JAVA ACADEMY - XIDERAL MONTERREY, N.L

WEEK 4
SPRING REST + JPA

Made by:
Luis Miguel Sánchez Flores

INTRODUCTION

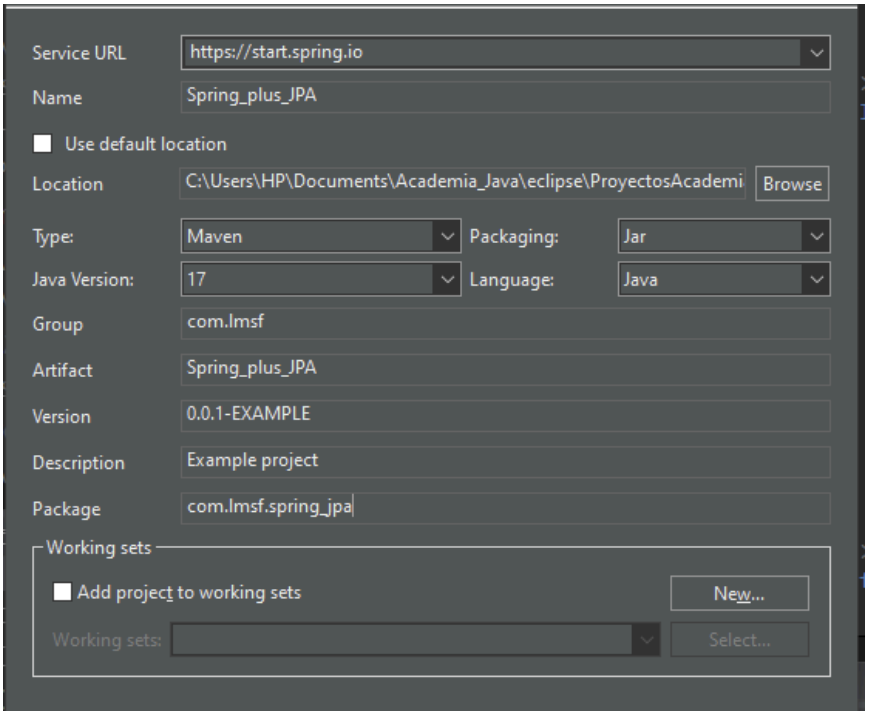
RESTFul API has become a *defacto* standard for data exchange, with most of the business applications having to communicate with both internal and external applications to successfully perform their tasks, and RESTFul API provides a simplistic, scalable and fast way to handle requests and responses for all kinds of data.
Because of this, many resort to the use of frameworks and libraries that allow them to implement these RESTFul services in a reliable and timely manner. One such powerful framework that simplifies the creation of RESTFul web services is **Spring REST.**

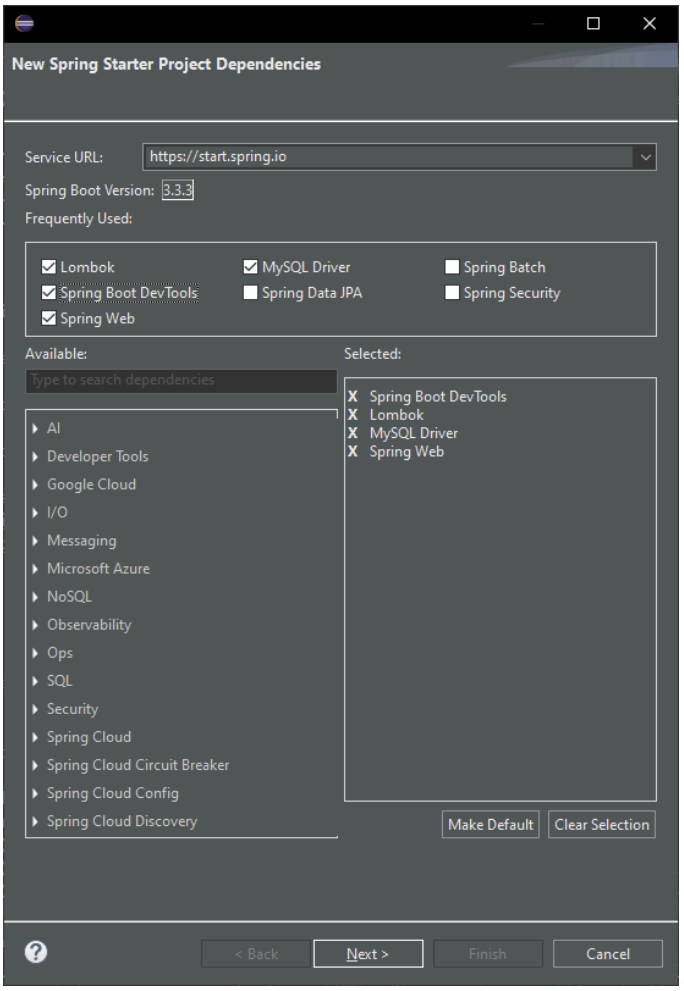
Spring REST provides a powerful toolkit for building flexible, secure and well-structured RESTFul web services through built-in components and a clear architectural pattern that allows developers to expose resources as endpoints in a secure and effective way. Spring REST often goes hand-in-hand with data persistence technologies like JPA to create a data-driven application that interacts with either an embedded or external database for CRUD operations. As such, developers are able to showcase persisted data through sophisticated RESTFul endpoints, enabling the creation of dynamic web applications.

Furthermore, Spring REST provides robust exception handling by providing meaningful HTTP status codes and error messages to clients or handling errors in a desired way, improving the overall user experience with the application.
Spring REST offers numerous benefits such as ease of use and effective management of HTTP requests, and when paired with JPA, it can provide a comprehensive solution that delivers high-quality APIs to access and manipulate data within a database.

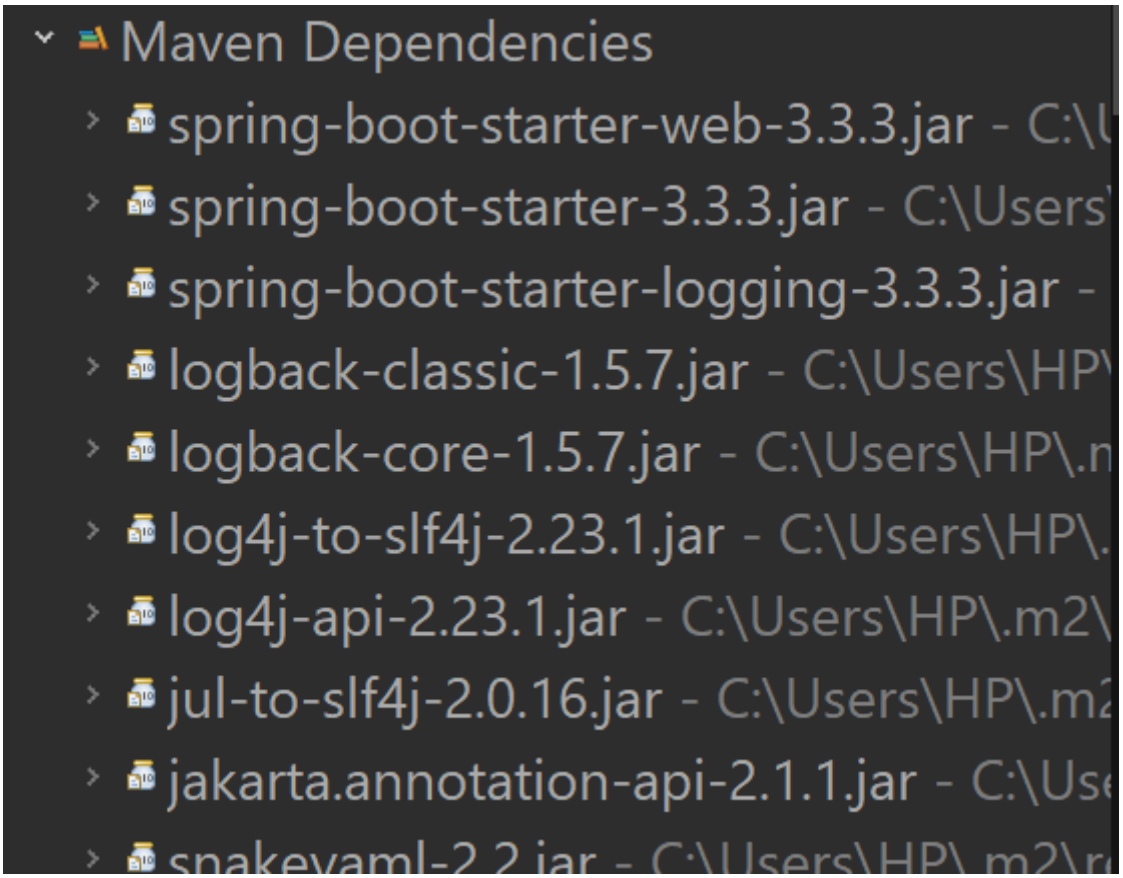
Setting up the project

Spring project can be easily set up either using the web-based **Spring Initializr** tool that can quickly generate a basic project structure with the necessary dependencies and configurations, or by leveraging the **Spring Tool Suite plugin** included in popular IDEs such as Netbeans and Eclipse to start off a Spring project in a direct way. The latter was used to generate the project, ensuring that it is a Maven project and using Java version 17, and then including the dependencies such as Spring Web and MySQL Driver to generate the REST application and be able to connect to the database, respectively. Lombok is also included to reduce boilerplate code when creating the entity:





The tool suite will take care of the Maven dependencies, installing them and building up the basic project structure.



With the project set up, the application properties were configured to setup the connection of the application with the database to be generated, including the following properties:

```
# SPRING + JPA project

spring.application.name=Spring_plus_JPA
spring.jpa.show-sql=true

#DataSource
spring.datasource.url = jdbc:mysql://localhost:3306/school_example
spring.datasource.username=springstudent
spring.datasource.password=springstudent

#Schema initialization with initial data:
spring.jpa.defer-datasource-initialization=true
spring.sql.init.mode=ALWAYS
spring.sql.init.schema-locations=classpath:schema.sql
spring.sql.init.data-locations=classpath:data.sql
```

Here, the application is going to set up a connection with the database of “school_example” which, as the name implies, will represent a database of a school administration, with the data of its students.
The application is designated with the username and password of “springstudent” to successfully connect to the database. The username and password were generated with the following SQL code:

```sql
-- Drop user first if they exist
DROP USER if exists 'springstudent'@'%' ;

-- Now create user with prop privileges
CREATE USER 'springstudent'@'%' IDENTIFIED BY 'springstudent';

GRANT ALL PRIVILEGES ON * . * TO 'springstudent'@'%';
```

Along with the creation of the user for the database, SQL scripts were developed to generate the `students` table that will include the students’ data , as well as to insert 10 initial records to the same table in order to start experimenting with CRUD operations later on:

schema.sql

```sql
DROP TABLE IF EXISTS `students`;

CREATE TABLE students (
  id INT AUTO_INCREMENT PRIMARY KEY,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  date_of_birth VARCHAR(50) NOT NULL,
  email VARCHAR(255) UNIQUE,
  gender VARCHAR(20),
  phone_number VARCHAR(20)
);
```

data.sql

```sql
insert into students (id, first_name, last_name, email, gender, date_of_birth, phone_number) values
                    (1, 'Chad', 'Barfitt', 'cbarfitt0@simplemachines.org', 'Other', '27-06-2002', '1171887923'),
                    (2, 'Millard', 'Simonich', 'msimonich1@google.ca', 'Male', '29-08-2004', '4977160910'),
                    (3, 'Mick', 'MacMeanma', 'mmacmeanma2@blinklist.com', 'Male', '25-05-2001', '3294515237'),
                    (4, 'Etti', 'Hendrikse', 'ehendrikse3@jigsy.com', 'Female', '10-02-2004', null),
                    (5, 'Dunn', 'Meneur', 'dmeneur4@photobucket.com', 'Male', '08-07-2003', null),
                    (6, 'Selia', 'Furneaux', 'sfurneaux5@i2i.jp', 'Female', '16-11-2002', '8747941454'),
                    (7, 'Nigel', 'Humfrey', 'nhumfrey6@over-blog.com', 'Male', '04-07-2005', '6722010693'),
                    (8, 'Alikee', 'Hugonnet', 'ahugonnet7@marriott.com', 'Female', '13-06-2004', '8615206281'),
                    (9, 'Willey', 'Struys', 'wstruys8@mail.ru', 'Male', '12-04-2004', '6144725276'),
                    (10, 'Farrand', 'Kolinsky', 'fkolinsky9@blogtalkradio.com', 'Female', '02-12-2003', '6638043043');
```

The two SQL files are referenced for the initialization of the table with the initial data in the `application.properties` file, so that the table is regenerated each time the Spring application is launched.

CREATED STUDENTS TABLE WITH INITIAL DATA:

```sql
SELECT * FROM school_example.students;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Conte

id	first_name	last_name	date_of_birth	email	gender	phone_number
1	Chad	Barfitt	27-06-2002	cbarfitt0@simplemachines.org	Other	1171887923
2	Millard	Simonich	29-08-2004	msimonich1@google.ca	Male	4977160910
3	Mick	MacMeanma	25-05-2001	mmacmeanma2@blinklist.com	Male	3294515237
4	Etti	Hendrikse	10-02-2004	ehendrikse3@jigsy.com	Female	NULL
5	Dunn	Meneur	08-07-2003	dmeneur4@photobucket.com	Male	NULL
6	Selia	Furneaux	16-11-2002	sfurneaux5@i2i.jp	Female	8747941454
7	Nigel	Humfrey	04-07-2005	nhumfrey6@over-blog.com	Male	6722010693
8	Alikee	Hugonnet	13-06-2004	ahugonnet7@marriott.com	Female	8615206281
9	Willey	Struys	12-04-2004	wstruys8@mail.ru	Male	6144725276
10	Farrand	Kolinsky	02-12-2003	fkolinsky9@blogtalkradio.com	Female	6638043043
NULL	NULL	NULL	NULL	NULL	NULL	NULL

Setting up the persistence layer / JPA repository

In order to interact with the database, the persistence layer is developed, which will be responsible for storing, updating and retrieving data within a database based on user requests in a consistent way.
It is here where the JPA type repository is implemented that will manage CRUD operations with the school database by means of an entity mapping the `Student` entity created previously into the `students` table.

For this purpose, the DAO pattern is used to structure the persistence layer and encapsulate the logic about data access, developing the following interface :

```java
package com.lmsf.spring_and_jpa.dao;

import java.util.List;

// Data Access Object (DAO)
public interface StudentDAO {

    //CREATE & UPDATE
    Student saveStudent(Student newStudent);

    //READ
    Student findByID(int id);
    long countNumberOfStudents();

    List<Student> findAll();
    List<Student> findStudentsByName(String firstName, String lastName);
    List<Student> findStudentsWithoutPhoneNum();
    List<Student> findStudentsByGender(Gender gender);


    //DELETE
    void deleteStudent(int id);

}
```

For the repository implementation, the `EntityManager` is injected into it that will play the role of a bridge between the application and the MySQL

database, keeping track of the entities that are created and managing to execute the CRUD operations through the use of POJOs:

```
16 @Repository
17 public class StudentDAOImpl implements StudentDAO {
18
19     @PersistenceContext
20     private EntityManager entityManager;
21
22
23
24     public StudentDAOImpl(EntityManager entityManager) {
25         this.entityManager = entityManager;
26     }
27
```

For the implementation of the functions defined in the `StudentDAO` interface, the EntityManager is leveraged to perform CRUD operations for the application, such as saving or updating a student using `merge()`, retrieve a `Student` instance by the `find()` method and eliminate a student from the database with `remove()`:

```
// Save a new student, or update an existing student's data:
@Override
public Student saveStudent(Student newStudent) {
    return entityManager.merge(newStudent);
}

// Find a student by their ID:
@Override
public Student findByID(int id) {
    return entityManager.find(Student.class, id);
}
```

It also provides methods for generating and executing queries such as `createQuery()` to manage or manipulate records based on desired conditions or criteria. We can either use native SQL queries or JPQL (Java Persistence Query Language) queries that make use of Java entities instead of the database table to retrieve data:

```java
// Get all the students in the database:
@Override
public List<Student> findAll() {

    // JPQL Query:
    TypedQuery<Student> searchQuery =
            entityManager.createQuery("FROM Student", Student.class);

    // Retrieve the list of results from the query:
    return searchQuery.getResultList();
}
```

A `TypedQuery` provides a more safer approach when executing JPQL queries by making sure that the correct type of object is retrieved, as well as safely substituting parameter placeholders with real values that prevents SQL injection, as demonstrated in the following method to retrieve a list students by their first and last name, setting up the arguments of said function as the parameters of the query to execute:

```java
// Retrieve students based on their first and last name:
@Override
public List<Student> findStudentsByName(String firstName, String lastName) {

    // JPQL Query
    TypedQuery<Student> searchQuery =
            entityManager.createQuery("FROM Student WHERE firstName LIKE :fname AND lastName LIKE :lname", Student.class);

    // Change placeholders with the provided arguments:
    searchQuery.setParameter("fname", firstName+"%");
    searchQuery.setParameter("lname", lastName+"%");

    // Return the list:
    return searchQuery.getResultList();
}
```

Creating the Service and Controller layer

With the persistence layer done, the service layer is created that will interact with the former to perform data operations, while taking care of the business logic that will dictate the way in which database entities will be created, updated, read and deleted, as well as acting as the intermediary between the controller and the repository by transforming the data into the appropriate representation. As with the persistence layer, an interface is created to define the contract of the operations that can the service can perform onto the database:

```java
public interface StudentService {

        //CREATE & UPDATE
        Student saveStudent(Student newStudent);

        //READ
        Student findByID(int id);
        long countNumberOfStudents();

        List<Student> findAll();
        List<Student> findStudentsByName(String firstName, String lastName);
        List<Student> findStudentsWithoutPhoneNum();
        List<Student> findStudentsByGender(Gender gender);


        //DELETE
        void deleteStudent(int id);
}
```

Within the implementation, the JPA repository is injected into it with the `studentDAO` variable, required to interact with the student database. The functions defined by the interface in which the `StudentDAO` methods are called to execute the desired CRUD operation on the database, along with the required business logic to achieve the desired behavior when interacting with the database:

```java
@Service
public class StudentServiceImpl implements StudentService{

    // Inject the JPA repository needed for interacting with the database:
    @Autowired
    private StudentDAO studentDAO;
```

```java
@Override
@Transactional // <- Specify as a transaction process to ensure data consistency
public Student saveStudent(Student newStudent) {

    Student addedStudent = studentDAO.saveStudent(newStudent);

    return addedStudent;

}
```

```java
@Override
public Student findByID(int id) {

    Student retrievedStudent = studentDAO.findByID(id);

    if(retrievedStudent == null) // <- If there was no student retrieved, throw a RuntimeException:
        throw new RuntimeException("\nNo student with ID #"+id+ " was found...");


    return retrievedStudent;
}
```

On the other hand, the controller layer is created with the `StudentController` class (specified as the REST Controller), which will be responsible for managing incoming HTTP requests, processing them and returning the desired response. It interacts with the service layer by delegating the processing of requests and demonstrating the results obtained from the service.
With the `@Autowired` annotation the service layer is injected into the controller, which is then administered by Spring to insert the appropriate service implementation for said controller.

The controller defines the endpoints of the REST applications, mapping the appropriate HTTP methods such as GET, POST, PUT and DELETE to indicate the type of request the user wants to execute, such as reading, adding,

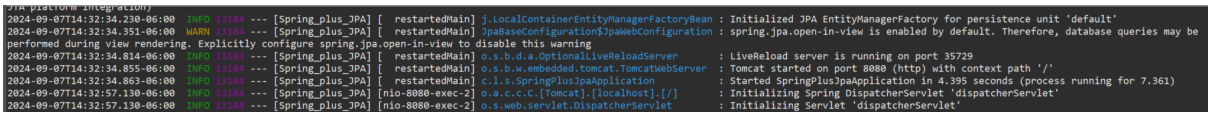
updating and deleting data from the database. This is done using the `@Mapping` annotation, specifying the URL path in which to execute the request, as well as its type.

```java
//GET Requests examples:

@GetMapping("/students")
public List<Student> findAllStudents() {
    return studentService.findAll();
}

@GetMapping("/students/size")
public String countStudents() {
    return "Number of students in database : " + studentService.countNumberOfStudents();
}

@GetMapping("/students/gender/{gender}")
public List<Student> searchByGender(@PathVariable Gender gender) {
    return studentService.findStudentsByGender(gender);
}

@GetMapping("/students/no-phone-number")
public List<Student> searchStudentsWithoutNumber() {
    return studentService.findStudentsWithoutPhoneNum();
}
```

For some methods, a `@PathVariable` annotation is provided by Spring to handle input data that the controller receives from the user in order to form the desired response. For example, the `findStudentsByGender()` method adds the gender input data passed in the request URL as an argument for the service in order to retrieve the list of students of the desired gender.

Running the application and Results

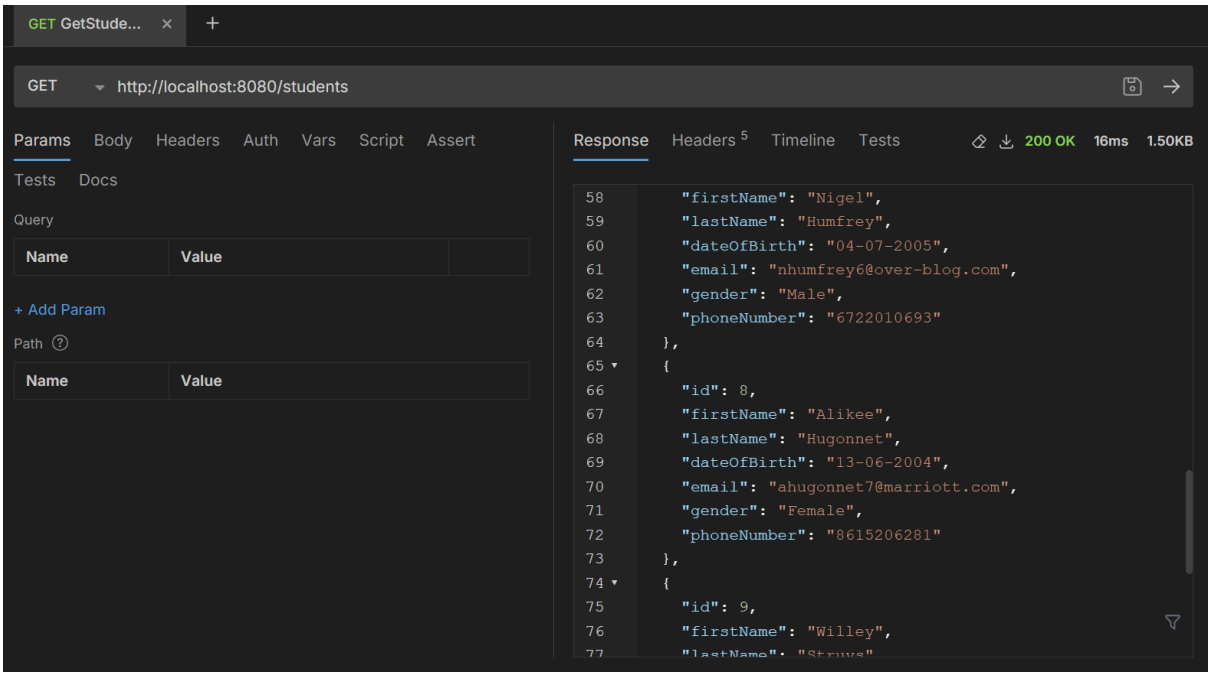
With all the components and layers in order, we can start trying out the REST application with a simple run command, as Spring takes care of configuring and deploying automatically an embedded server such as Tomcat, based on the `application.properties` file, streamlining the process of deploying and running Java web applications:



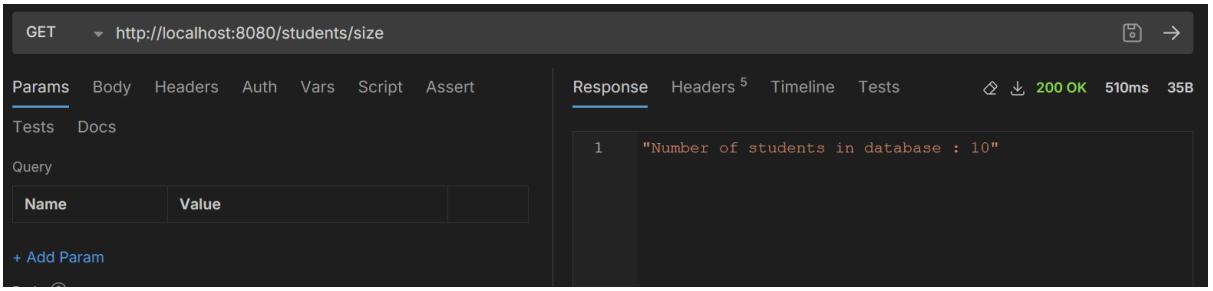
With the server up and running, we can start making requests to the application. We can use any browser such as **Chrome** or **Firefox** to make GET requests. But for POST, PUT and DELETE requests, special programs like **Postman** and **Bruno** are used to send the information to an API in a simple and timely manner. In this case, Bruno is used to send the requests to the localhost:8080 URL where the server was deployed on.

Here are some results obtained with the HTTP requests made to the Spring REST application:

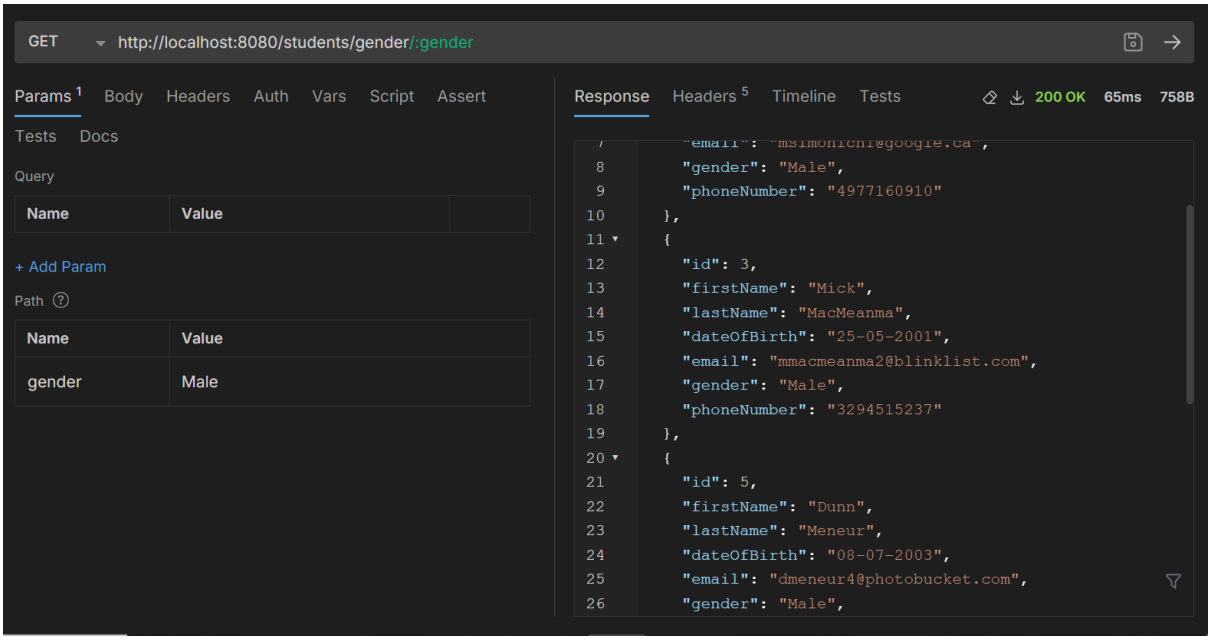
Retrieve All the students:



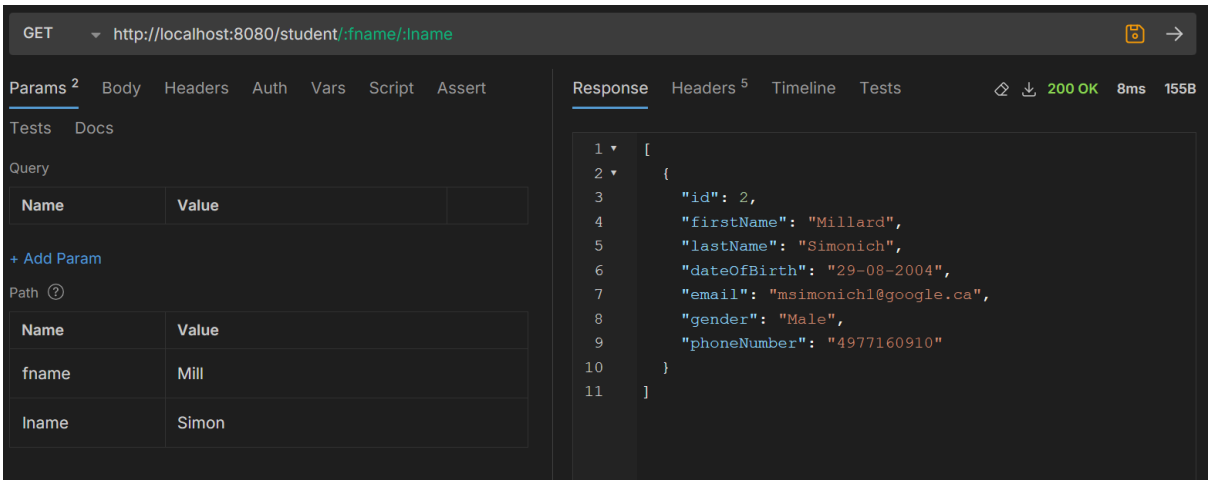
Get the number of students in the database:



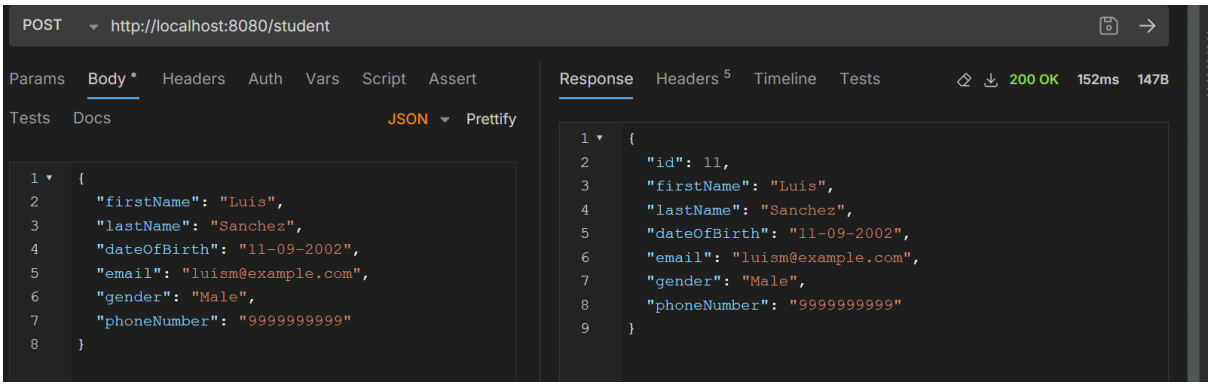
Get the list of male students (passing Male as the path value):

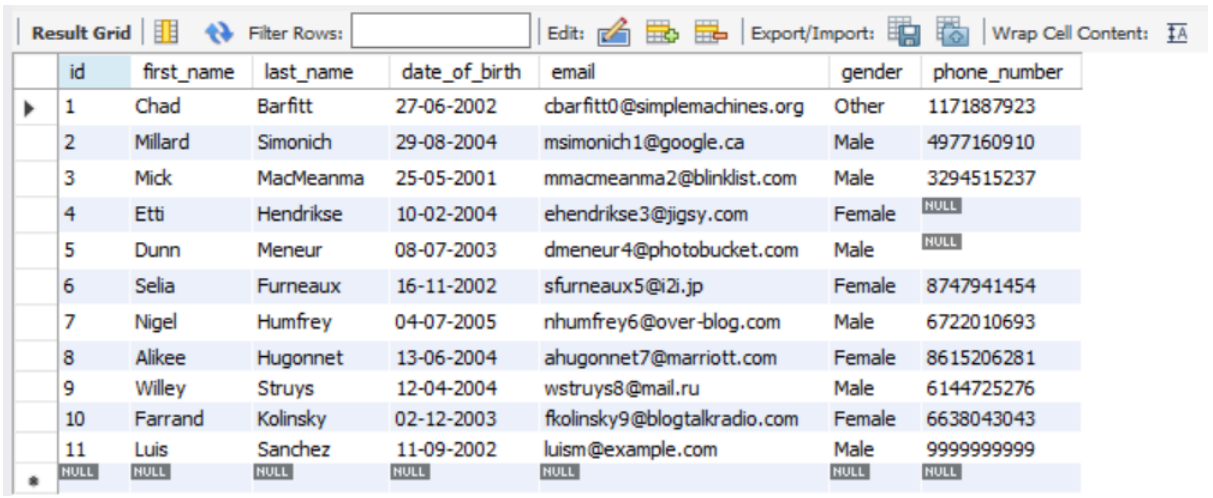


Get student by first and last name:



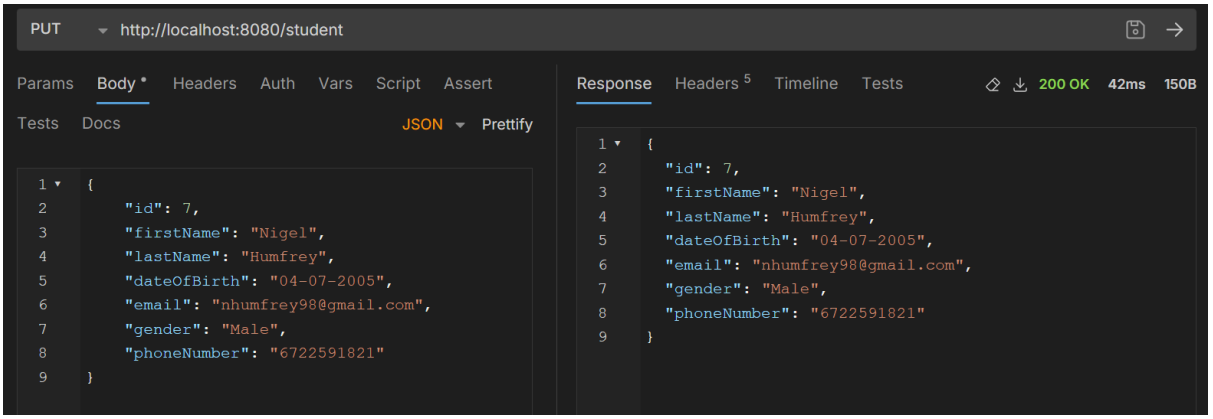
Add student to the database:





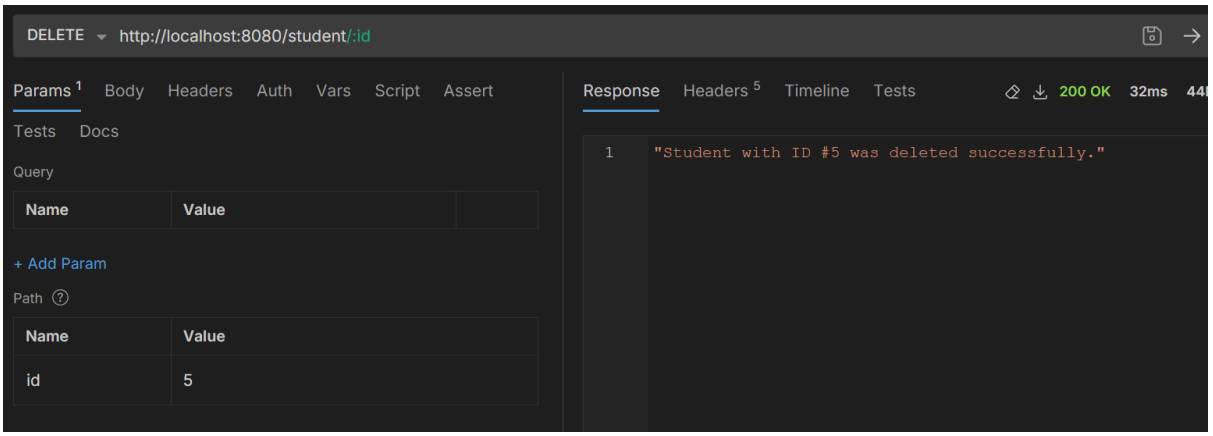
id	first_name	last_name	date_of_birth	email	gender	phone_number
1	Chad	Barfitt	27-06-2002	cbarfitt0@simplemachines.org	Other	1171887923
2	Millard	Simonich	29-08-2004	msimonich1@google.ca	Male	4977160910
3	Mick	MacMeanma	25-05-2001	mmacmeanma2@blinklist.com	Male	3294515237
4	Etti	Hendrikse	10-02-2004	ehendrikse3@jigsy.com	Female	NULL
5	Dunn	Meneur	08-07-2003	dmeneur4@photobucket.com	Male	NULL
6	Selia	Furneaux	16-11-2002	sfurneaux5@i2i.jp	Female	8747941454
7	Nigel	Humfrey	04-07-2005	nhumfrey6@over-blog.com	Male	6722010693
8	Alikee	Hugonnet	13-06-2004	ahugonnet7@marriott.com	Female	8615206281
9	Willey	Struys	12-04-2004	wstruys8@mail.ru	Male	6144725276
10	Farrand	Kolinsky	02-12-2003	fkolinsky9@blogtalkradio.com	Female	6638043043
11	Luis	Sanchez	11-09-2002	luism@example.com	Male	9999999999
NULL	NULL	NULL	NULL	NULL	NULL	NULL

Update student #7 info (email and phone number):



7	Nigel	Humfrey	04-07-2005	nhumfrey98@gmail.com	Male	6722591821

Delete the student with ID #5:



id	first_name	last_name	date_of_birth	email	gender	phone_number
1	Chad	Barfitt	27-06-2002	cbarfitt0@simplemachines.org	Other	1171887923
2	Millard	Simonich	29-08-2004	msimonich1@google.ca	Male	4977160910
3	Mick	MacMeanma	25-05-2001	mmacmeanma2@blinklist.com	Male	3294515237
4	Etti	Hendrikse	10-02-2004	ehendrikse3@jigsy.com	Female	NULL
6	Selia	Furneaux	16-11-2002	sfurneaux5@i2i.jp	Female	8747941454