

## 4.2 Activité pratique : Spring Boot

Cette activité pratique consiste à créer une application Spring Boot permettant de gérer une liste d'étudiants. Les données seront stockées dans une base de données MySQL, et l'application exposera des services web REST pour interagir avec ces données. L'objectif de cette activité est de mettre en pratique les concepts de Spring Boot, Spring Data JPA, et REST.

### 4.2.1 Crédit du Projet Spring Boot

#### Utilisation de Spring Initializr

La première étape consiste à créer un nouveau projet Spring Boot en utilisant Spring Initializr. Spring Initializr est un outil en ligne qui permet de configurer rapidement un projet en sélectionnant les dépendances nécessaires.

1. Accéder à <https://start.spring.io> et configurer le projet avec les paramètres suivants :
  - Project** : Maven Project
  - Language** : Java
  - Spring Boot** : Version stable la plus récente
  - Group** : com.example
  - Artifact** : student-management
  - Packaging** : Jar
  - Java** : 8 ou supérieur
  - Dependencies** : Web, JPA, REST Repository, Devtools, Validation et MySQL Driver

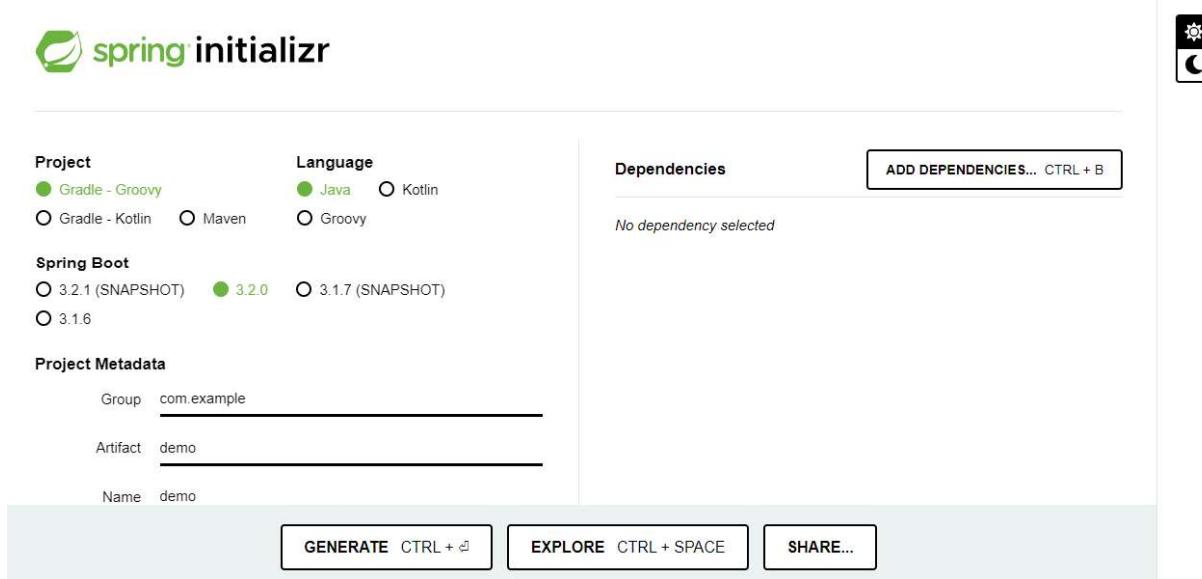


Figure 4.1: Configuration du projet dans Spring Initializr.

2. Télécharger le projet et l'importer dans l'IDE de votre choix (IntelliJ IDEA, Eclipse, etc.).
3. Exécuter la commande suivante pour rendre le projet compatible avec Eclipse :

```
mvn eclipse:eclipse
```

#### Analyse du Fichier pom.xml

Le fichier pom.xml contient toutes les dépendances nécessaires pour exécuter une application Spring Boot. Voici quelques points clés à noter :

- Projet Parent** : Le projet dépend du pom parent de Spring Boot, ce qui permet de bénéficier de configurations par défaut et de versions cohérentes de bibliothèques.

- **Starters** : Les starters incluent les bibliothèques nécessaires pour démarrer rapidement le projet. Par exemple :
  - spring-boot-starter-web : pour les applications Web et RESTful.
  - spring-boot-starter-data-jpa : pour la persistance des données avec JPA.
  - spring-boot-starter-test : pour les tests unitaires et d'intégration.
  - spring-boot-starter-validation : pour la validation des données dans les applications Spring, en utilisant des annotations comme @NotNull, @Size, et @Email.
- **Version de Java** : Assurer que la version de Java utilisée est spécifiée dans la propriété java.version du fichier pom.xml.

#### 4.2.2 Configuration de la Base de Données MySQL

##### Création de la Base de Données

Avant de configurer Spring Boot pour utiliser MySQL, il est nécessaire de créer la base de données MySQL où les données des étudiants seront stockées.

1. Ouvrir un terminal MySQL ou utiliser un outil comme MySQL Workbench.
2. Créer une base de données nommée studentdb :

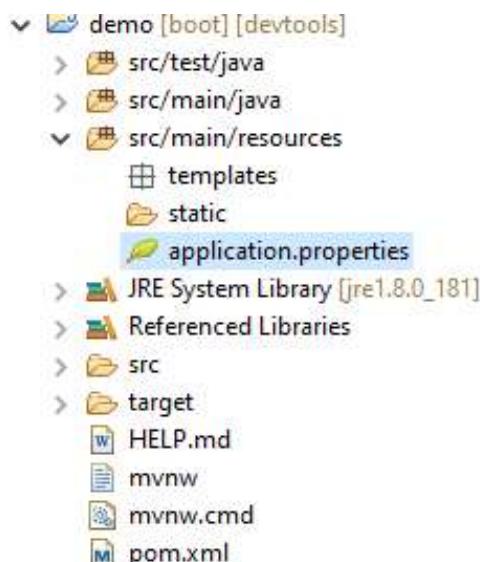
---

```
CREATE DATABASE studentdb;
```

---

##### Configuration dans application.properties

Le fichier application.properties se trouve dans le dossier src/main/resources et contient toutes les configurations nécessaires pour que l'application Spring Boot se connecte à la base de données MySQL.



**Figure 4.2: Fichier application.properties dans le projet Spring Boot.**

3. Configurer les paramètres suivants dans le fichier application.properties :

---

```
# =====
# = DATA SOURCE CONFIGURATION
# =====
spring.datasource.url =
  jdbc:mysql://localhost:3306/studentdb?serverTimezone=UTC
spring.datasource.username = root
spring.datasource.password =
```

---

```
# =====
# = JPA / HIBERNATE CONFIGURATION
# =====
spring.jpa.show-sql = true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.hibernate.ddl-auto = update
```

---

4. La propriété `spring.jpa.hibernate.ddl-auto` est définie sur `update` pour que Hibernate mette à jour automatiquement la base de données en fonction des entités définies dans le projet.

#### 4.2.3 Crédation des Entités et Repositories

##### Définition des Entités

Les entités représentent les objets métier de l'application. Dans cette activité, l'entité `Student` sera créée pour représenter un étudiant.

1. Créer un sous-package `com.example.studentmanagement.entity`.
2. Dans ce package, créer la classe `Student` annotée avec `@Entity` :

---

```
package com.example.demo.entities;
import jakarta.persistence.*;
import java.util.Date;

@Entity
public class Student {
    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    private int id;
    private String nom;
    private String prenom;
    @Temporal(TemporalType.DATE)
    private Date dateNaissance;

    // Getters et Setters
}
```

---

3. Cette classe sera utilisée par JPA pour créer la table correspondante dans la base de données.



**Remarque :** L'annotation `@Entity` indique que cette classe sera utilisée comme entité dans la base de données. L'annotation `@Id` spécifie l'identifiant unique de chaque enregistrement (ici, l'étudiant).

##### Création des Repositories

Les repositories sont des interfaces qui permettent d'accéder aux données stockées dans la base de données. Spring Data JPA fournit une implémentation automatique des méthodes CRUD (*Create, Read, Update, Delete*), évitant ainsi d'avoir à écrire des requêtes SQL manuellement pour ces opérations basiques. Grâce à cette abstraction, les interactions avec la base de données sont considérablement simplifiées, permettant aux développeurs de se concentrer sur la logique métier.



**Remarque :** Spring Data JPA repose sur le principe de *repository pattern* qui sépare la logique de la couche de persistance de la logique métier. Ainsi, les développeurs peuvent facilement interagir avec les entités stockées dans la base de données en utilisant des méthodes prédéfinies, sans avoir à se soucier des détails de l'implémentation de ces opérations.

4. Créer un sous-package com.example.studentmanagement.repository. Ce package contiendra toutes les interfaces qui interagissent directement avec la base de données.
5. Dans ce package, créer l'interface StudentRepository qui étend JpaRepository :

---

```

package com.example.demo.repositories;

import java.util.Collection;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;
import com.example.demo.entities.Student;

@Repository
public interface StudentRepository extends JpaRepository<Student, Integer> {

    // Recherche d'un étudiant par son identifiant
    Student findById(int id);

    // Requête personnalisée pour compter les étudiants par année de naissance
    @Query("SELECT YEAR(s.dateNaissance), COUNT(s) FROM Student s GROUP BY
           YEAR(s.dateNaissance)")
    Collection<Object[]> findNbrStudentByYear();
}

```

---

6. Cette interface hérite de JpaRepository, qui est une interface générique fournie par Spring Data JPA. L'interface JpaRepository offre plusieurs méthodes par défaut telles que save(), findAll(), delete() et findById(), qui sont prêtes à l'emploi et permettent de gérer facilement les opérations CRUD pour l'entité Student.



**Remarque :** L'interface JpaRepository est paramétrée avec deux arguments :

- Le type de l'entité gérée, ici Student.
- Le type de l'identifiant de l'entité, ici Integer.

Cela permet à Spring de générer automatiquement les implémentations des méthodes de l'interface JpaRepository, comme findById().

7. L'annotation @Repository signale à Spring que cette interface est un composant **Repository** qui doit être pris en charge par le framework. Spring gère la création d'une instance de cette interface à l'exécution et l'injecte automatiquement là où elle est nécessaire (par exemple dans un service ou un contrôleur).



**Remarque :** L'annotation @Repository fait également office de mécanisme de gestion des exceptions. Spring convertit automatiquement les exceptions de persistance spécifiques à la base de données (comme SQLException) en exceptions Spring plus génériques, comme DataAccessException.

8. En plus des méthodes CRUD standards, il est possible de définir des requêtes personnalisées à l'aide de l'annotation @Query. Dans l'exemple ci-dessus, la méthode findNbrStudentByYear() utilise une requête JPQL (*Java Persistence Query Language*) pour compter le nombre d'étudiants groupés par leur année de naissance.



**Remarque :** La requête personnalisée @Query("SELECT YEAR(s.dateNaissance), COUNT(s) FROM Student s GROUP BY YEAR(s.dateNaissance)") regroupe les étudiants par année de naissance et retourne le nombre d'étudiants pour chaque année. Ce type de requête est très utile pour générer des rapports ou des statistiques sur les données.

#### 4.2.4 Création des Services et Contrôleurs REST

Les services et contrôleurs REST sont des composants essentiels dans une architecture Spring Boot. Le service contient la logique métier et interagit avec les repositories pour accéder aux données, tandis que le contrôleur REST expose des endpoints HTTP permettant aux clients de communiquer avec l'application.

##### Création de la Classe Service

La classe service est responsable de la gestion des opérations métiers sur les entités. Elle interagit avec le repository pour effectuer des opérations CRUD sur la base de données.

---

```
package com.example.demo.services;

import com.example.demo.entities.Student;
import com.example.demo.repositories.StudentRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Collection;
import java.util.List;
import java.util.Optional;

@Service
public class StudentService {

    @Autowired
    private StudentRepository studentRepository;

    public Student save(Student student) {
        return studentRepository.save(student);
    }

    public boolean delete(int id) {
        Optional<Student> studentOptional =
            Optional.ofNullable(studentRepository.findById(id));
        if (studentOptional.isPresent()) {
            studentRepository.delete(studentOptional.get());
            return true;
        } else {
            return false;
        }
    }

    public List<Student> findAll() {
        return studentRepository.findAll();
    }

    public long countStudents() {
        return studentRepository.count();
    }

    public Collection<?> findNbrStudentByYear() {
        return studentRepository.findNbrStudentByYear();
    }
}
```

---

**Explication du code :**

- **@Service** : Cette annotation indique à Spring que cette classe contient de la logique métier et qu'elle doit être gérée comme un bean Spring. Spring crée automatiquement une instance de cette classe pour l'injecter là où nécessaire.
- **@Autowired** : Cette annotation indique que Spring doit injecter une instance du `StudentRepository` dans la classe `StudentService`. Cela évite d'avoir à instancier manuellement cette dépendance.
- `save(Student student)` : Cette méthode utilise le repository pour enregistrer un étudiant dans la base de données. Si l'étudiant a déjà un `id`, il sera mis à jour. Sinon, un nouvel enregistrement sera créé.
- `delete(int id)` : Cette méthode tente de récupérer un étudiant par son identifiant (via `findById()`). Si l'étudiant est trouvé, il est supprimé de la base de données. Sinon, la méthode renvoie `false`, signalant que l'étudiant n'existe pas.
- `findAll()` : Cette méthode renvoie la liste complète des étudiants présents dans la base de données en appelant `findAll()` sur le repository.
- `countStudents()` : Cette méthode retourne le nombre total d'étudiants en base de données en appelant `count()` sur le repository.
- `findNbrStudentByYear()` : Cette méthode exécute une requête personnalisée définie dans le repository pour retourner le nombre d'étudiants groupés par année de naissance.

**Création du Contrôleur REST**

Le contrôleur REST gère les requêtes HTTP envoyées par le client (comme une interface web ou une application mobile). Il utilise les services pour effectuer des opérations et retourne les résultats au client via des réponses HTTP.

```
package com.example.demo.controllers;

import com.example.demo.entities.Student;
import com.example.demo.services.StudentService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.Collection;
import java.util.List;

@RestController
@RequestMapping("students")
public class StudentController {

    @Autowired
    private StudentService studentService;

    @PostMapping("/save")
    public ResponseEntity<Student> save(@RequestBody Student student) {
        Student savedStudent = studentService.save(student);
        return new ResponseEntity<>(savedStudent, HttpStatus.CREATED);
    }

    @DeleteMapping("/delete/{id}")
    public ResponseEntity<Void> delete(@PathVariable("id") int id) {
        boolean deleted = studentService.delete(id);
        if (deleted) {
            return new ResponseEntity<>(HttpStatus.NO_CONTENT);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }
}
```

```

    }

}

@GetMapping("/all")
public ResponseEntity<List<Student>> findAll() {
    List<Student> students = studentService.findAll();
    return new ResponseEntity<>(students, HttpStatus.OK);
}

@GetMapping("/count")
public ResponseEntity<Long> countStudent() {
    long count = studentService.countStudents();
    return new ResponseEntity<>(count, HttpStatus.OK);
}

@GetMapping("/byYear")
public ResponseEntity<Collection<?>> findByYear() {
    Collection<?> studentsByYear = studentService.findNbrStudentByYear();
    return new ResponseEntity<>(studentsByYear, HttpStatus.OK);
}
}

```

---



#### Explication du code :

- `@RestController` : Cette annotation signale à Spring que cette classe gère des requêtes HTTP. Chaque méthode correspond à un endpoint REST exposé à un client.
- `@RequestMapping("students")` : Cela définit que tous les endpoints de ce contrôleur seront accessibles via une URL qui commence par `/students`. Par exemple, `/students/save` pour ajouter un étudiant.
- `@PostMapping("/save")` : Cette méthode gère les requêtes HTTP de type POST envoyées à `/students/save`. Le corps de la requête contient les données JSON de l'étudiant à sauvegarder, et ces données sont converties en un objet `Student` grâce à l'annotation `@RequestBody`.
- `ResponseEntity<Student>` : Cette classe permet de construire une réponse HTTP avec un corps et un statut. Ici, après avoir sauvégarde l'étudiant, la méthode renvoie l'objet `Student` enregistré ainsi qu'un code de statut HTTP 201 (CREATED).
- `@DeleteMapping("/delete/{id}")` : Cette méthode gère les requêtes DELETE envoyées à `/students/delete/{id}`. Elle appelle le service pour supprimer l'étudiant correspondant à l'identifiant fourni dans l'URL. Si l'étudiant est supprimé avec succès, un statut HTTP 204 (NO CONTENT) est renvoyé, sinon un statut 404 (NOT FOUND).
- `@GetMapping("/all")` : Cette méthode gère les requêtes GET envoyées à `/students/all`. Elle renvoie une liste de tous les étudiants dans une réponse HTTP avec le statut 200 (OK).
- `@GetMapping("/count")` : Cette méthode renvoie le nombre total d'étudiants via une réponse HTTP avec le statut 200 (OK).
- `@GetMapping("/byYear")` : Cette méthode renvoie le nombre d'étudiants groupés par année de naissance. Le résultat est encapsulé dans une réponse HTTP avec un statut 200 (OK).

#### 4.2.5 Exécution de l'Application

Pour s'assurer que l'application Spring Boot fonctionne correctement, il est important de suivre quelques étapes clés pour démarrer l'application et tester les endpoints REST. L'IDE (comme IntelliJ IDEA ou Eclipse) fournit des outils qui facilitent cette démarche.

##### 1. Localisation de la classe principale :

- La classe principale de l'application est généralement nommée `DemoApplication.java` ou `StudentManagementApplication.java`.

- Cette classe contient la méthode `main()`, qui est le point d'entrée de l'application Spring Boot.
  - L'annotation `@SpringBootApplication` permet à Spring Boot de configurer automatiquement les beans nécessaires et de démarrer l'application.

## **2. Exécution de l'application :**

- Dans l'IDE, localiser la classe principale, puis cliquer sur le bouton *Run* ou utiliser le raccourci clavier pour lancer l'application.
  - Alternativement, ouvrir un terminal à la racine du projet et exécuter la commande suivante:

```
mvn spring-boot:run
```

- Cette commande utilise Maven pour compiler et exécuter l'application Spring Boot.

### **3. Vérification du démarrage correct de l'application :**

- Une fois l’application démarrée, le terminal ou la console de l’IDE affichera des logs. Ces logs indiquent si l’application Spring Boot a démarré correctement.
  - Chercher un message du type Started DemoApplication in X seconds qui confirme que l’application est prête à accepter des requêtes.

**Figure 4.3:** Démarrage de l’application Spring Boot avec succès.



**Remarque :** Si des erreurs apparaissent dans les logs, comme des erreurs de connexion à la base de données MySQL ou des exceptions de configuration, il est important de les corriger avant de poursuivre avec les tests.

# Test des Endpoints REST

Pour vérifier que l'application fonctionne correctement et que les endpoints REST sont bien exposés, nous allons utiliser un client HTTP comme **Postman** ou **Advanced Rest Client**.

## Utilisation de Advanced Rest Client pour Tester les Endpoints

Advanced Rest Client est un outil pratique pour tester les API REST. Il permet d'envoyer des requêtes HTTP (GET, POST, DELETE, etc.) et de visualiser les réponses renvoyées par le serveur. Suivons les étapes pour tester les endpoints exposés par notre application Spring Boot.

#### **1. Ajout d'un Étudiant (POST) :**

- Ouvrir Advanced Rest Client et configurer une requête POST à l'URL `http://localhost:8080/students/save`.
  - Dans l'onglet **Body**, choisir le type `raw` et sélectionner `JSON` pour indiquer que le corps de la requête est un objet JSON.
  - Envoyer les informations de l'étudiant sous la forme suivante :

```
{ "nom": "TACHGAR"
```

```

    "prenom": "Mohamed",
    "dateNaissance": "1985-09-01"
}

```

- Appuyer sur le bouton *Send* pour envoyer la requête au serveur. Si l'étudiant est ajouté avec succès, Advanced Rest Client renverra un code de réponse HTTP 201 Created, avec les détails de l'étudiant ajouté.

The screenshot shows the Advanced Rest Client interface. At the top, the method is set to POST and the URL is http://localhost:8080/students/save. Below this, under 'Parameters', there is a 'Headers' tab with 'Body content type' set to application/json. Under the 'Body' tab, the JSON payload is defined:

```

{
  "nom": "LACHGAR",
  "prenom": "Mohamed",
  "dateNaissance": "1985-09-01"
}

```

At the bottom of the interface, the status bar shows '200 OK' and '430 ms'. To the right, there is a 'DETAILS' button.

**Figure 4.4: Test de l'ajout d'un étudiant avec Advanced Rest Client.**



**Remarque :** Si l'application ne fonctionne pas correctement, un code d'erreur HTTP, tel que 500 (Internal Server Error), peut être renvoyé. Dans ce cas, il est important de vérifier les logs pour identifier la cause de l'erreur.

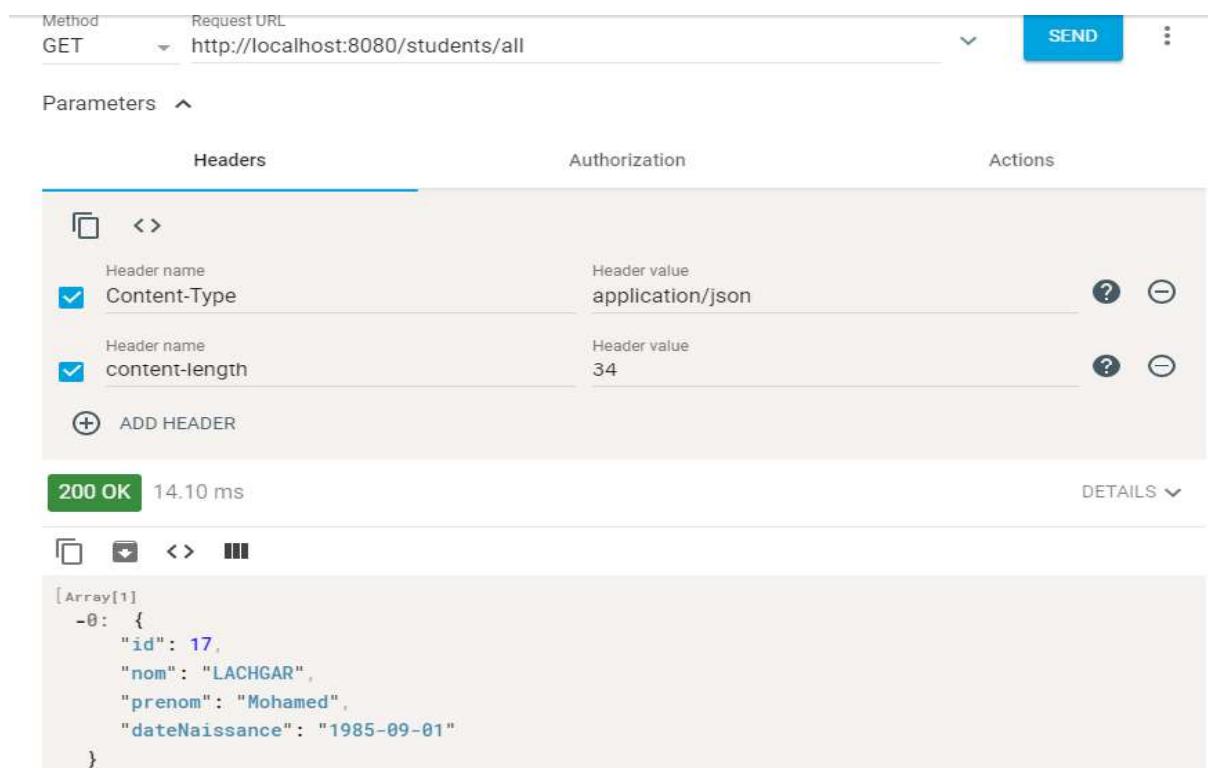
## 2. Récupération de la Liste des Étudiants (GET) :

- Envoyer une requête GET à l'URL <http://localhost:8080/students/all>. Cette requête demande au serveur de renvoyer la liste de tous les étudiants enregistrés dans la base de données.
- La réponse devrait contenir un tableau JSON avec les détails de chaque étudiant, par exemple :

```

[
  {
    "id": 1,
    "nom": "LACHGAR",
    "prenom": "Mohamed",
    "dateNaissance": "1985-09-01"
  }
]

```



**Figure 4.5: Récupération de la liste des étudiants via Advanced Rest Client.**



**Remarque :** La requête GET à l'URL /students/all ne modifie pas la base de données. Elle ne fait que lire les données et les renvoyer sous forme de réponse HTTP.

### 3. Suppression d'un Étudiant (DELETE) :

- Envoyer une requête DELETE à l'URL `http://localhost:8080/students/delete/\{id\}`, en remplaçant `\{id\}` par l'identifiant de l'étudiant à supprimer. Par exemple, `/students/delete/1` supprimera l'étudiant avec l'identifiant 1.
- Si la suppression est réussie, le serveur renverra un code de statut HTTP 204 No Content.



Chaque endpoint correspond à une opération spécifique (POST pour créer, GET pour lire, DELETE pour supprimer). L'utilisation de Advanced Rest Client permet de tester ces opérations de manière simple et rapide.

#### 4.2.6 Tests Unitaires avec JUnit et Mockito

Les tests unitaires jouent un rôle crucial dans la validation du comportement des composants d'une application. Dans cette section, l'objectif est d'illustrer comment tester un contrôleur Spring Boot en utilisant deux outils majeurs : JUnit 5 pour l'exécution des tests et Mockito pour simuler (*mock*) les interactions avec les services. Ces tests permettent de vérifier le bon fonctionnement du contrôleur StudentController de manière isolée, sans dépendre de la base de données ou d'autres composants externes.

##### Présentation des Outils Utilisés

- JUnit 5 :** Un framework de test populaire qui permet de définir, organiser et exécuter des tests unitaires en Java. Il facilite l'écriture d'assertions, c'est-à-dire des vérifications de l'exactitude des résultats d'une méthode.

- **Mockito** : Un framework de moquage qui permet de simuler le comportement des objets dépendants. Dans le cadre des tests de contrôleurs, il permet de créer des objets simulés pour les services, sans avoir à les implémenter réellement.

Ces outils sont essentiels pour écrire des tests rapides, indépendants des bases de données ou des API externes, garantissant ainsi que le code peut être vérifié sans surcharge excessive.

### Code des Tests Unitaires

Voici un exemple de test unitaire pour le contrôleur StudentController utilisant JUnit 5 et Mockito. Ce test simule les comportements du service StudentService et vérifie les réponses du contrôleur pour les méthodes save, delete, findAll, count, et findByYear.

```
package com.example.demo;

import com.example.demo.controllers.StudentController;
import com.example.demo.entities.Student;
import com.example.demo.services.StudentService;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.Arrays;
import java.util.Collection;
import java.util.List;
import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.when;

@SpringBootTest
class StudentControllerTest {

    @Mock
    private StudentService studentService;

    @InjectMocks
    private StudentController studentController;

    @Test
    void testSaveStudent() {
        // Créer un étudiant pour le test
        Student student = new Student();
        student.setId(1);
        student.setNom("Mido");

        // Simuler la sauvegarde
        when(studentService.save(any(Student.class))).thenReturn(student);

        // Appeler le contrôleur pour sauvegarder l'étudiant
        ResponseEntity<Student> response = studentController.save(student);
    }
}
```

```
// Vérifier le statut de la réponse et l'étudiant sauvegardé
assertEquals(HttpStatus.CREATED, response.getStatusCode());
assertEquals("Mido", response.getBody().getNom());
}

@Test
void testDeleteStudent() {
    // Simuler la suppression d'un étudiant
    when(studentService.delete(1)).thenReturn(true);

    // Appeler le contrôleur pour supprimer l'étudiant
    ResponseEntity<Void> response = studentController.delete(1);

    // Vérifier le statut de la réponse
    assertEquals(HttpStatus.NO_CONTENT, response.getStatusCode());
}

@Test
void testfindAllStudents() {
    // Créer des étudiants fictifs
    Student student1 = new Student();
    Student student2 = new Student();

    // Simuler la récupération des étudiants
    when(studentService.findAll()).thenReturn(Arrays.asList(student1, student2));

    // Appeler le contrôleur pour récupérer tous les étudiants
    ResponseEntity<List<Student>> response = studentController.findAll();

    // Vérifier que la liste renvoyée contient bien les étudiants
    assertEquals(2, response.getBody().size());
    assertEquals(HttpStatus.OK, response.getStatusCode());
}

@Test
void testCountStudents() {
    // Simuler le comptage des étudiants
    when(studentService.countStudents()).thenReturn(10L);

    // Appeler le contrôleur pour compter les étudiants
    ResponseEntity<Long> response = studentController.countStudent();

    // Vérifier que le nombre renvoyé est correct
    assertEquals(10L, response.getBody());
    assertEquals(HttpStatus.OK, response.getStatusCode());
}

@Test
void testFindByYear() {
    // Simuler la récupération par année
    when(studentService.findNbrStudentByYear()).thenReturn(Arrays.asList());

    // Appeler le contrôleur pour récupérer le nombre d'étudiants par année
    ResponseEntity<Collection<?>> response = studentController.findByYear();

    // Vérifier que la collection renvoyée est vide
}
```

```

        assertEquals(0, response.getBody().size());
        assertEquals(HttpStatus.OK, response.getStatusCode());
    }
}

```

---

### Explication des Tests

Ce code contient cinq tests unitaires pour les méthodes principales du contrôleur `StudentController`. Chaque test est structuré de manière à simuler les interactions avec le service `StudentService` et à vérifier le comportement attendu du contrôleur.

1. **Annotation `@SpringBootTest`** : Cette annotation permet de charger le contexte Spring Boot pour les tests. Cela inclut tous les composants nécessaires comme les contrôleurs et services.
2. **Annotation `@Mock`** : L'annotation `@Mock` est utilisée pour simuler (*mock*) le service `StudentService`. Grâce à Mockito, des objets simulés sont créés pour remplacer les vrais objets sans toucher aux bases de données ni aux appels réels aux services.
3. **Annotation `@InjectMocks`** : Injecte les objets simulés dans le contrôleur `StudentController`. Cela permet de tester le contrôleur tout en utilisant des dépendances simulées.
4. **Test `testSaveStudent()`** :
  - Un objet `Student` est créé pour le test avec l'ID et le nom "Mido".
  - Le comportement de la méthode `studentService.save()` est simulé pour renvoyer cet objet.
  - Le test vérifie que le code HTTP renvoyé est `CREATED` et que le nom de l'étudiant enregistré est bien "Mido".
5. **Test `testDeleteStudent()`** :
  - Simule la suppression d'un étudiant et vérifie que le contrôleur renvoie le code HTTP `NO_CONTENT` (suppression réussie).
6. **Test `testFindAllStudents()`** :
  - Crée deux étudiants fictifs, simule leur récupération par le service, et vérifie que le contrôleur renvoie bien ces deux étudiants avec un statut `OK`.
7. **Test `testCountStudents()`** :
  - Simule le retour de `10L` (nombre d'étudiants) par le service et vérifie que ce nombre est correctement renvoyé par le contrôleur avec un statut `OK`.
8. **Test `testFindByYear()`** :
  - Simule une collection vide d'étudiants par année et vérifie que le contrôleur renvoie une collection vide avec un statut `OK`.

### Outils et Stratégies de Test

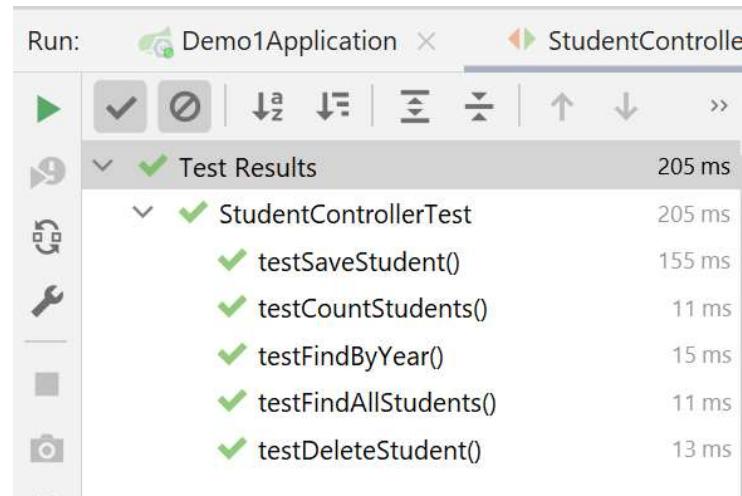
- **Mockito** : Utilisé pour simuler les services. Cela permet de tester le contrôleur sans devoir implémenter le service ou les accès à la base de données.
- **JUnit 5** : Utilisé pour écrire des assertions, qui permettent de vérifier que les résultats des méthodes du contrôleur sont conformes aux attentes.



**Remarque :** Les tests unitaires sont un élément clé pour garantir la fiabilité d'une application. Ils permettent de s'assurer que chaque composant se comporte comme prévu, même lorsqu'il est testé isolément. En combinant JUnit et Mockito, il est possible de simuler des scénarios réalistes tout en maintenant des tests rapides et efficaces.

### Résultats des Tests Unitaires

Après l'exécution des tests unitaires dans l'IDE, les résultats montrent que tous les tests définis dans la classe `StudentControllerTest` ont été exécutés avec succès. Chaque test vérifie une fonctionnalité spécifique du contrôleur `StudentController`, notamment l'ajout, la suppression, la récupération, et le comptage des étudiants.



**Figure 4.6:** Résultats des tests unitaires dans l'IDE.

Tous les tests suivants ont été validés avec succès :

- `testSaveStudent()` : Vérifie que l'étudiant est bien sauvegardé et que le statut CREATED est renvoyé.
- `testCountStudents()` : Vérifie que le nombre total d'étudiants est correctement renvoyé par le service.
- `testFindByYear()` : Vérifie que la récupération des étudiants par année fonctionne et que la collection renvoyée est correcte.
- `testFindAllStudents()` : Assure que la liste de tous les étudiants est récupérée avec succès.
- `testDeleteStudent()` : Vérifie que l'étudiant est correctement supprimé avec un retour NO\_CONTENT.



**Remarque :** Tous les tests se sont terminés en moins de 205 ms, ce qui montre l'efficacité et la rapidité des tests unitaires lorsque les services sont simulés avec Mockito.

#### 4.2.7 Intégration de Swagger

Swagger (via `springdoc-openapi`) est un outil puissant pour documenter les API REST et fournir une interface interactive permettant de tester les endpoints. Avec Spring Boot, il est possible de configurer Swagger automatiquement pour documenter les endpoints de l'API. Voici les étapes détaillées pour ajouter et configurer Swagger avec `springdoc-openapi` dans un projet Spring Boot.

##### Étape 1 : Ajouter la Dépendance Swagger

La première étape consiste à ajouter la dépendance `springdoc-openapi` au fichier `pom.xml`. Cela permet d'intégrer Swagger dans l'application Spring Boot pour générer automatiquement la documentation des endpoints REST.

---

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.0.2</version>
</dependency>
```

---



**Remarque :** Cette dépendance doit être ajoutée sous la balise `<dependencies>` du fichier `pom.xml`, permettant ainsi à Spring Boot de télécharger et d'intégrer Swagger (via `springdoc-openapi`) dans le projet.

## Étape 2 : Configurer Swagger

Après l'ajout de la dépendance, Swagger est automatiquement configuré pour documenter les endpoints des contrôleurs REST. Aucune configuration supplémentaire n'est nécessaire, mais il est possible d'enrichir la documentation en annotant les classes de contrôleurs avec des annotations Swagger. Par exemple, voici une configuration pour un contrôleur qui gère les étudiants :

```
@RestController
@RequestMapping("/api")
public class StudentController {

    @Operation(summary = "Récupérer tous les étudiants")
    @GetMapping("/students")
    public List<Student> getAllStudents() {
        // Code pour récupérer tous les étudiants
    }
}
```

- `@Operation` : Cette annotation décrit le comportement du endpoint. Ici, elle indique que ce endpoint permet de récupérer la liste de tous les étudiants.
- `@GetMapping("/students")` : Cette annotation Spring Boot expose une URL /students qui, lorsqu'elle est appelée via une requête GET, renverra la liste des étudiants.



**Remarque :** Il est possible d'ajouter des annotations similaires pour d'autres méthodes HTTP telles que POST, PUT, et DELETE afin de documenter l'intégralité des endpoints de l'API.

## Étape 3 : Accéder à l'Interface Swagger

Une fois l'application démarrée avec Spring Boot, Swagger (via springdoc-openapi) génère automatiquement une interface interactive accessible via un navigateur web. Pour accéder à cette interface, suivre les étapes suivantes :

1. Démarrer l'application en utilisant la commande suivante dans le terminal :

```
mvn spring-boot:run
```

2. Ouvrir un navigateur et accéder à l'URL suivante : <http://localhost:8080/swagger-ui.html>

The screenshot shows the Swagger UI interface. At the top, there's a navigation bar with the Swagger logo, the URL '/v3/api-docs', and a green 'Explore' button. Below the navigation bar, the title 'OpenAPI definition' is displayed, followed by 'v0 OAS3'. A 'Servers' dropdown menu shows the URL 'http://localhost:8082 - Generated server url'. The main content area is titled 'profile-controller'. It contains two API endpoints: 'GET /profile' and 'GET /profile/students'. Each endpoint has its method, path, and a small dropdown arrow indicating more details.

Figure 4.7: Interface Swagger documentant les endpoints de l'application.

L'interface Swagger permet de :

- Visualiser les différents endpoints de l'application.
- Lire la documentation associée à chaque endpoint, incluant les types de requêtes HTTP acceptées et les réponses attendues.
- Tester les endpoints en envoyant des requêtes directement depuis l'interface utilisateur, sans avoir besoin d'utiliser des outils externes comme Advanced Rest Client.



**Remarque :** Swagger est un outil très utile pour générer de la documentation en temps réel et tester les fonctionnalités d'une API sans quitter le navigateur. L'intégration de Swagger facilite également la collaboration entre les développeurs, notamment dans les équipes distantes.