TP — Manipulation d'un projet Spring Boot

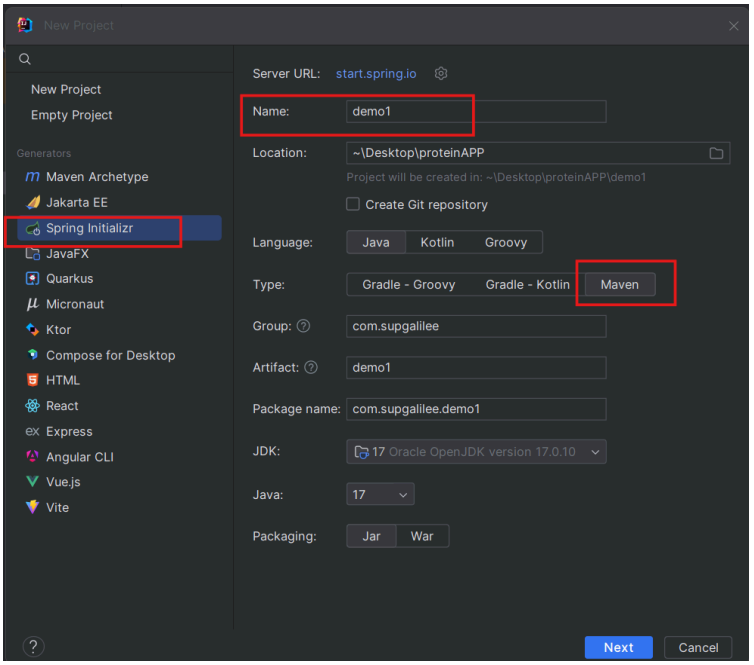
Ce TP vise à manipuler un projet **Spring Boot**, en découvrant ses **différentes couches logiques** (Controller, DAO, Service, etc.), en créant des API REST, et en gérant des données à travers une base H2 ou en mémoire. L'objectif est de comprendre **le rôle de chaque couche** et comment elles collaborent.

Couche	Rôle	Contenu Physique
Controller	Interface entre le front-end (ou client) et l'application	Gère les requêtes HTTP (GET, POST, etc.)
Service *(optionnelle)*	Contient la logique métier	Gère les requêtes HTTP (GET, POST, etc.)
DAO / Repository	Accès aux données	Interagit avec la base de données via JPA
Model (ou Entity)	Représente les objets métiers	Classes Java annotées @Entity

Partie 1 : Création du projet Spring Boot pour manipuler la couche DAO et le méthodes REST (CRUD)

Opération	Annotation Spring	Verbe HTTP	Description
Create	@PostMapping	POST	Ajouter une nouvelle ressource
Read (all)	@GetMapping	GET	Récupérer toutes les ressources
Read (one)	@GetMapping	GET	Récupérer une ressource par son identifiant
Update	@PutMapping	PUT	Modifier une ressource existante
Delete	@DeleteMapping	DELETE	Supprimer une ressource via son identifiant

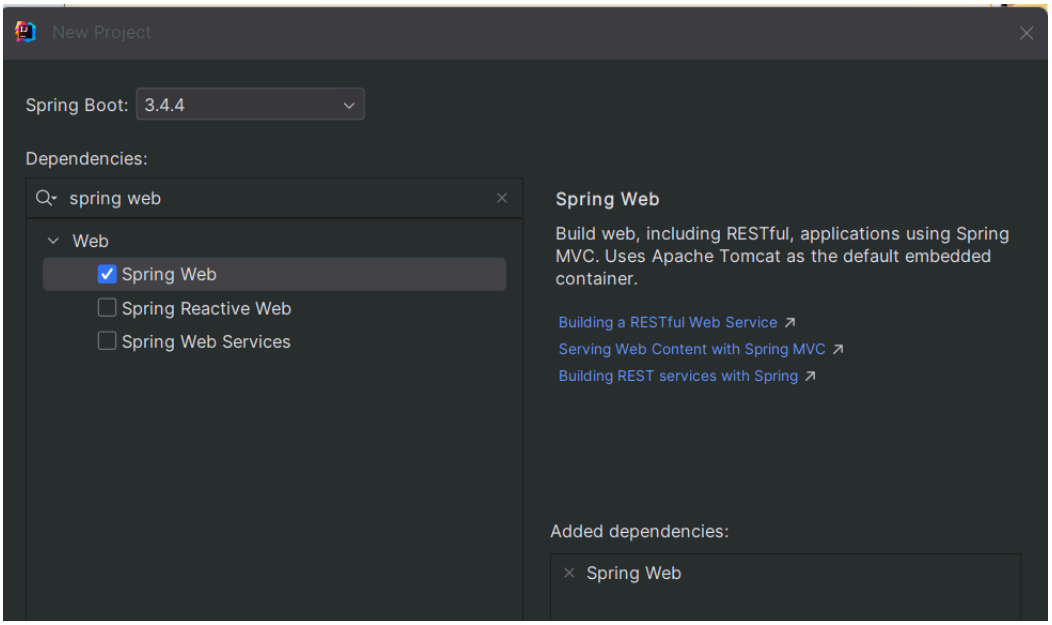
On commence par créer un projet via **Spring Initializr** (IntelliJ IDEA) avec :



Explication : Pourquoi utiliser un projet Maven avec Spring Boot ?

→ Un projet Maven avec Spring Boot permet de minimiser les configurations manuelles grâce à la gestion automatique des dépendances via le fichier **pom.xml**. Dès sa création, il est prêt à recevoir des API REST, avec un serveur Tomcat embarqué et une structure standardisée qui simplifie le développement.

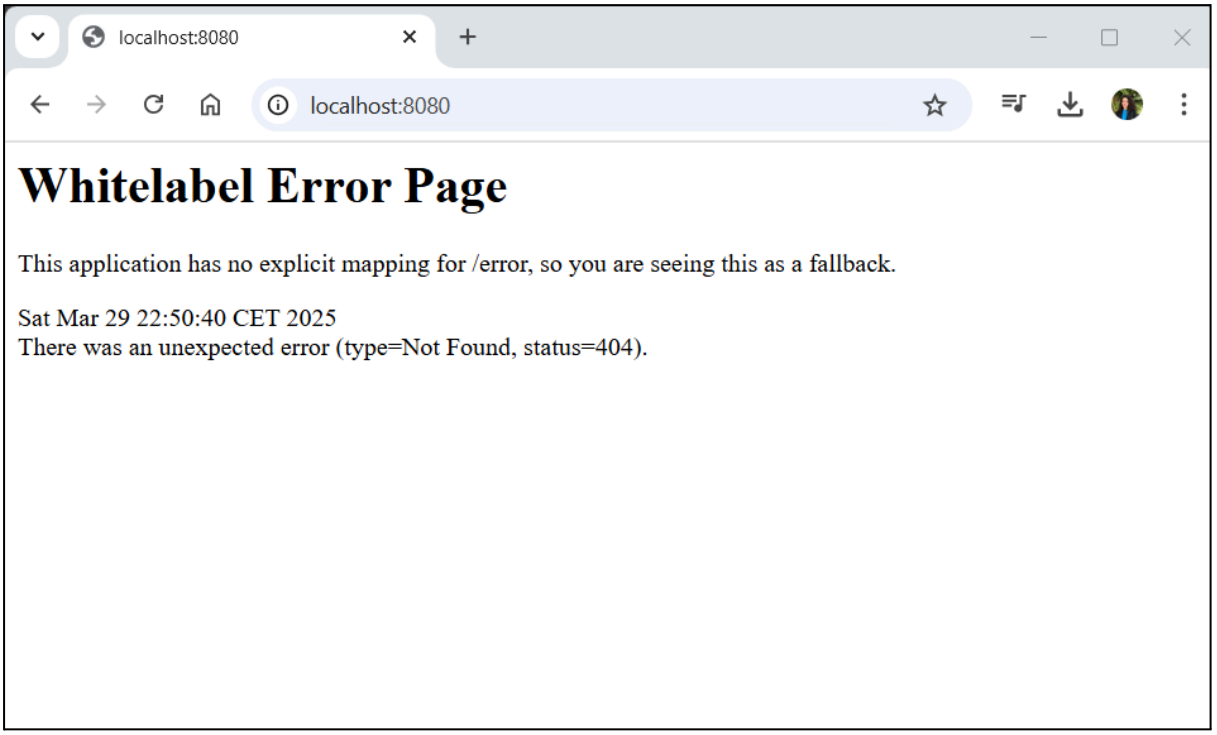
Aprés, nous avons choisi la dépendance **spring web** pour créer des API REST, gérer les requêtes HTTP, intégrer Tomcat automatiquement et convertir les données en JSON.



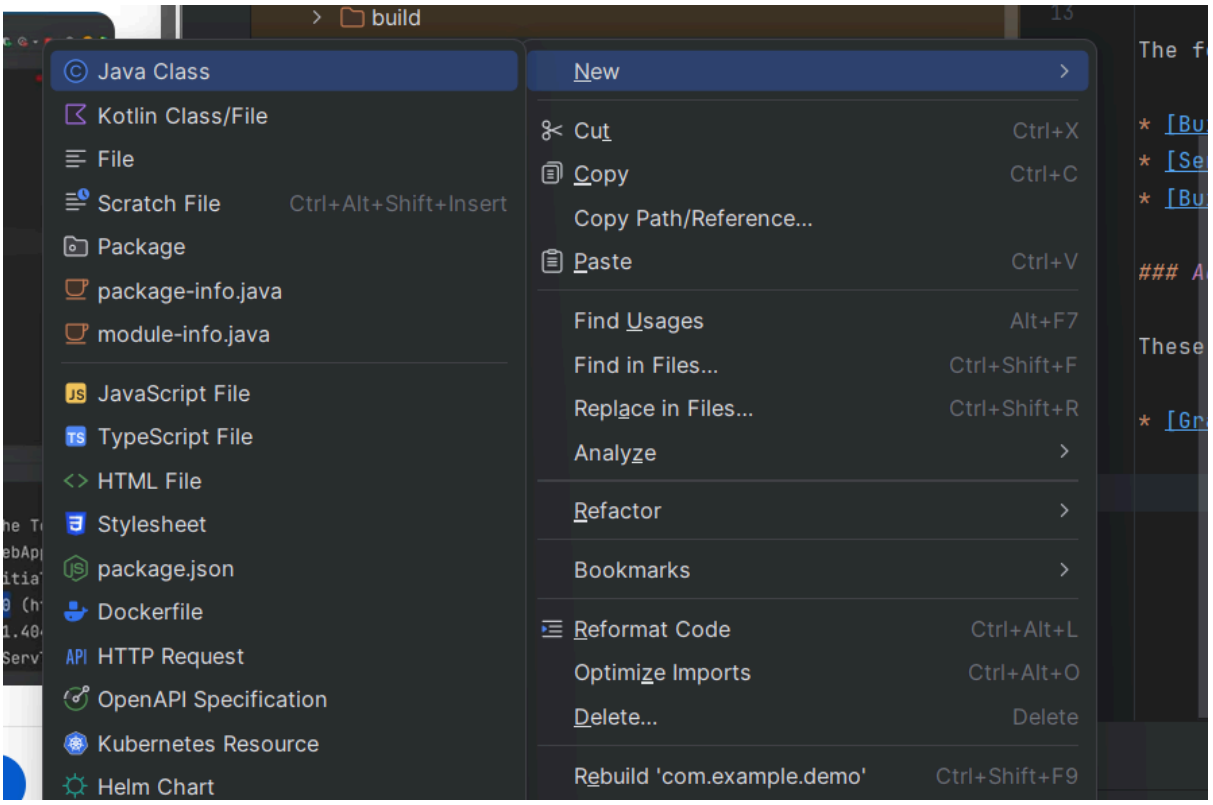
On exécute le projet en cliquant sur **Run** afin de vérifier que l'application démarre correctement et que la configuration est fonctionnelle.

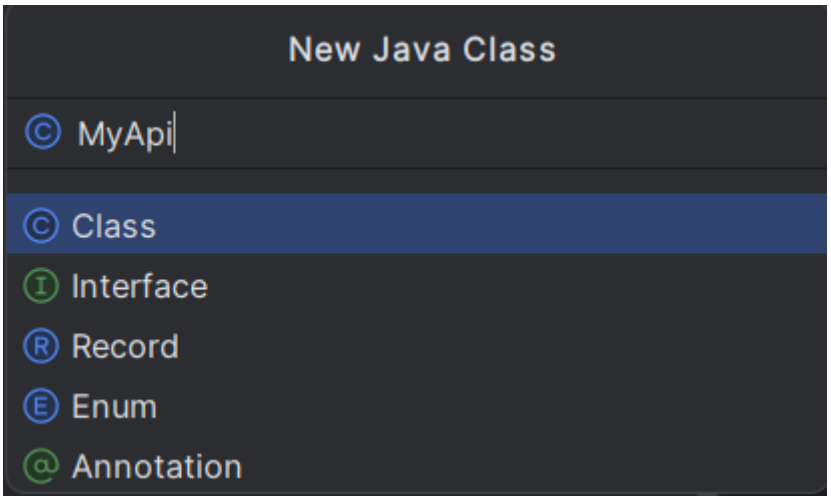
```
2025-03-29T22:48:17.213+01:00  INFO 6288 --- [demo] [           main] com.example.demo.DemoApplication         : Starting DemoApplication using Java 17.0.10 with PID 6288
2025-03-29T22:48:17.218+01:00  INFO 6288 --- [demo] [           main] com.example.demo.DemoApplication         : No active profile set, falling back to 1 default profile:
2025-03-29T22:48:18.236+01:00  INFO 6288 --- [demo] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8080 (http)
2025-03-29T22:48:18.260+01:00  INFO 6288 --- [demo] [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2025-03-29T22:48:18.262+01:00  INFO 6288 --- [demo] [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.39]
2025-03-29T22:48:18.332+01:00  INFO 6288 --- [demo] [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2025-03-29T22:48:18.333+01:00  INFO 6288 --- [demo] [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 10
2025-03-29T22:48:18.795+01:00  INFO 6288 --- [demo] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
2025-03-29T22:48:18.813+01:00  INFO 6288 --- [demo] [           main] com.example.demo.DemoApplication         : Started DemoApplication in 2.25 seconds (process running f
```

On ouvre la page par défaut à l'adresse **localhost:8080** pour s'assurer que le serveur est bien lancé et prêt à recevoir des requêtes.



Une fois le projet lancé avec succès, nous créons notre première classe MyApi, un **contrôleur REST** qui contiendra les fonctions nécessaires pour gérer les différentes opérations de cette première partie.

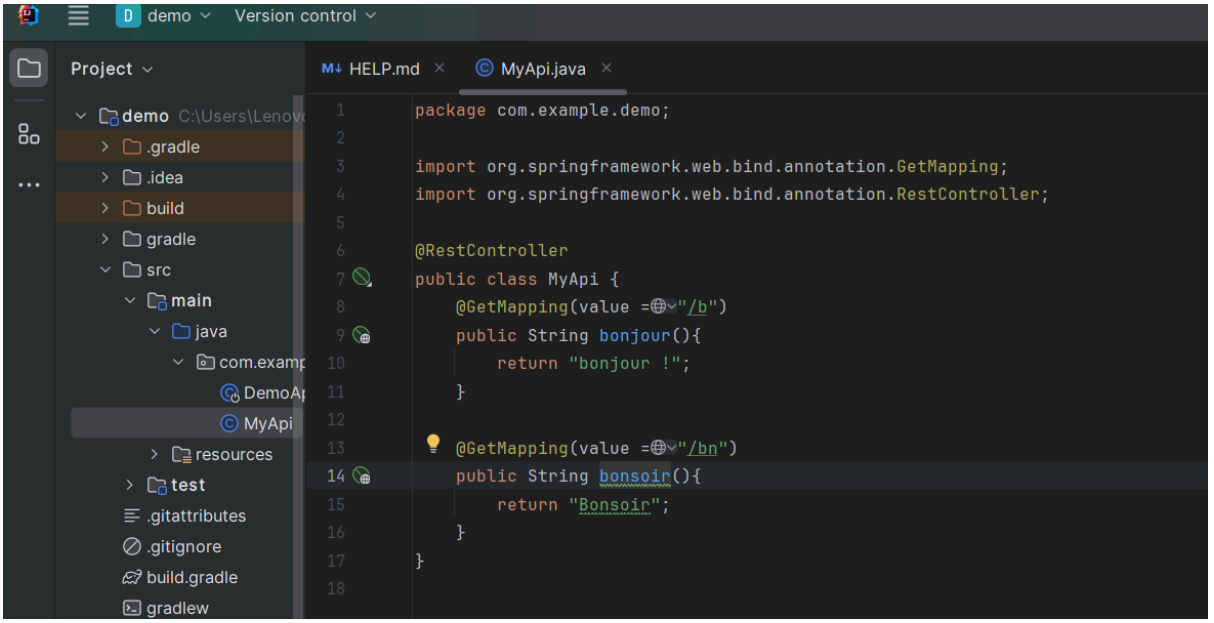




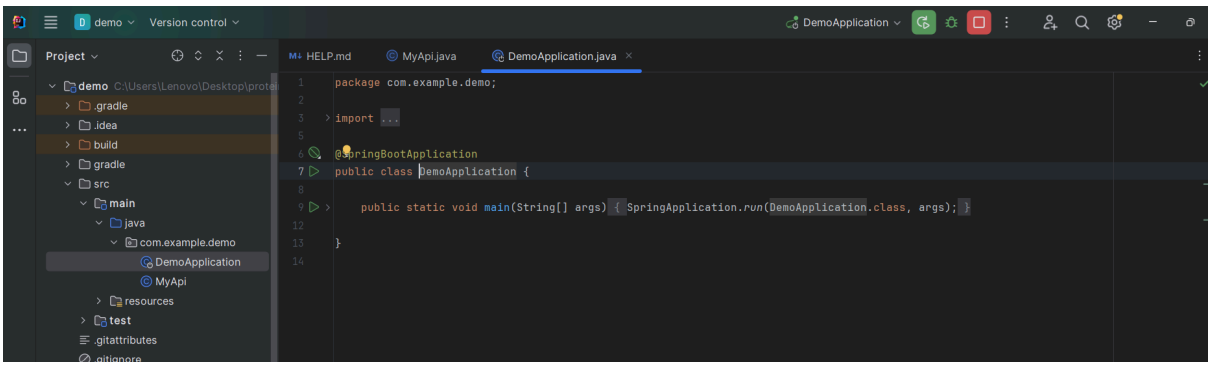
Pour assurer le bon fonctionnement du contrôleur, nous utilisons les annotations suivantes :

- **@RestController :** indique que la classe expose des endpoints REST et que les résultats seront automatiquement convertis en JSON.

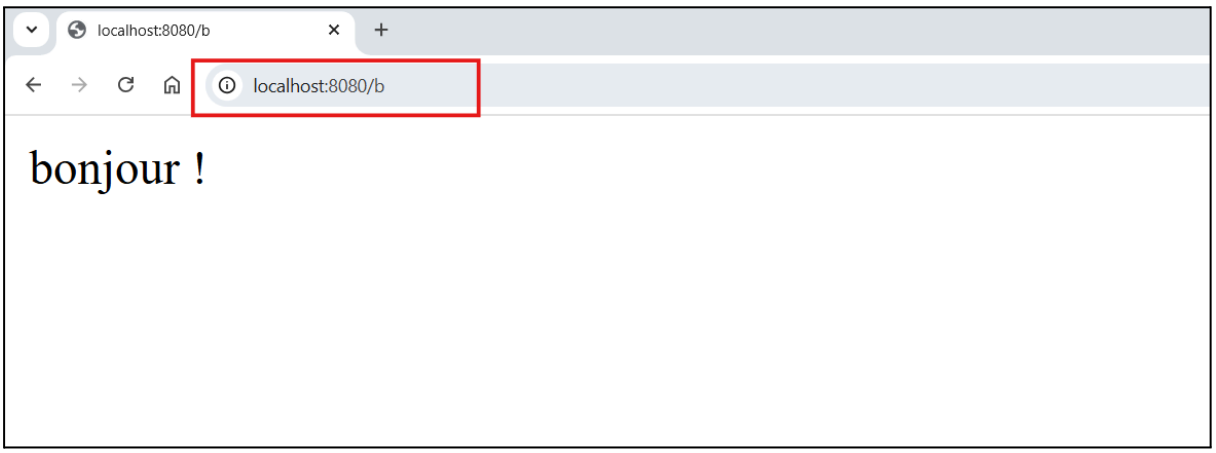
1. **Définition de deux fonctions GET : Bonjour et Bonsoir**

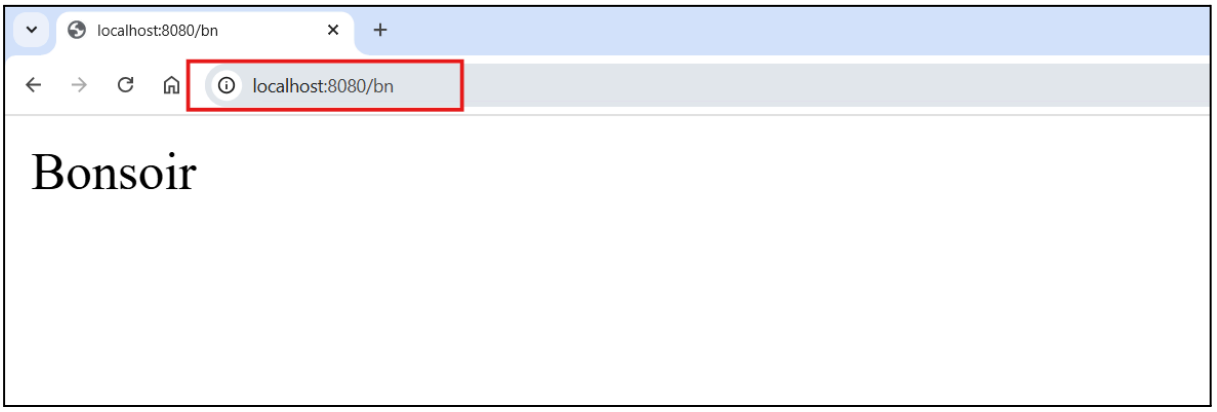


Nous lançons l'application pour test en exécutant la classe principale DemoApplication, qui contient la méthode main responsable du démarrage de Spring Boot et du serveur embarqué.



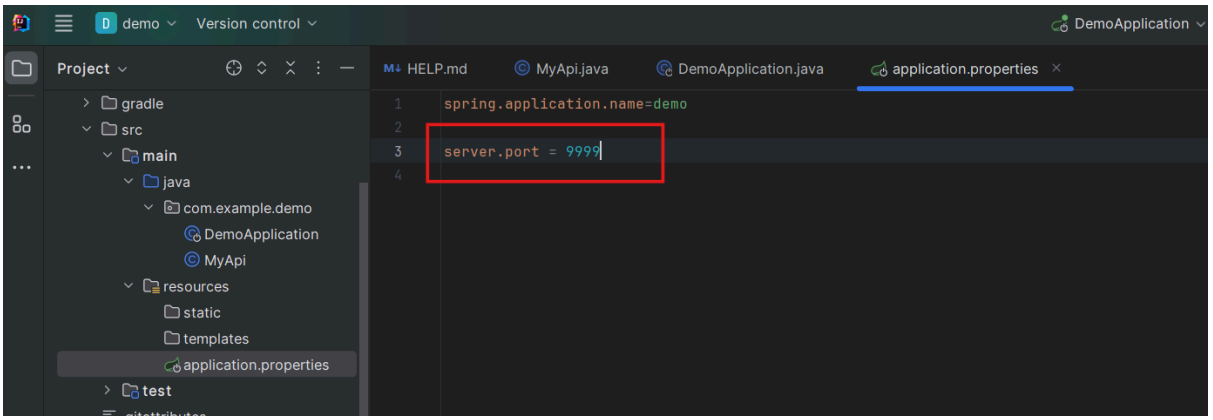
Résultat :





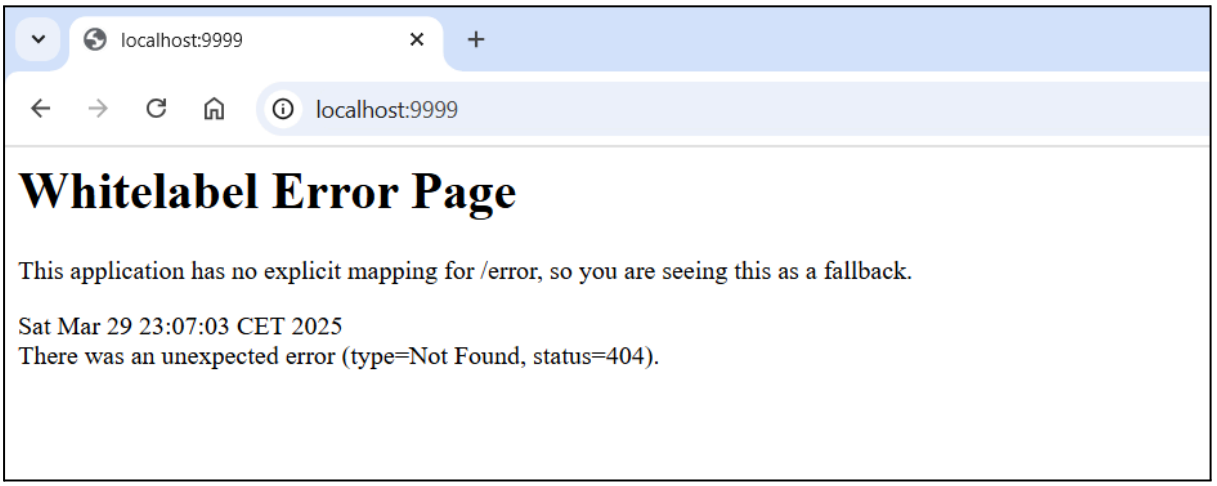
Changement du port d'écoute de l'application :

Le port d'écoute de l'application peut être modifié en configurant le fichier application.properties, en ajoutant par exemple :



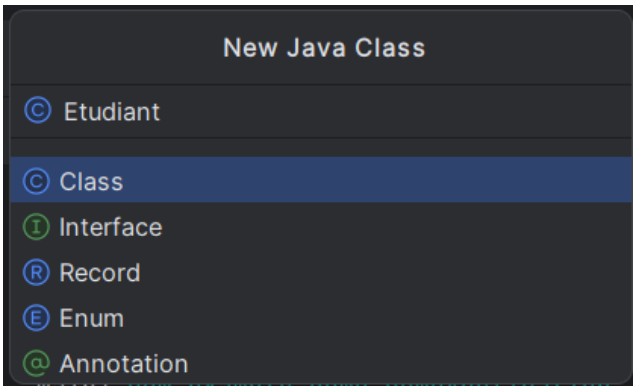
→ Cela permet de démarrer l'application sur un port différent de celui par défaut (8080).

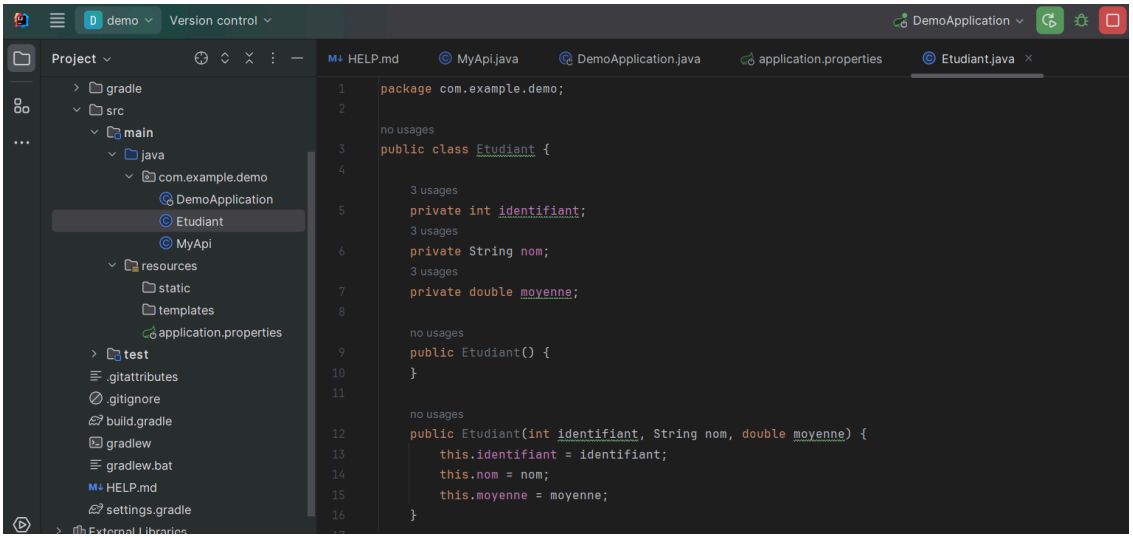
Vérification :

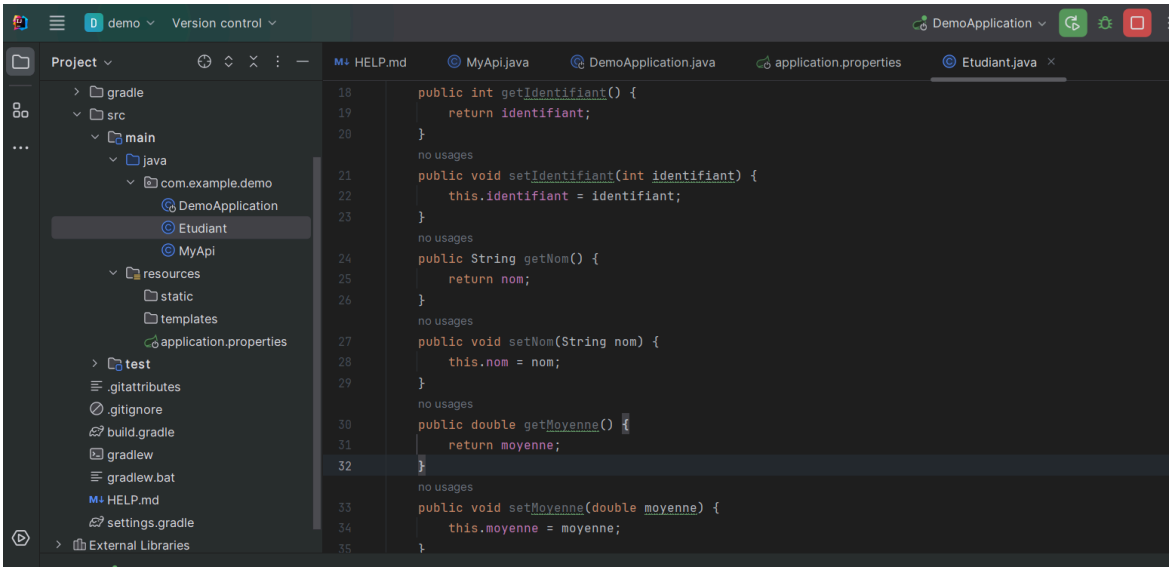


Création du premier objet : Etudiant

Java étant un langage orienté objet, nous allons créer notre première classe Etudiant, qui représente un objet métier contenant des attributs tels que id, nom, ville et âge. Cette classe servira de modèle de données pour les opérations à venir.





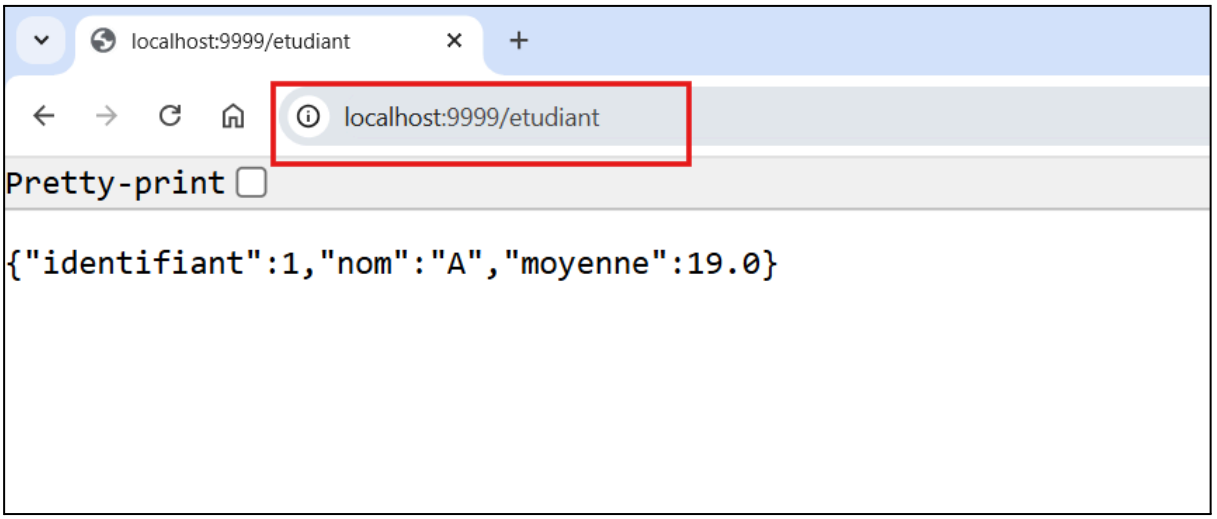


Après avoir défini la classe **Etudiant**, nous ajoutons un **constructeur avec paramètres**, un **constructeur vide**, ainsi que les **getters et setters** pour chaque attribut.
→ Cela permet de créer facilement des objets **Etudiant** et de manipuler leurs données dans l'application.

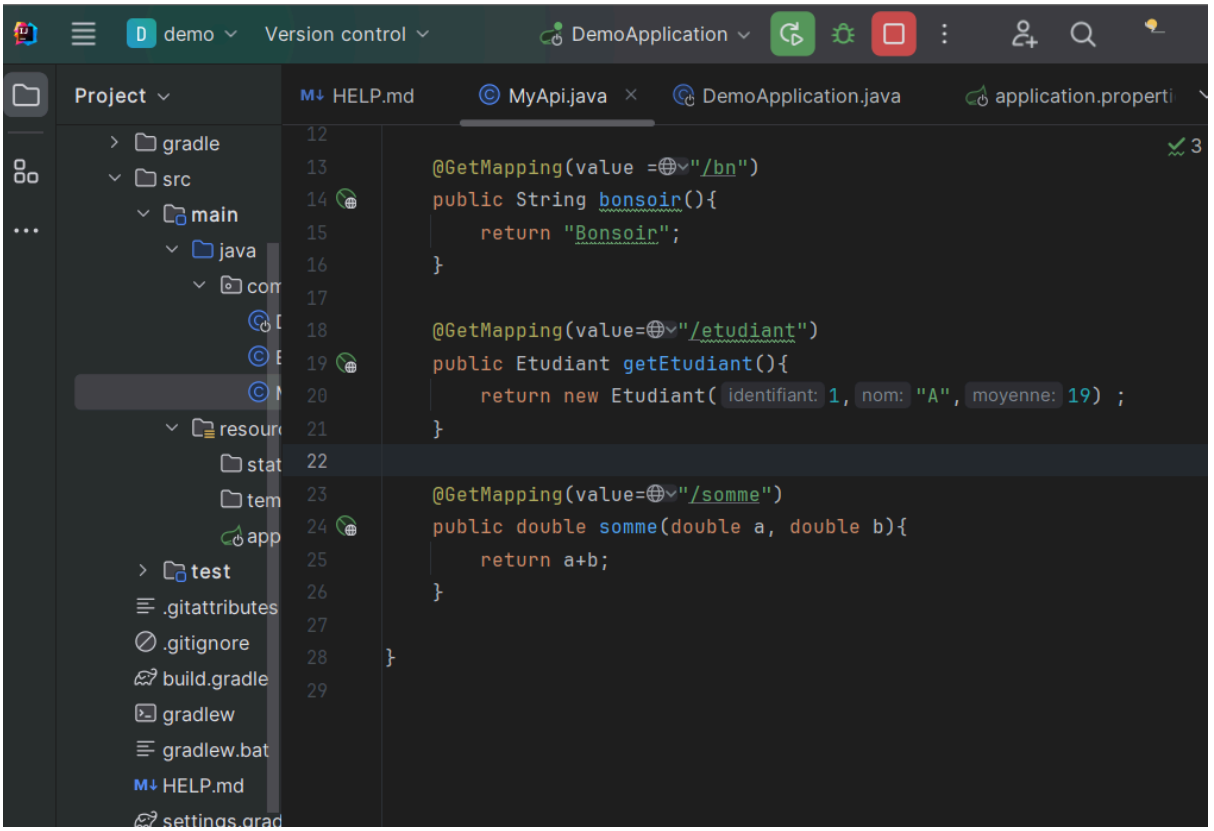
Implémentation des API REST

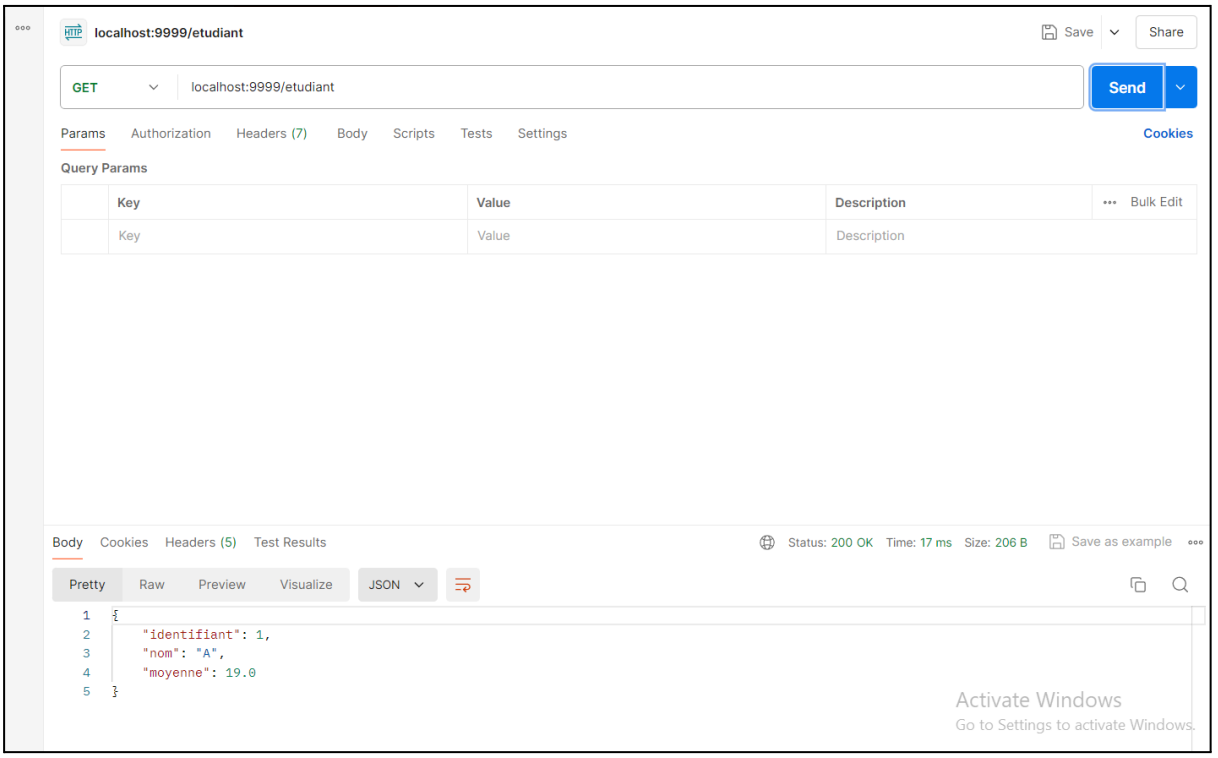
1. **getEtudiant :** Cette méthode permet de **récupérer un étudiant**, en effectuant une recherche dans une liste en mémoire :

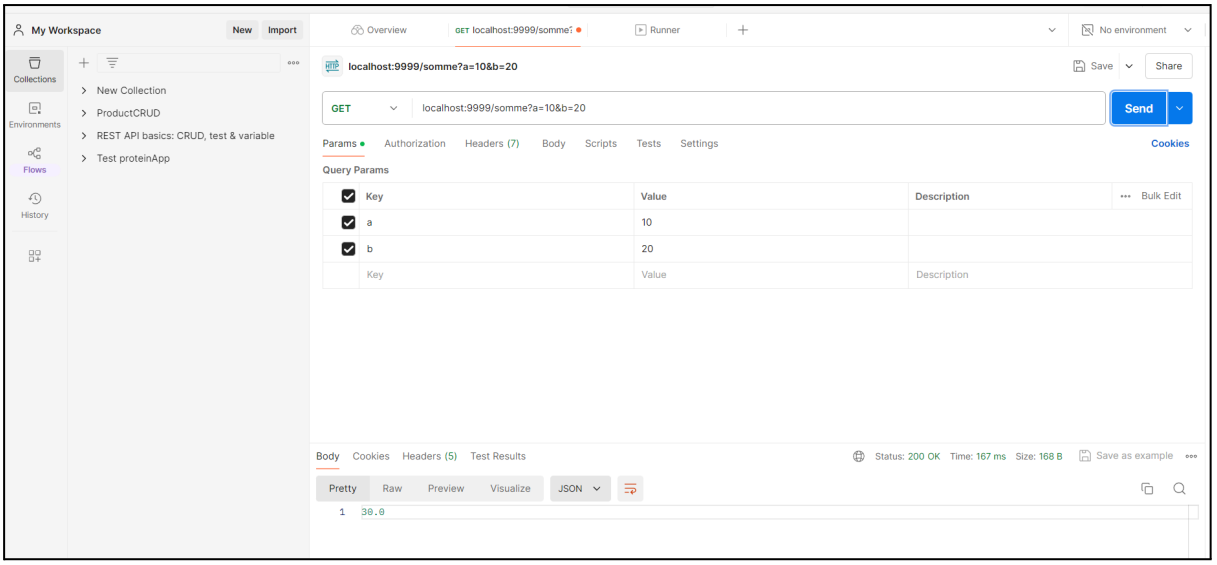
Lorsqu'on appelle l'endpoint via un navigateur, les données de l'objet Etudiant sont automatiquement **affichées au format JSON**.



2. **Méthode personnalisée : somme :** En plus des opérations classiques, on peut ajouter une méthode personnalisée, comme une fonction qui retourne la **somme de deux nombres** via une requête GET.

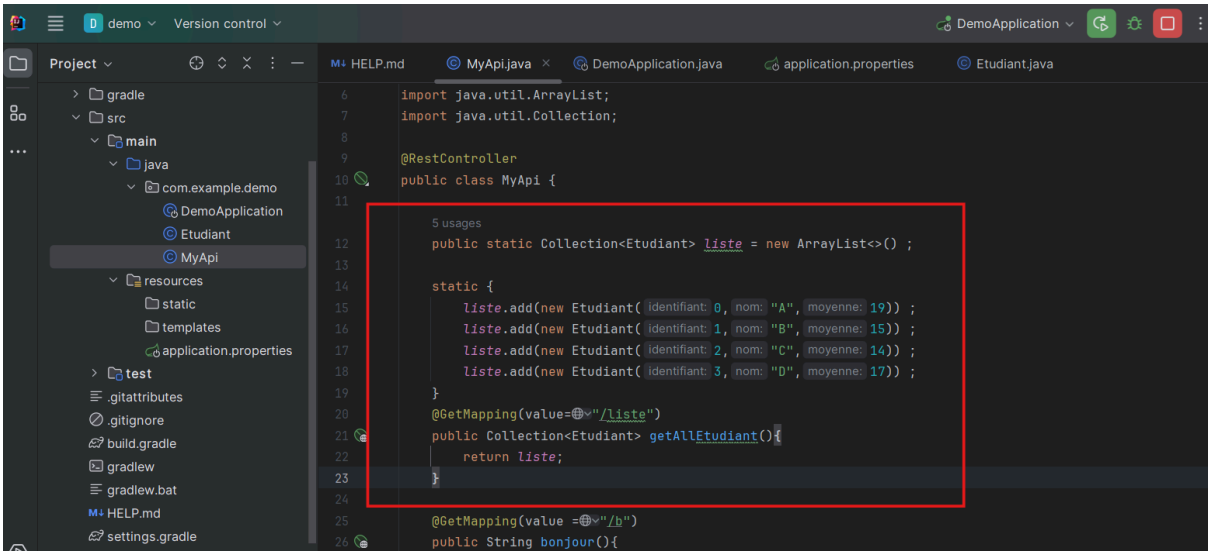






Manipulation de la couche DAO en mémoire

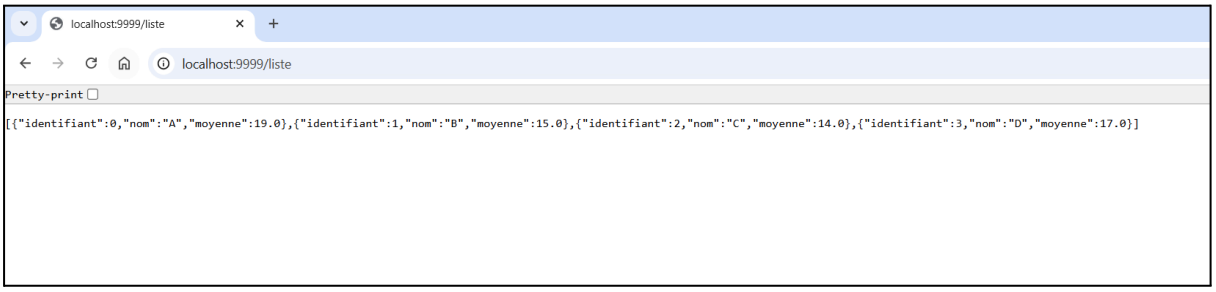
Dans cette phase, nous manipulons la **couche DAO** sans utiliser de base de données. Les étudiants sont stockés dans une **liste statique en mémoire (RAM)**.



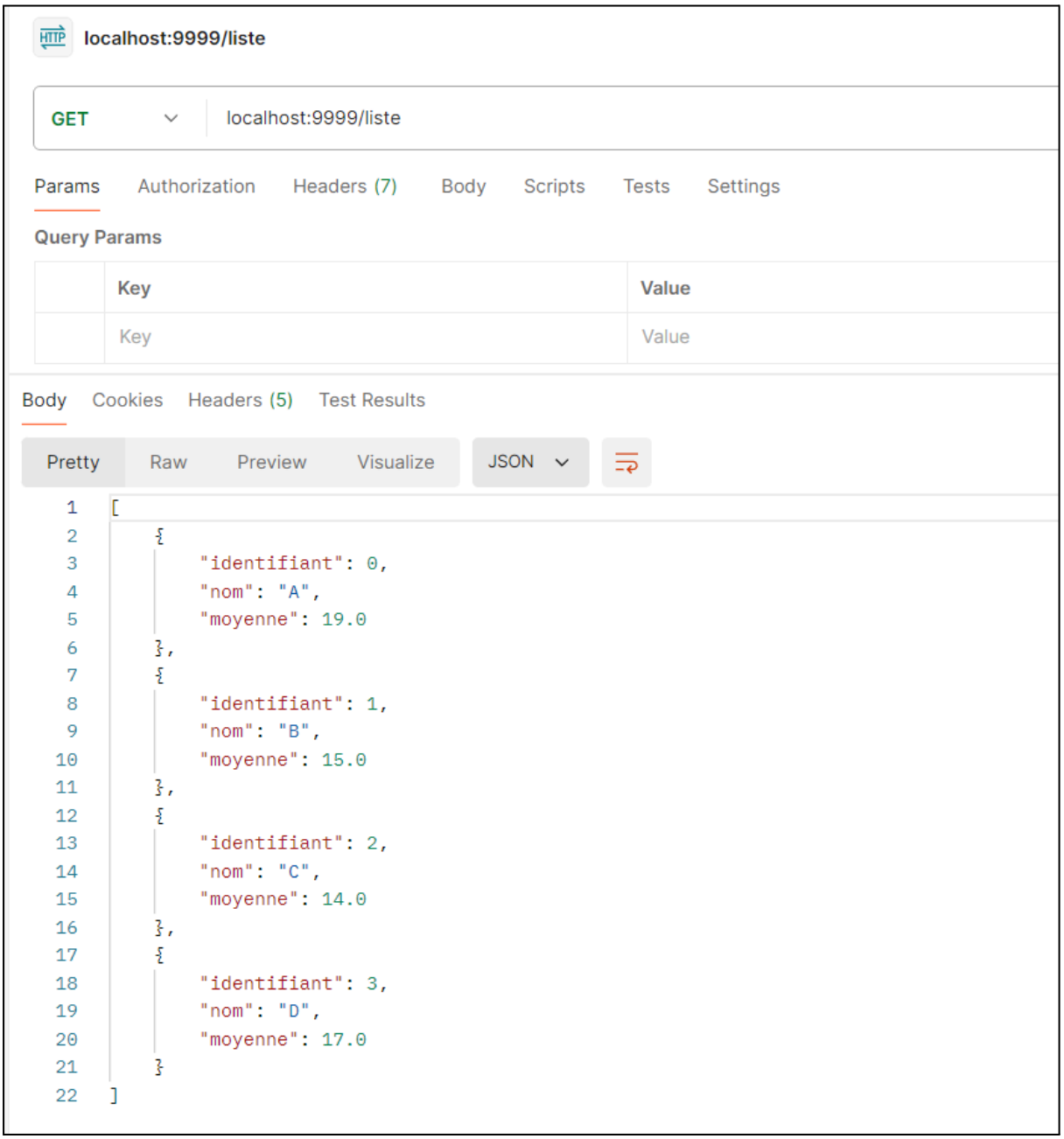
Explication : Pourquoi static ?

La liste est déclarée `static` pour qu'elle soit **partagée entre toutes les instances** du contrôleur, et ainsi accessible globalement pendant toute la durée de l'exécution. Cependant, comme les données ne sont pas persistées, elles sont **perdues à chaque redémarrage** de l'application.

Résultat



Avec Postman :

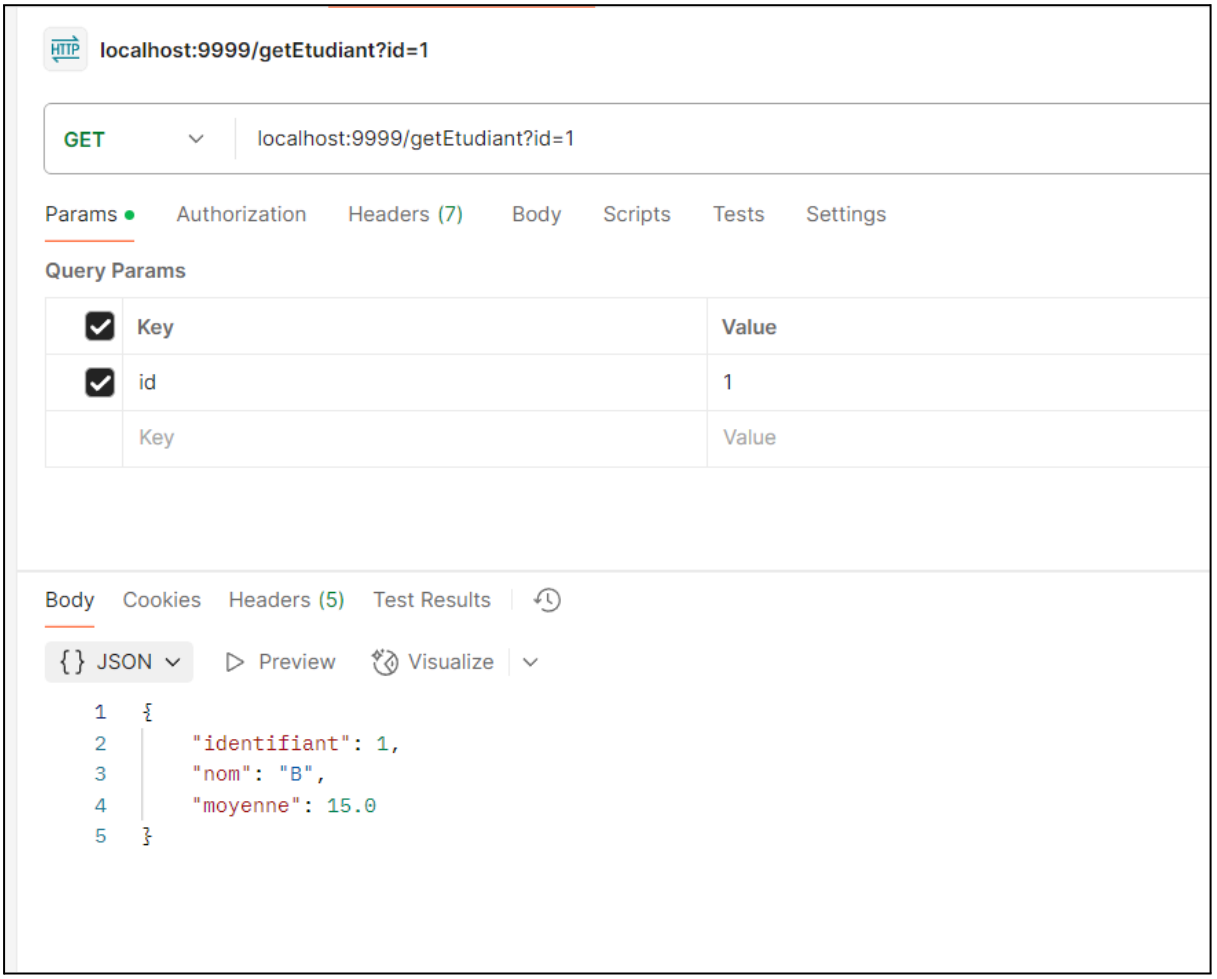


Implémentation des opérations CRUD

Nous mettons ensuite en place les différentes méthodes CRUD, en commençant par :

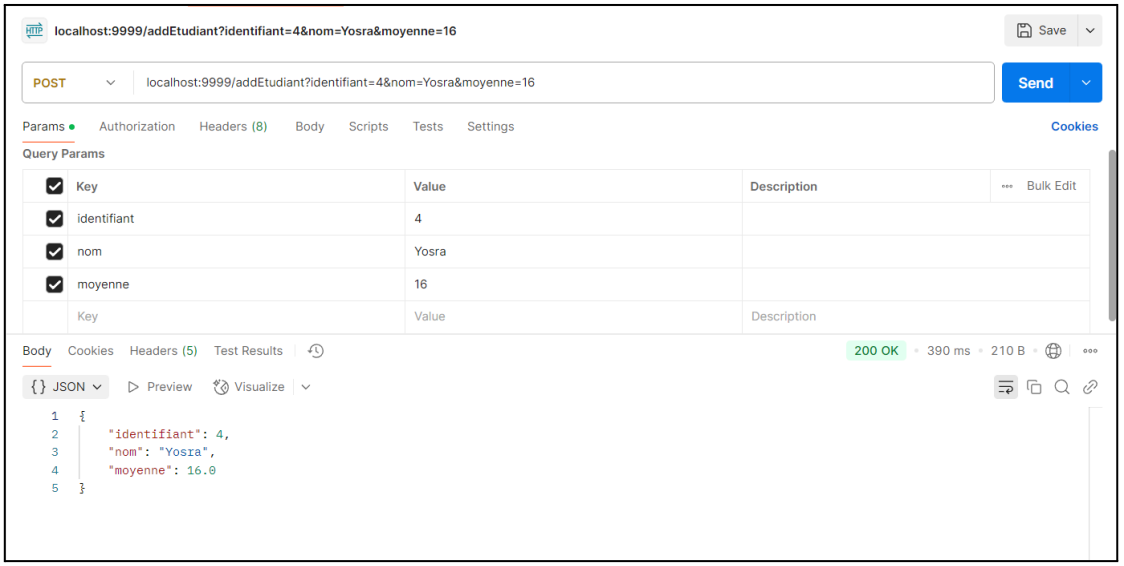
1. **Récupérer un étudiant par identifiant**

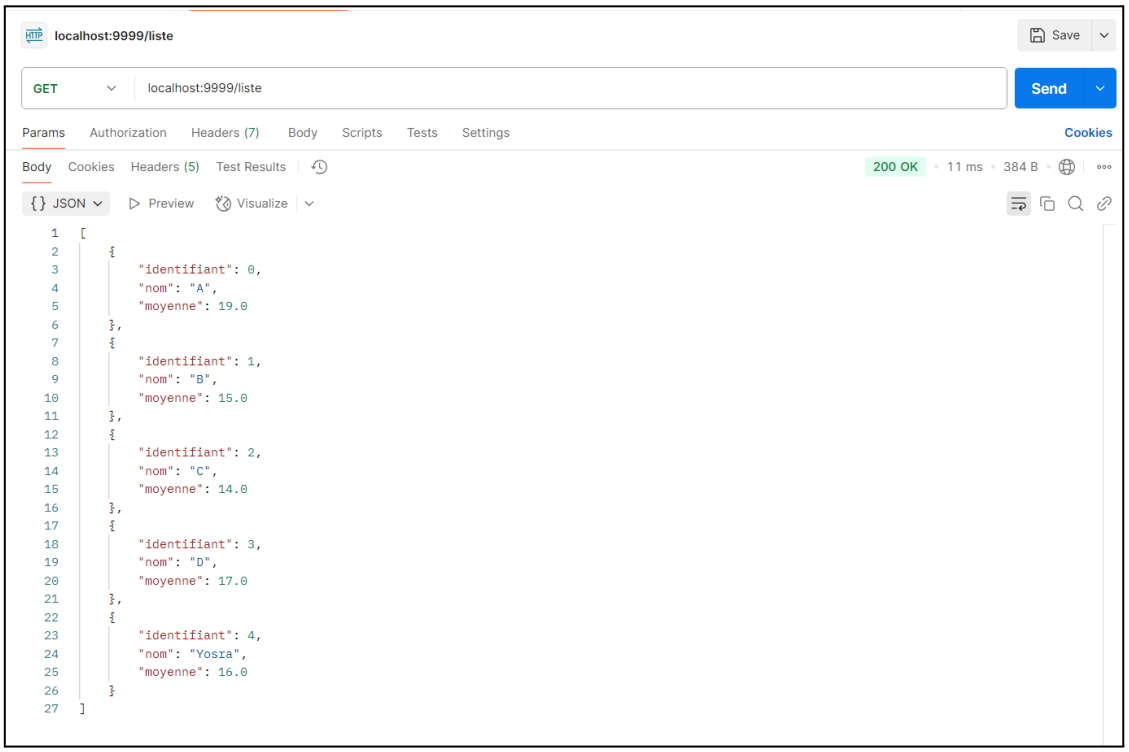
```java
@GetMapping(value="/getEtudiant")
public Etudiant getEtudiant(int identifiant){
    return liste.get(identifiant) ;
}
```



2. **Ajouter un étudiant (POST)**

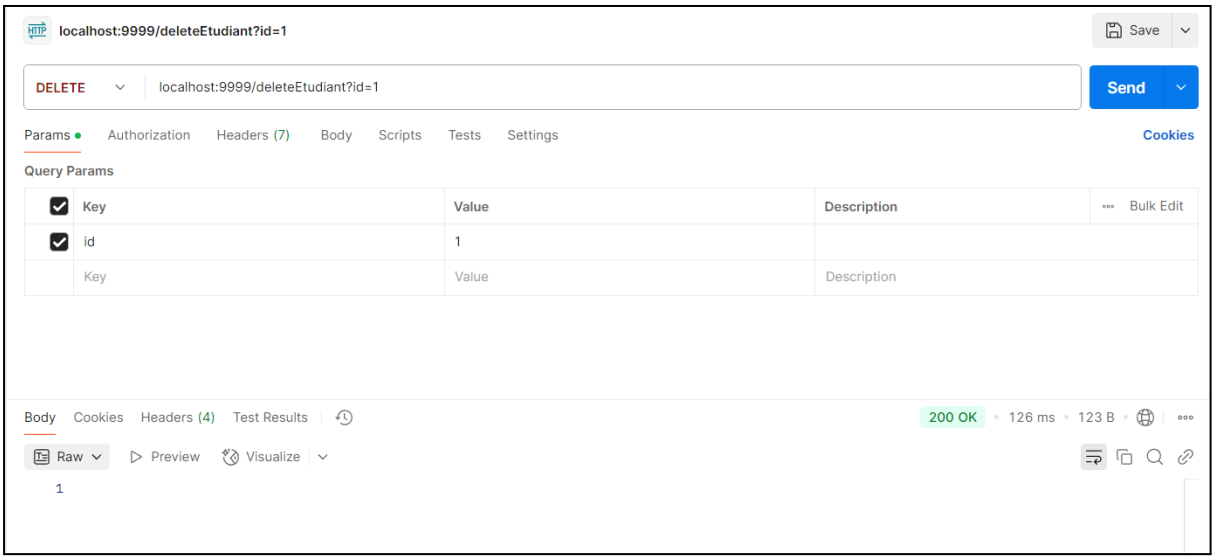
```
27
28      @PostMapping(value = "/addEtudiant")
29      public Etudiant addEtudiant(Etudiant etudiant) {
30          liste.add(etudiant);
31          return etudiant;
32      }
```

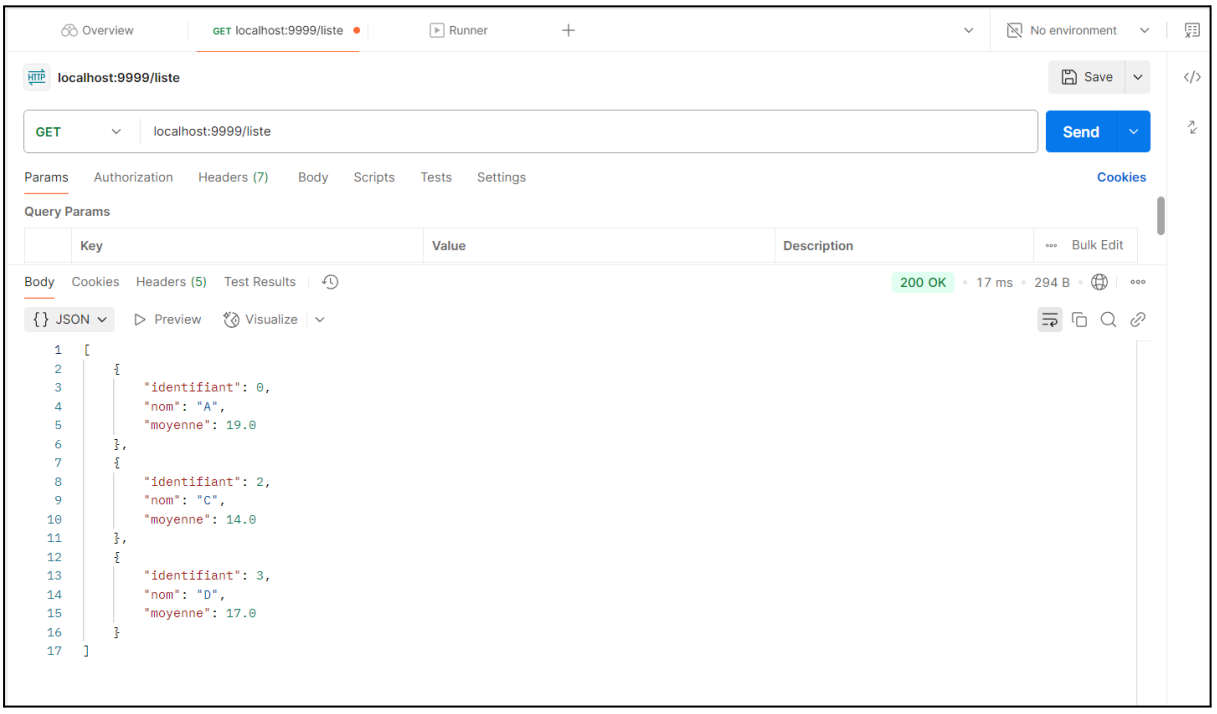




3. **Supprimer un étudiant (DELETE)**

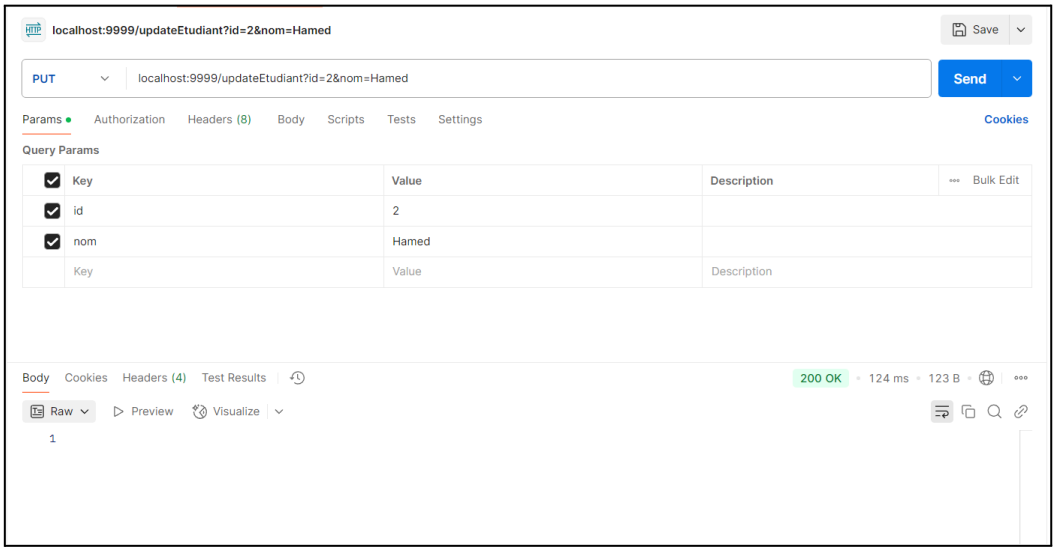
```java
@DeleteMapping(value = "/deleteEtudiant")
public void deleteEtudiant(int id) {
    liste.remove(id);
}
```

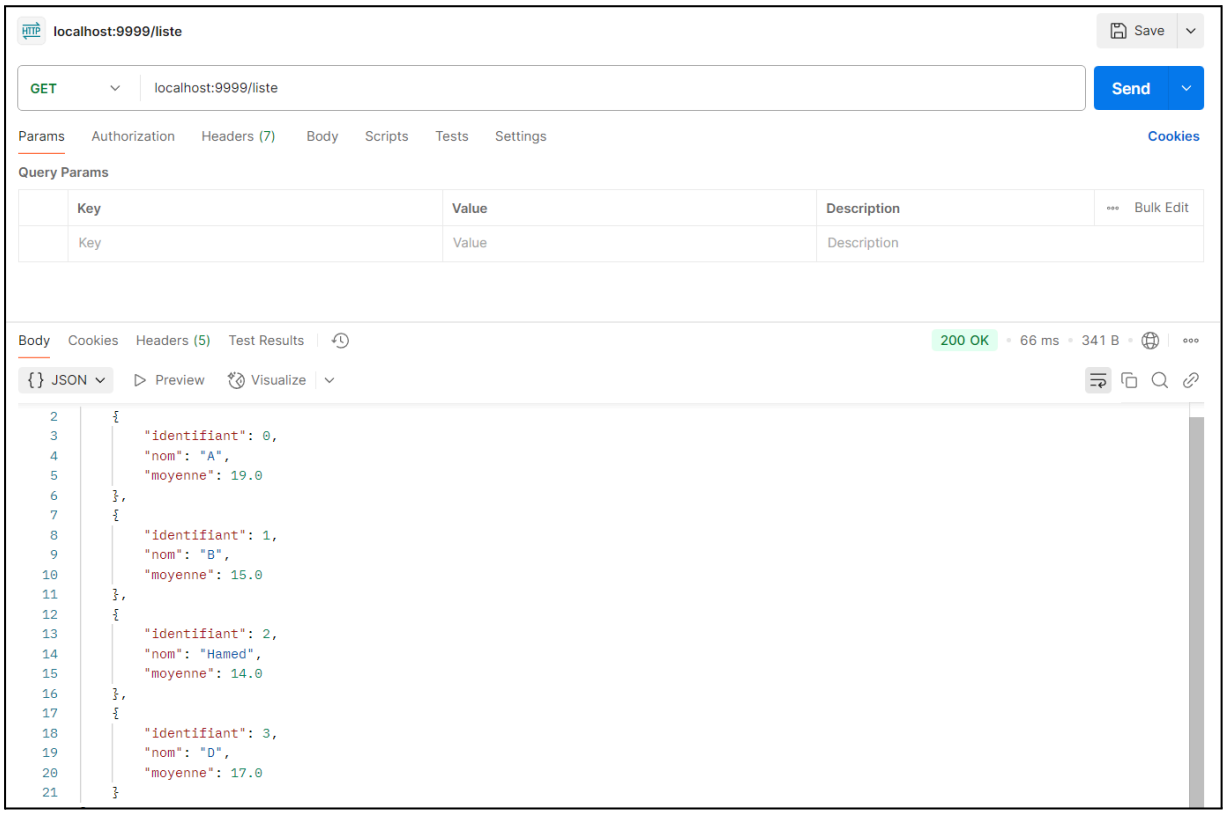




4. **Modifier un étudiant (PUT)**

```
@PutMapping(value = "/updateEtudiant")
public void updateEtudiant(int id , String nom ) {
    liste.get(id).setNom(nom);
}
```



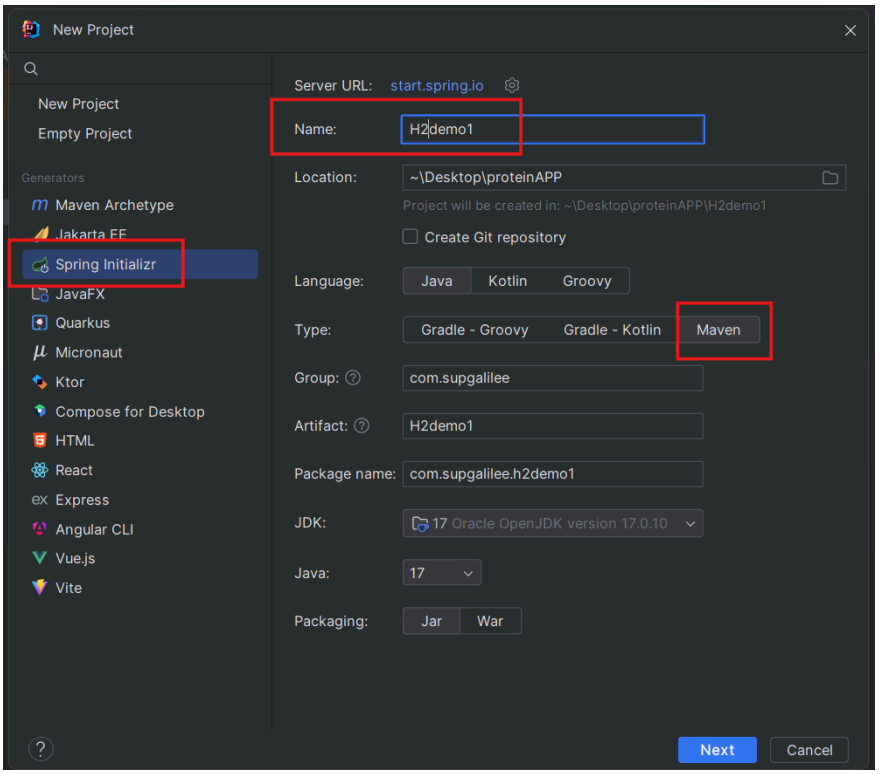


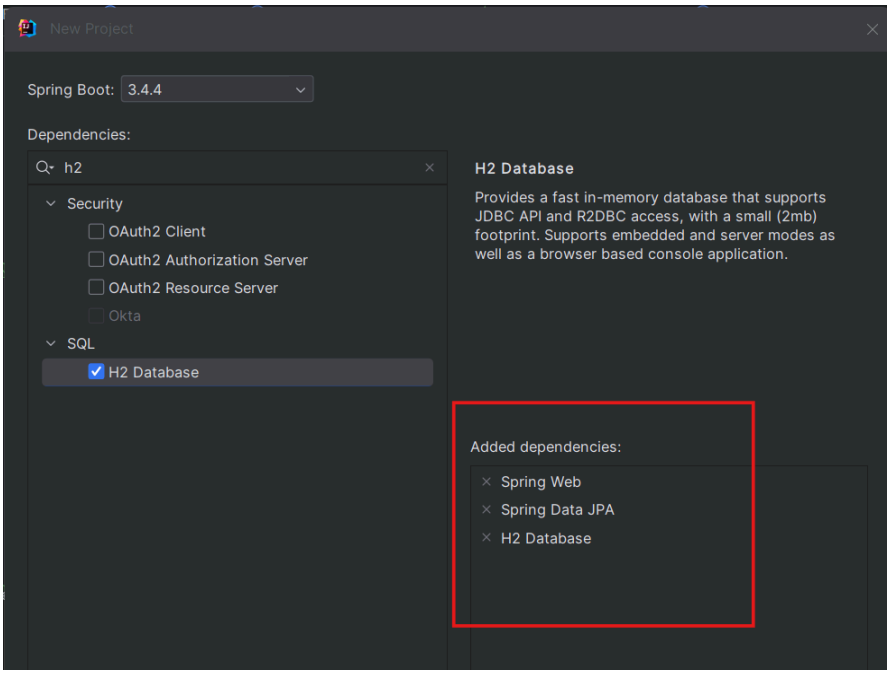
Manipulation de JPA et H2

Après avoir testé la couche DAO en mémoire, nous passons à une gestion réelle des données en utilisant :

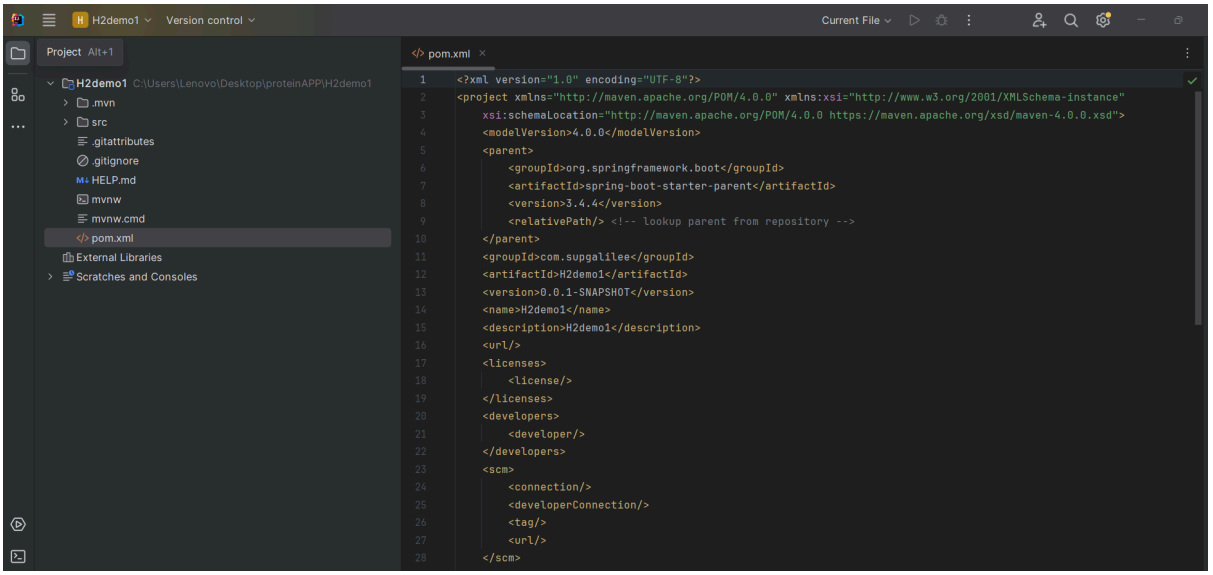
- **JPA (Java Persistence API)** pour la gestion des entités et des opérations en base

- **H2**, une base de données embarquée en mémoire, idéale pour les tests et le développement local

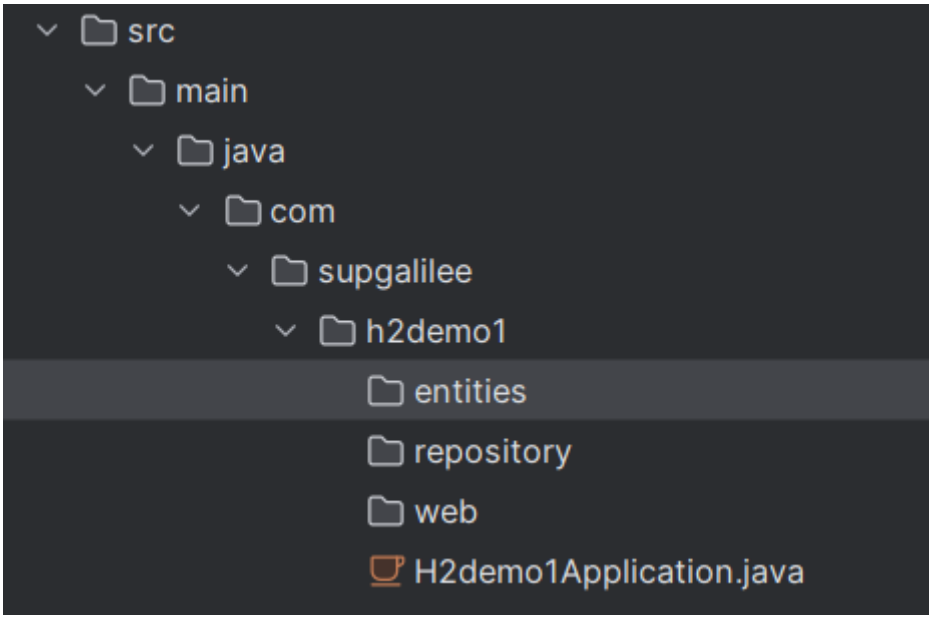




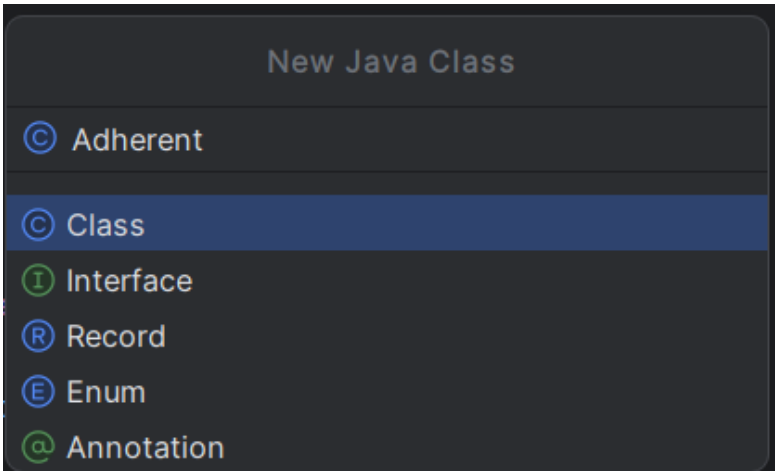
Le fichier pom.xml :

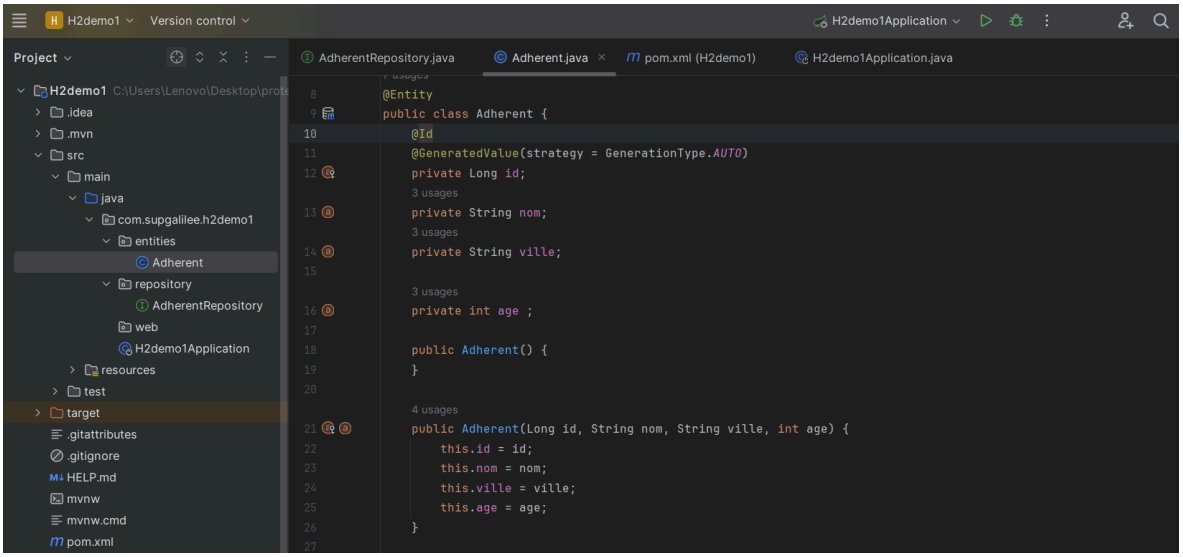


Architecture de projet :



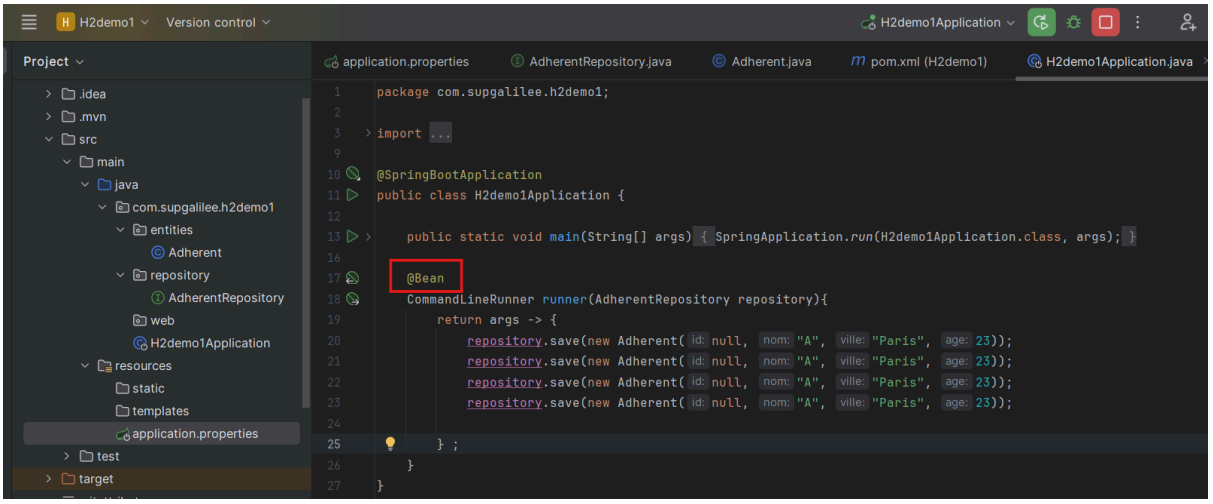
Création d'une entité Adherent



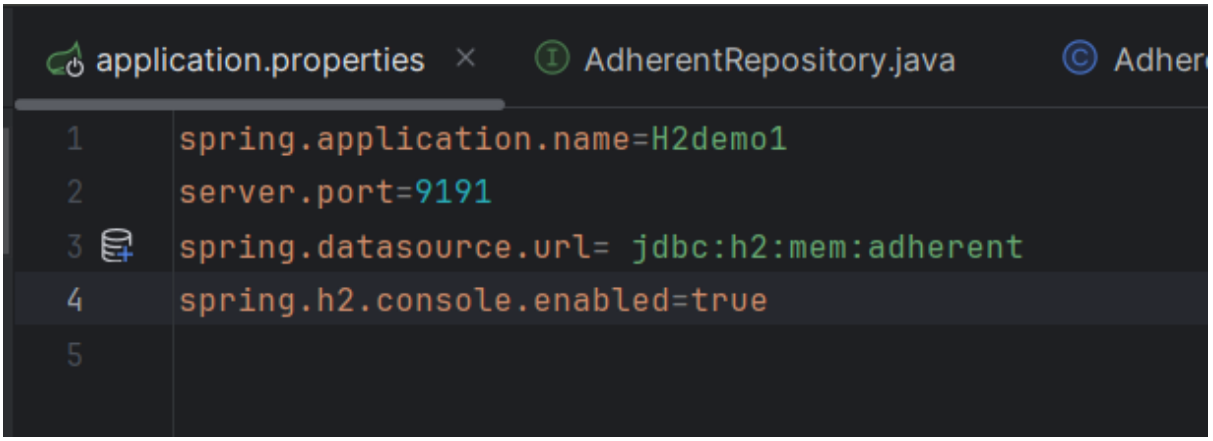


Interface DAO : AdherentRepository

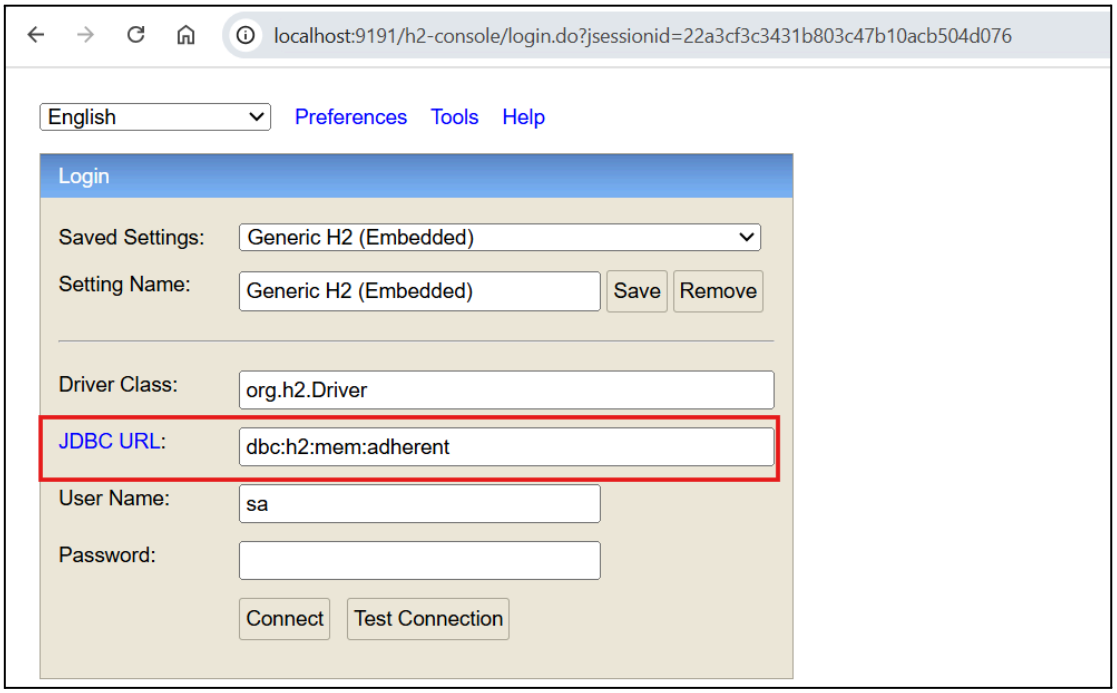
Insertion de données au démarrage

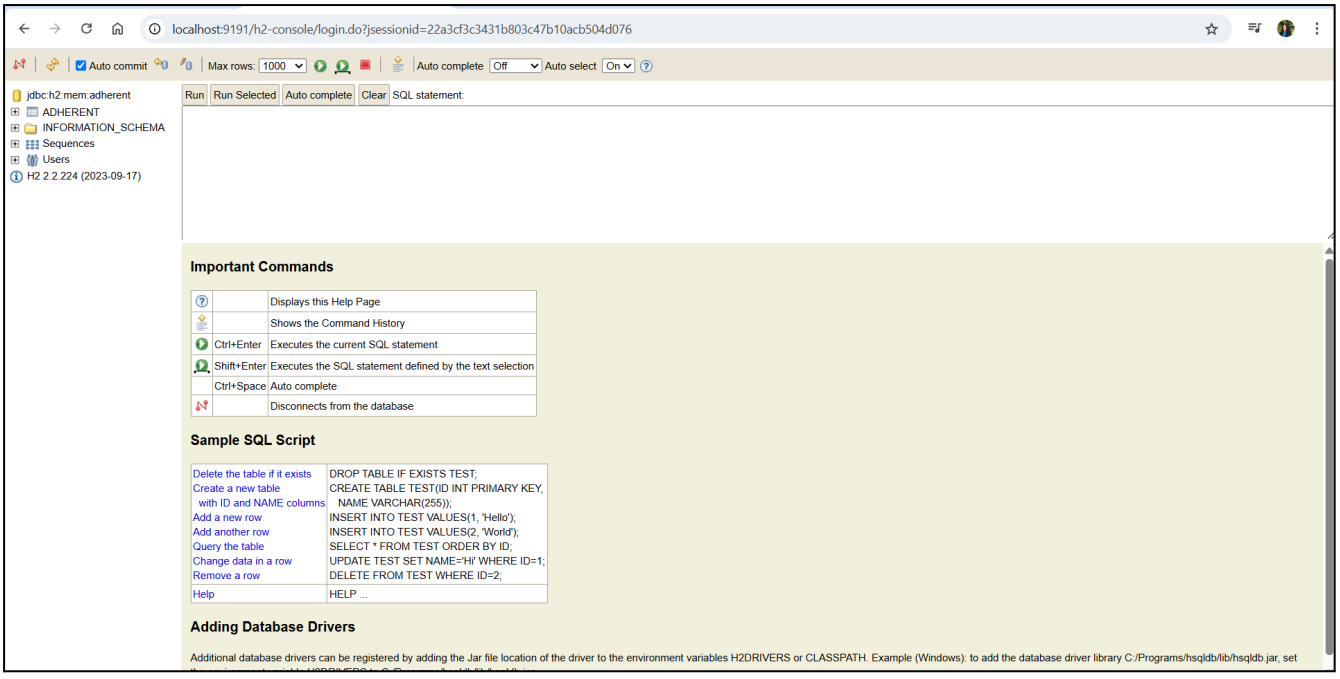


Ce fichier application.properties configure l'application Spring Boot nommée H2demo1 pour qu'elle s'exécute sur le port 9191, utilise une base de données H2 en mémoire nommée adherent, et active la console web H2 pour permettre la visualisation des données à l'adresse http://localhost:9191/h2-console.

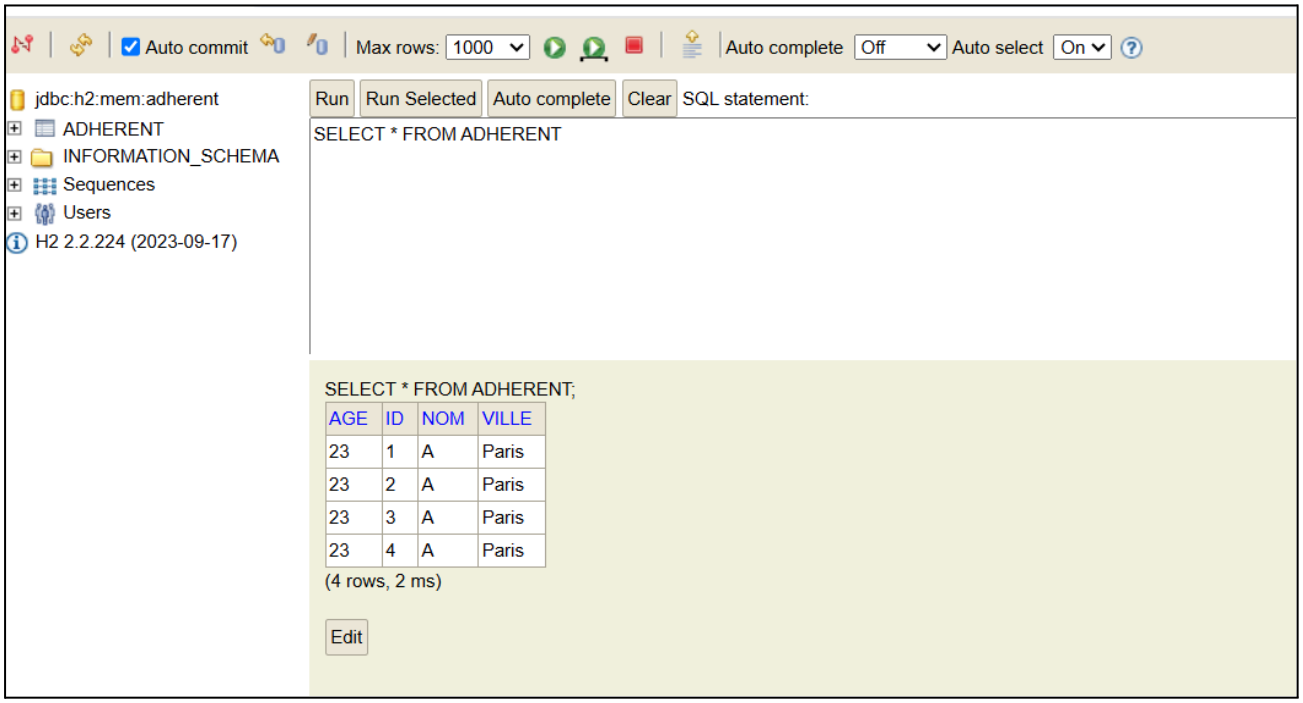


Accès à la console H2





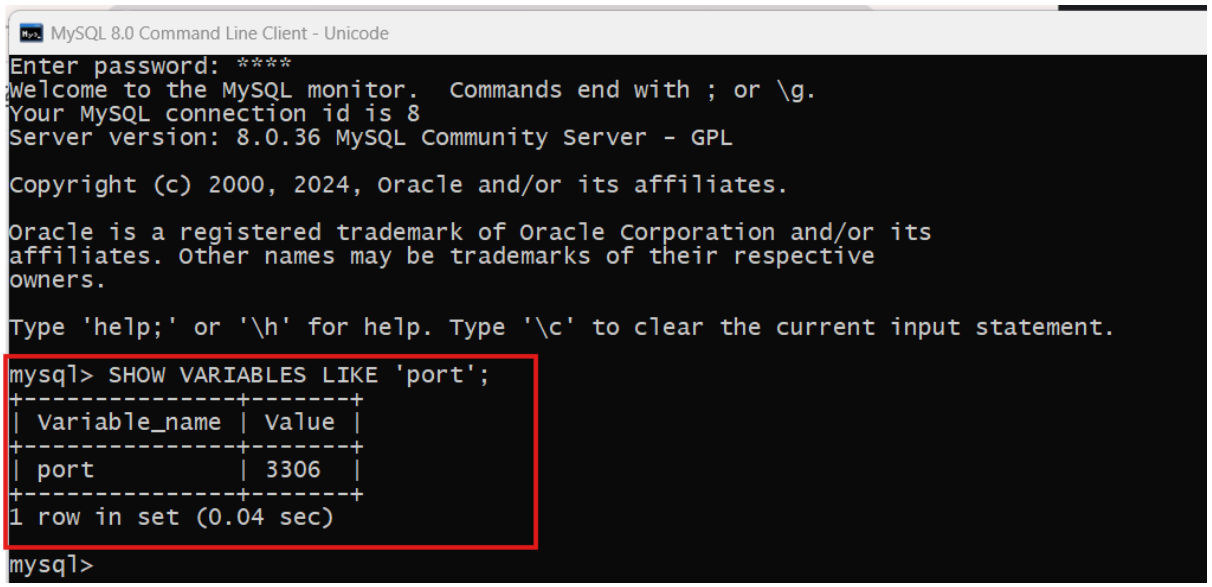
voit le résultat d'une requête SQL **SELECT * FROM ADHERENT**, qui affiche toutes les lignes présentes dans la table **ADHERENT**.



Connexion de Spring Boot avec une base MySQL

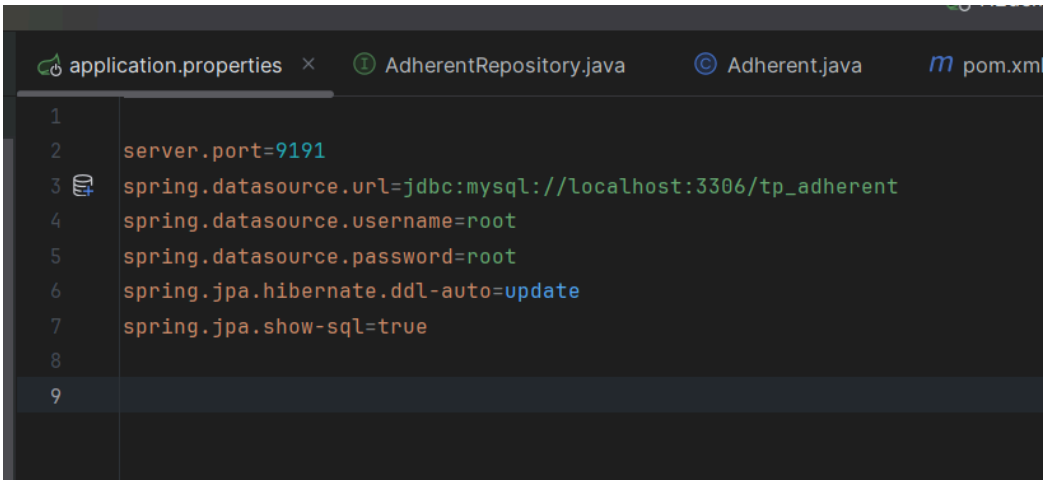
Après avoir manipulé les données en mémoire et avec **H2**, nous avons connecté notre projet Spring Boot à **une base de données MySQL locale**.

Pour vérifier le port de mysql, exécutez cette commande :

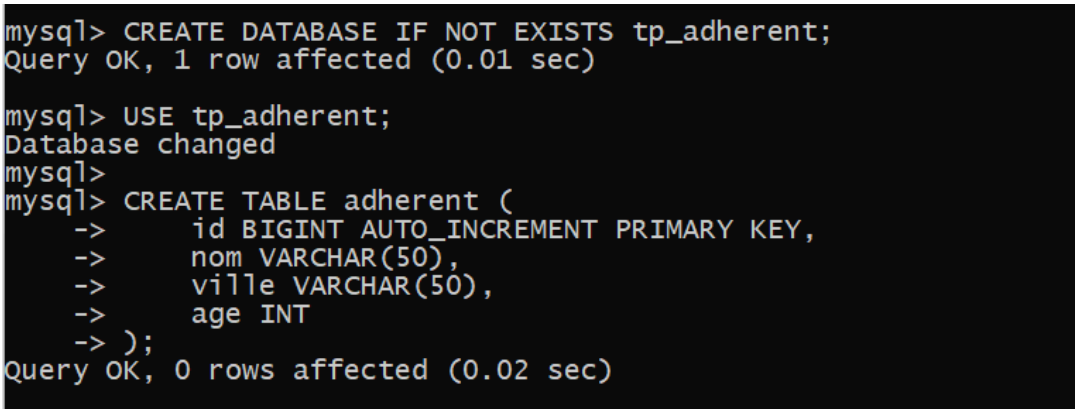


Configuration

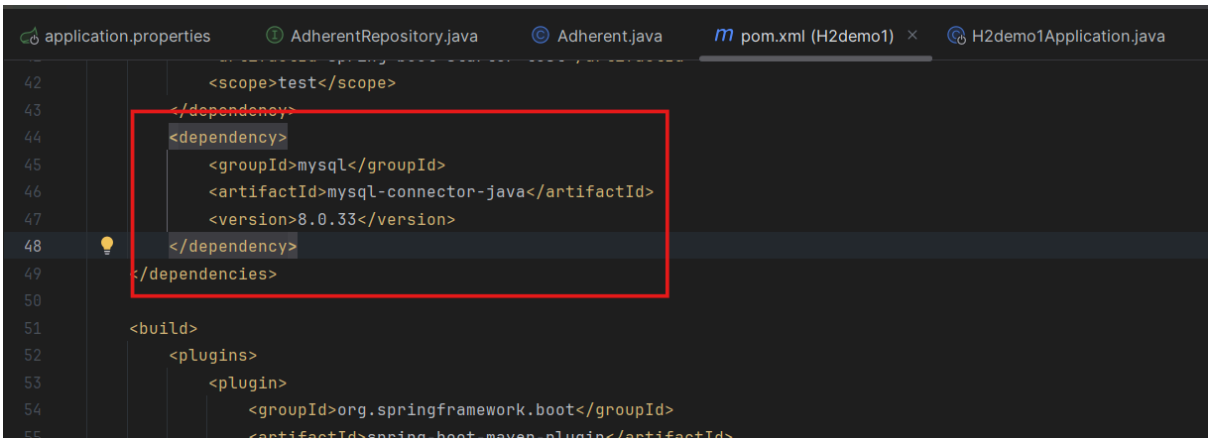
La connexion a été établie via le fichier **application.properties**, en définissant :



→ Cette configuration permet à Spring Boot de se connecter à la base **tp_adherent**, d'afficher les requêtes SQL générées par JPA, et de créer ou mettre à jour automatiquement la table **adherent**.
de l'autre côté, on crée la base et la table dans mysql server :



Et pour connecter **Spring Boot à MySQL**, tu dois ajouter la **dépendance du driver MySQL** dans ton fichier **pom.xml**.



Insertion automatique des données

Grâce à un **CommandLineRunner**, plusieurs objets **Adherent** ont été insérés dès le démarrage de l'application. **Hibernate** a généré les requêtes SQL suivantes :

```
Hibernate: select next_val as id_val from adherent_seq for update
Hibernate: update adherent_seq set next_val= ? where next_val=?
Hibernate: insert into adherent (age,nom,ville,id) values (?,?,?,?)
Hibernate: select next_val as id_val from adherent_seq for update
Hibernate: update adherent_seq set next_val= ? where next_val=?
Hibernate: insert into adherent (age,nom,ville,id) values (?,?,?,?)
Hibernate: insert into adherent (age,nom,ville,id) values (?,?,?,?)
Hibernate: insert into adherent (age,nom,ville,id) values (?,?,?,?)
```

Vérification dans MySQL

Une requête SQL exécutée dans le client MySQL :

```
mysql> SELECT * from adherent;
+----+------+-------+------+
| id | nom  | ville | age  |
+----+------+-------+------+
|  1 | A    | Paris |   23 |
|  2 | A    | Paris |   23 |
|  3 | A    | Paris |   23 |
|  4 | A    | Paris |   23 |
+----+------+-------+------+
4 rows in set (0.00 sec)
```

→ Cette étape démontre que notre application est capable de **communiquer avec une vraie base de données**, et que l'injection des données fonctionne correctement. Cela prépare le terrain pour créer des APIs REST complètes avec persistance réelle.