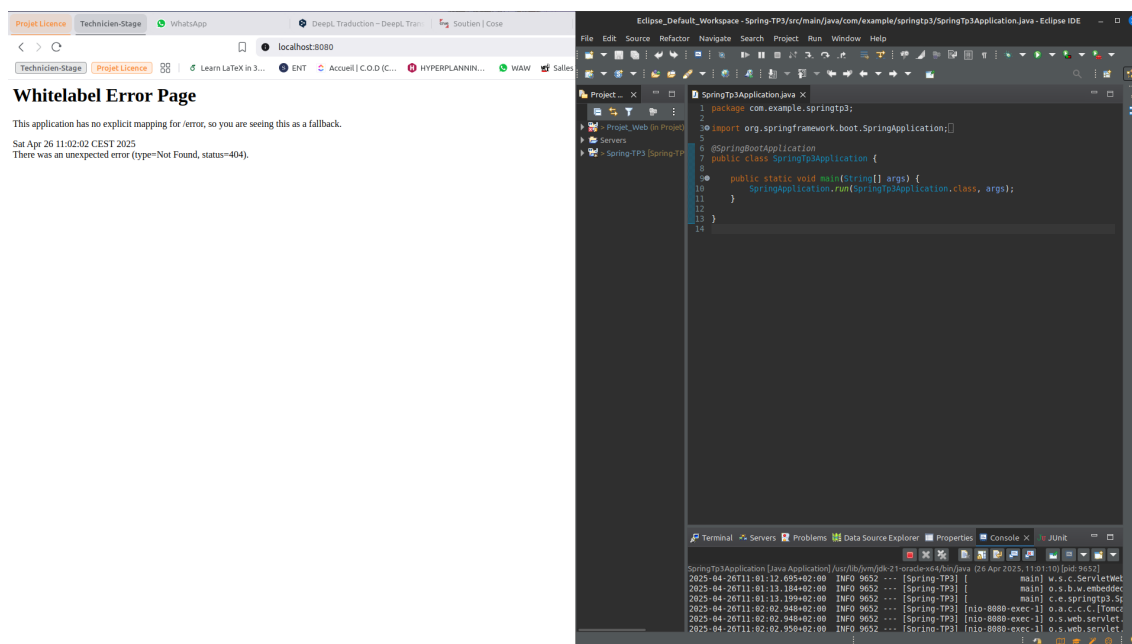


FIGURE 1 – Illustration du lancement de SpringBoot

Les micro-services doivent être écrit au niveau de **com.example.[Nom_projet]**.

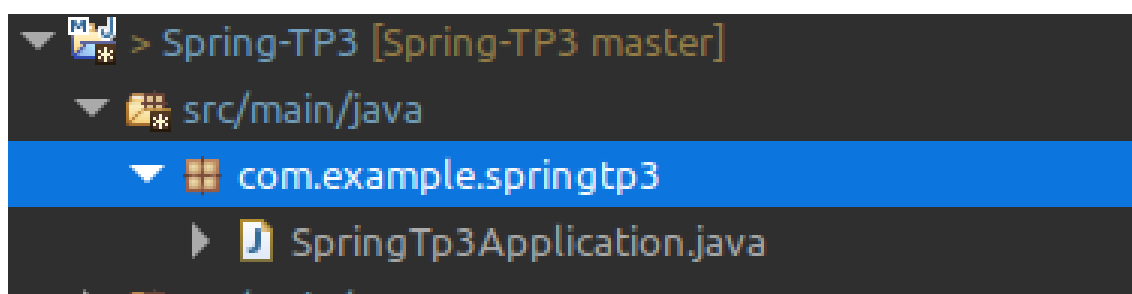


FIGURE 2 – Localisation dans Eclipse

On définit une nouvelle app via deux annotations primordiales.

- **@RestController** : Pour définir une API REST. Elle se met au niveau de la nouvelle classe
- **@GetMapping(value = "/bn")** : Comme une Servlet. On définit la HTTP Response en précisant l'URL d'accès. Chaque méthode peut représenter une page.

Attention : Il faut arrêter la SpringBootApplication pour appliquer les changements. Relancer l'app, alors quelle est déjà lancé ne marchera pas. Il faudra relancer Eclipse pour arreter l'app une bonne fois pour toutes. Pour eviter tout ça arrete l'app via la console.

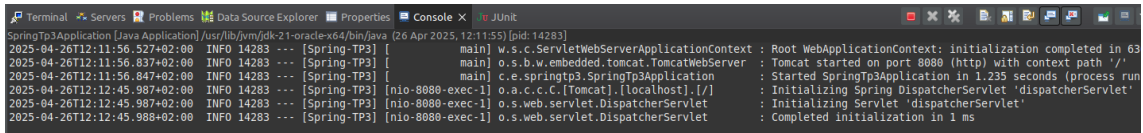


FIGURE 3 – Arrêt de l'application via la console

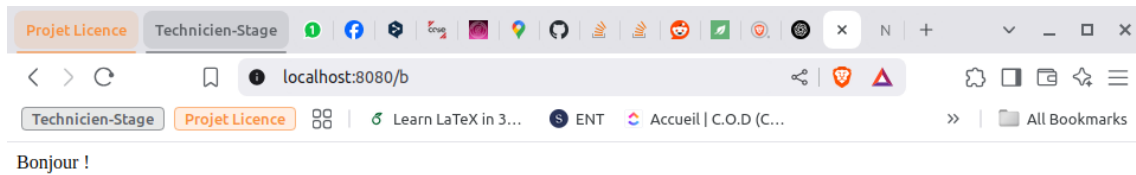


FIGURE 4 – Résultats

2.1 Changer le port d'écoute de l'application

Pour changer le port d'écoute. On modifie le fichier dans `src/main/ressources/main.properties` et on écrit : `server.port=9999`

2.2 Afficher les informations d'un objet

On va générer une nouvelle classe `Étudiant` avec les attributs `id`, `nom` et `moyenne`. Jusqu'à là c'est du Java classique (Constructeur par défaut, Constructeur normal, Getters et Setters). On peut afficher les attributs d'un objet en retournant directement l'objet d'une classe dans une méthode qu'on mappe. L'affichage se fera sous format JSON.

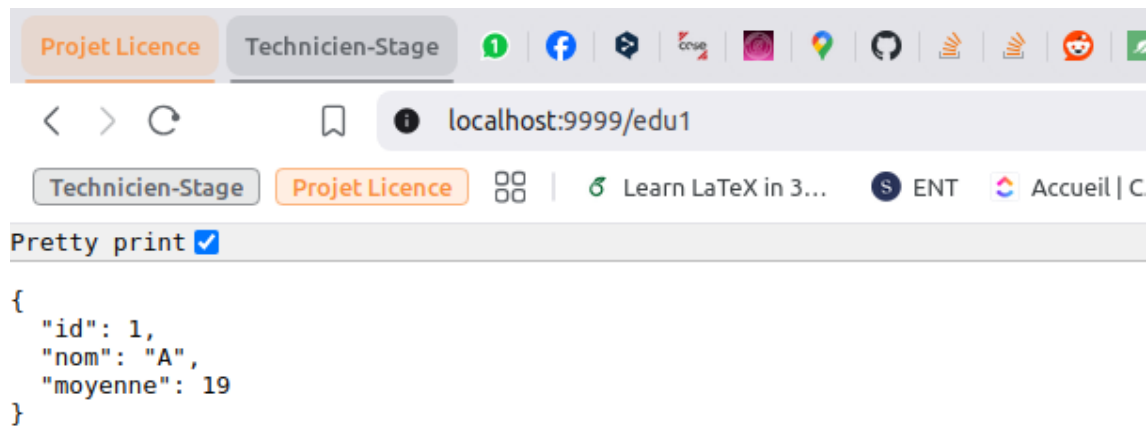


FIGURE 5 – Résultats

3 Tester les micro-services

Plusieurs outils existent pour tester un micro-services (PostMan, DeepLogin, SoapUI). Dans notre cas, je vais utiliser SOAPUI, car il est déjà présent dans ma machine.

Pour tester un micro-services. On va écrire une fonction avec l'endpoint(/**somme**) qui retourne la somme de deux nombres fournis en paramètre. Au niveau de l'URL, on doit passer les deux paramètres comme-ci : `/somme?a=1&b=1`. On peut tester manuellement via le navigateur, mais SOAPUI nous permet de le faire via une interface graphique.

foo_bar

3.1 Etape 1 : Choisir un projet REST

L'URI est le début de l'URL, c'est à dire : `http://localhost`

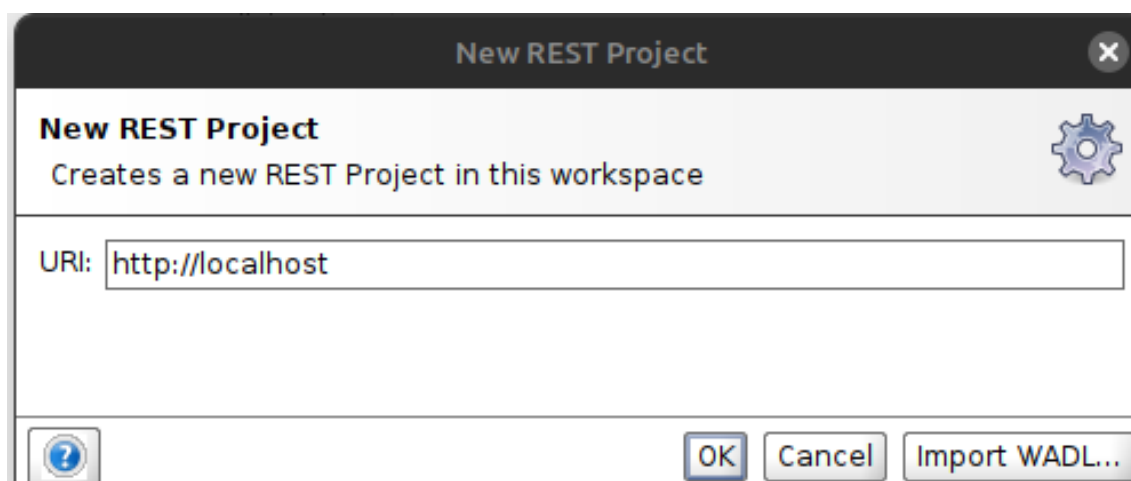


FIGURE 6 – On ne précise pas encore le port

3.2 Etape 2 : Tester

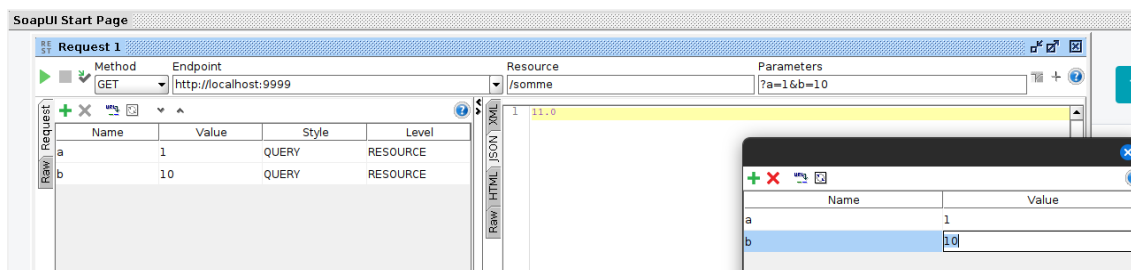


FIGURE 7 – Interface SoapUI

Il y a trois champs important dans SoapUI :

- **Entrypoint** : Le point d'entrée de l'app. C'est le début de l'URL
- **Ressources** : C'est la page spécifique auquel on veut accéder
- **Parameters** : Les paramètre à envoyer (HTTP Response)

On peut sauvegarder au fur et à mesure pour garder les paramètres de la requêtes.

Remarque : Les requêtes sont en GET

4 Persistance des informations via une ArrayList

Une étape phare d'une API REST est le fait de sauvegarder de manière pérenne les informations. Pour le moment, on va utiliser une collection d'étudiant statique qui sera

stocké dans une ArrayList.

```
public static Collection<Etudiant> liste = new ArrayList();
```

On va ensuite ajouter au fur et à mesure des étudiants dans la liste via **liste.add()**.
Tout se fait en static.

Enfin, on fait une nouvelle méthode avec l'endpoint **/list** qui retourne la collection.

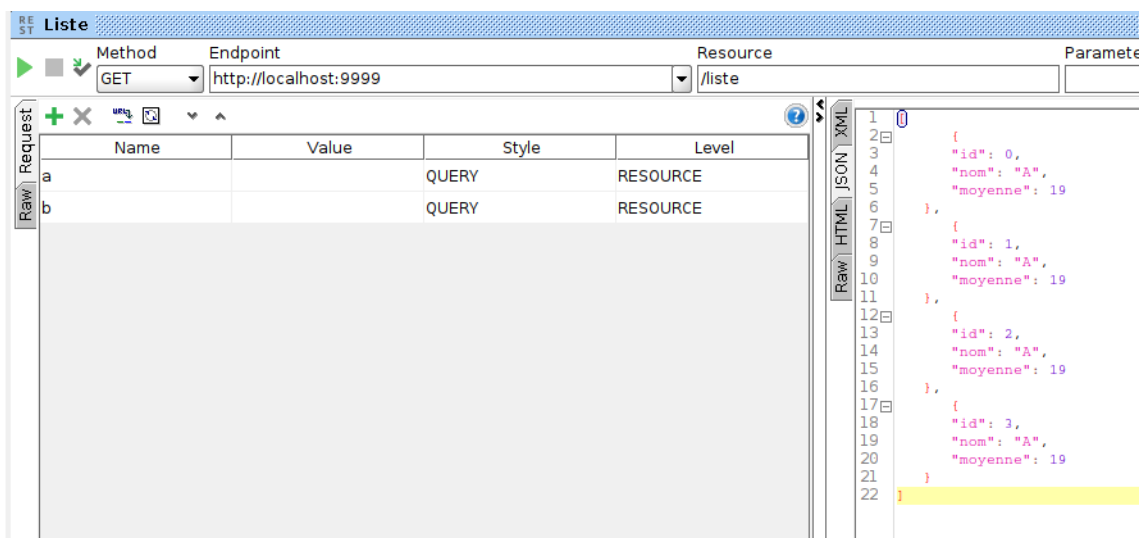


FIGURE 8 – Affichage des élèves en JSON

4.1 Affichage d'un étudiant via son identifiant

On peut retourner un élève en fonction de son identifiant. En faisant, une nouvelle méthode **getEtudiant**, on doit préciser l'id de l'étudiant via l'URL (Comme auparavant).

Si l'étudiant existe, on peut voir ses informations sinon, il y a une erreur.

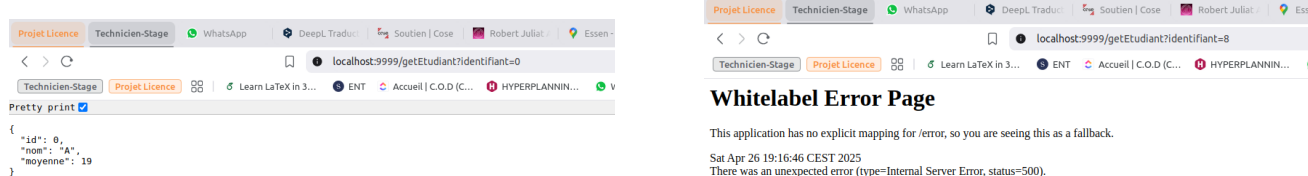


FIGURE 9 – Etudiant 0

FIGURE 10 – Etudiant introuvable

4.2 Ajouter des étudiants

Rappel des type de requête HTTP :

- GET : Renvoie une ressources
- POST : Générer une ressources
- PUT : Modifier une ressources
- DELETE : Supprimer une ressources

Remarque : C'est une convention, on peut tout à fait faire le travail de l'une avec l'autre.

On va utiliser **POST**, pour faire une méthode pour ajouter un étudiant dans la liste. On commence à écrire une méthode qui prend en argument un étudiant. Les attributs de l'étudiant sont directement passé dans l'URL. L'entrypoint est **addEtudiant**. Via SoapUI, on peut ajouter un étudiant.

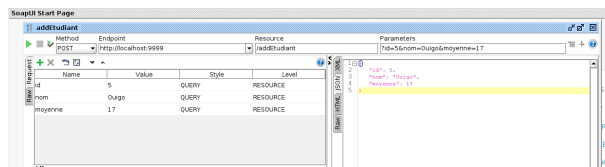


FIGURE 11 – Ajout de l'étudiant 5

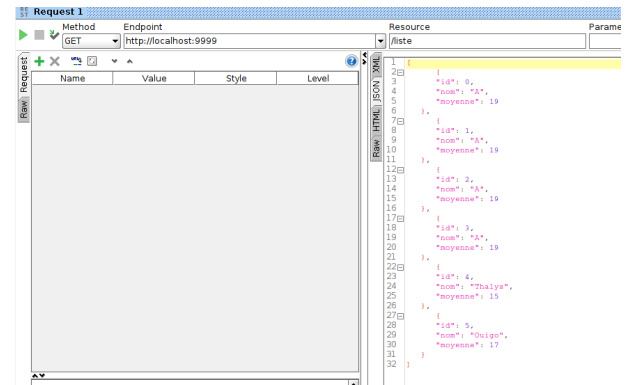


FIGURE 12 – Liste de tous les étudiants

On peut dans le même esprit écrire des méthodes pour modifier et supprimer un élève avec les requetes PUT et DELETE.

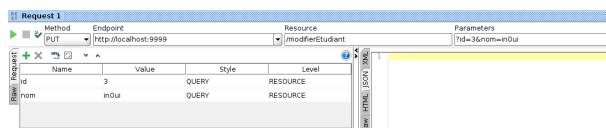


FIGURE 13 – Modification de l'étudiant 3

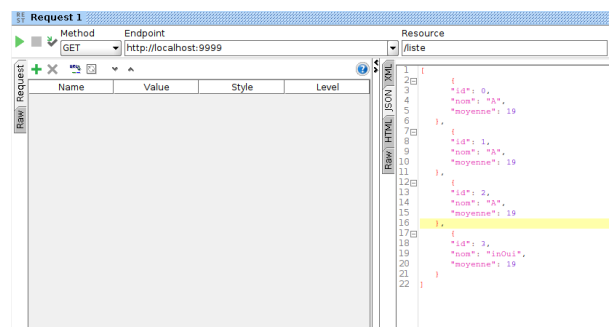


FIGURE 14 – Liste de tous les étudiants

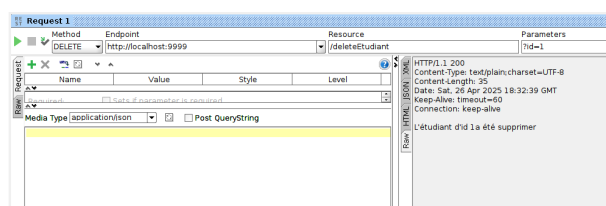


FIGURE 15 – Suppression de l'étudiant 1

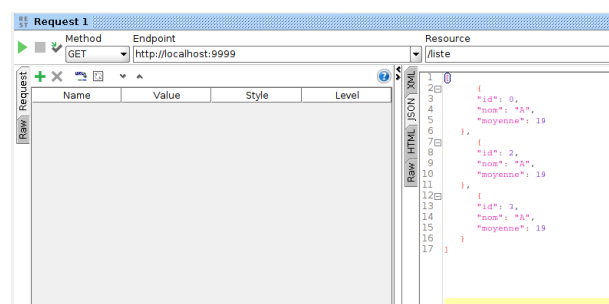


FIGURE 16 – Liste de tous les étudiants

Remarque : Les changements sont persistant selon la durée de vie du processus du SpringBootApplication. Son arrêt entraîne la suppression de tout ce qu'il y a dans la RAM.

5 Persistance avec base de donnée H2

La partie suivante consiste à utiliser le controleur JPA pour faire l'interfaçage entre le Java et le SGBD **H2**. **H2** est un SGBD écrit entièrement en Java. Il est destiné au devs et non à la production. De ce point de vue, H2 à des similitudes avec **SQLite**. Cette partie à pu marcher avec l'IDE **IntelliJ** (Je ne sais pas pourquoi).

5.1 Installation

On va générer un nouveau projet. On reprend les mêmes étapes précédentes à la différence des dépendances. On va rajouter les dépendances **H2 Database** et **Spring Data JPA**.

Remarque : Pour mon cas, j'ai du utiliser JDK 21 avec Java 17 pour m'assurer que le projet soit compatible avec Maven.

On va générer deux nouveau package dans **com.example.h2demo_tp3**. Le package **entities** qui représente les entités de notre application. Enfin, le package **repository**. Ce package permet d'organiser les accès à la base de donnée.

5.2 Micro-services avec Hibernate

Pour écrire notre nouvelle API REST. On va d'abord importer les bibliothèques **Hibernate** via les modules **jakarta.persistence** pour manipuler la base de donnée **H2**. Il est à noter qu'il existe d'autres contrôleur que **Hibernate**.

On commence par écrire une classe Java classique avec ces attributs et méthodes. Dans notre cas, on s'intéresse à un Adhérent. On doit préciser lequel des attributs est une clé primaire avec l'annotation **@Id**. De même, la classe **Adherent** doit être annoté par **@Entity** pour qu'une table soit généré.

On va ensuite définir une interface **AdherentRepository**. Cette interface va hériter de **JpaRepository<Adherent,Long>**. Le 1er paramètre doit être notre entité(Classe) tandis que le deuxième doit être le type de la clé primaire. La déclaration de cette interface va nous fournir un ensemble d'opérations **CRUD** (Create,Read,Update,Delete) de base sur nos données. L'avantage est que n'avons plus besoin de générer des méthodes pour manipuler nos entités (Comme vu auparavant). Ces méthodes sont directement générées.

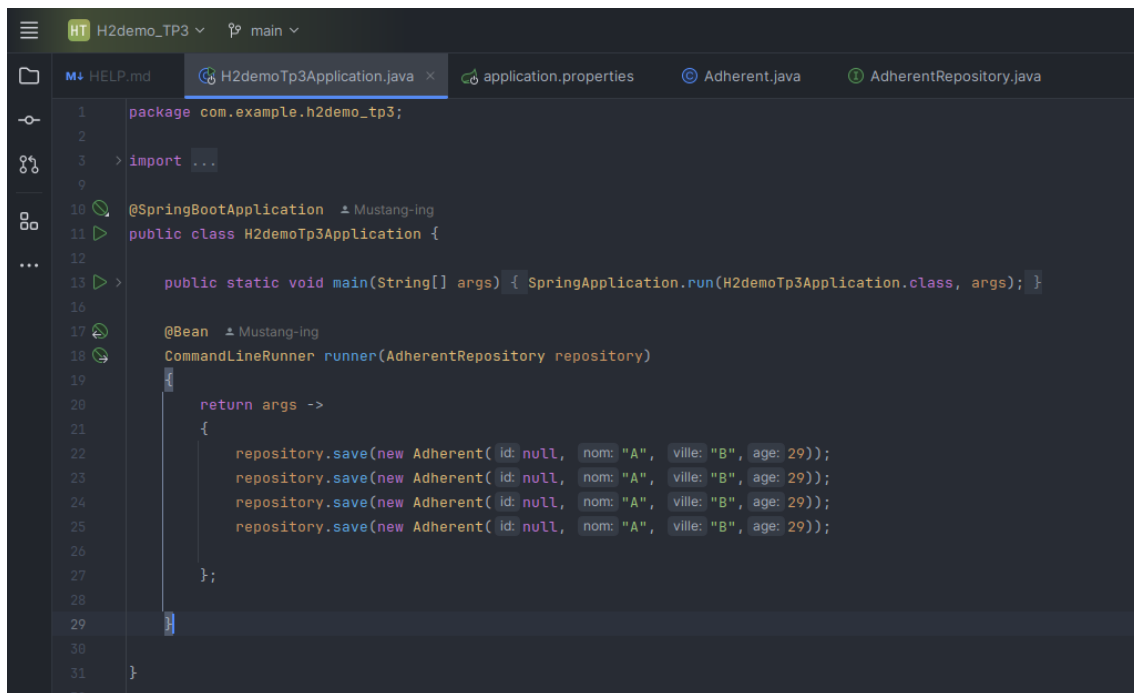
```
package com.example.h2demo_tp3.repository;

import com.example.h2demo_tp3.entities.Adherent;
import org.springframework.data.jpa.repository.JpaRepository;
```

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

Un exemple de ces bénéfices est la classe [Nom_projet]Application. Ici, on va pouvoir directement enregistrer des instances d'Adhérent dans la base de donnée en important l'interface. Spring va exécuter la méthode **runner de type CommandLineRunner** pendant l'exécution. On va ainsi dès le démarrage enregistrer 4 adhérents.

Rmq : La syntaxe `return -> args` est une fonction lambda (Anonyme) dans Java.



```
1 package com.example.h2demo_tp3;
2
3 > import ...
4
5
6
7
8
9
10
11 @SpringBootApplication
12 public class H2demoTp3Application {
13
14     public static void main(String[] args) { SpringApplication.run(H2demoTp3Application.class, args); }
15
16
17     @Bean
18     CommandLineRunner runner(AdherentRepository repository)
19     {
20         return args ->
21         {
22             repository.save(new Adherent( id: null, nom: "A", ville: "B", age: 29));
23             repository.save(new Adherent( id: null, nom: "A", ville: "B", age: 29));
24             repository.save(new Adherent( id: null, nom: "A", ville: "B", age: 29));
25             repository.save(new Adherent( id: null, nom: "A", ville: "B", age: 29));
26         }
27     }
28
29 }
```

FIGURE 17 – Application.java

Pour voir l'ensemble des ces résultats, on doit éditer le fichier **properties** dans res-sources. De cette manière :

```
1 spring.application.name=H2demo_TP3
2 server.port=9191
3 spring.datasource.url=jdbc:h2:mem:adherent
4 spring.h2.console.enabled=true
```

La ligne 3, précise l'URL de la JDBC. La ligne 4 active le mode console. C'est-à-dire le GUI du SGBD H2.

Après avoir lancé le serveur, on peut consulter l'interface de H2. Via l'url .

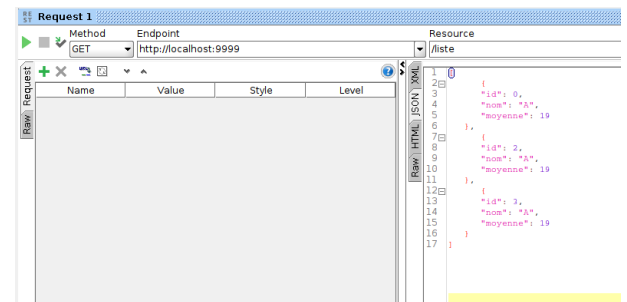
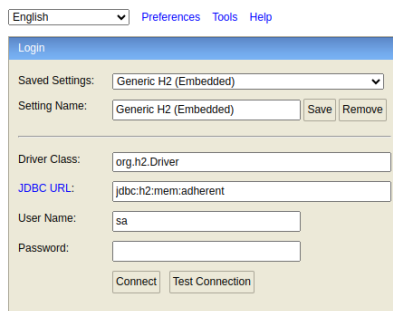


FIGURE 19 – Interface GUI de H2

FIGURE 18 – On peut appuyer sur **Test connection** pour tester la connection

On peut visualiser tous les adhérent avec une requête SQL.

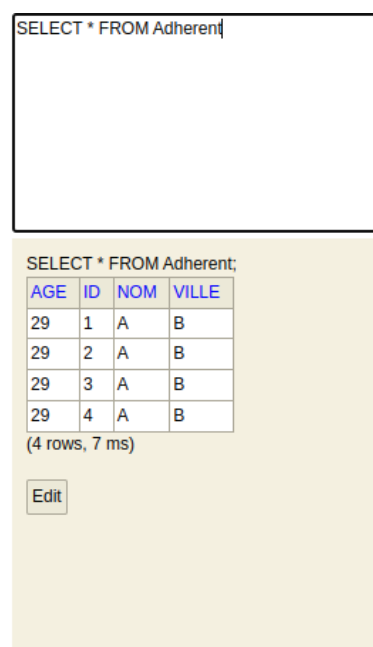


FIGURE 20 – Application.java