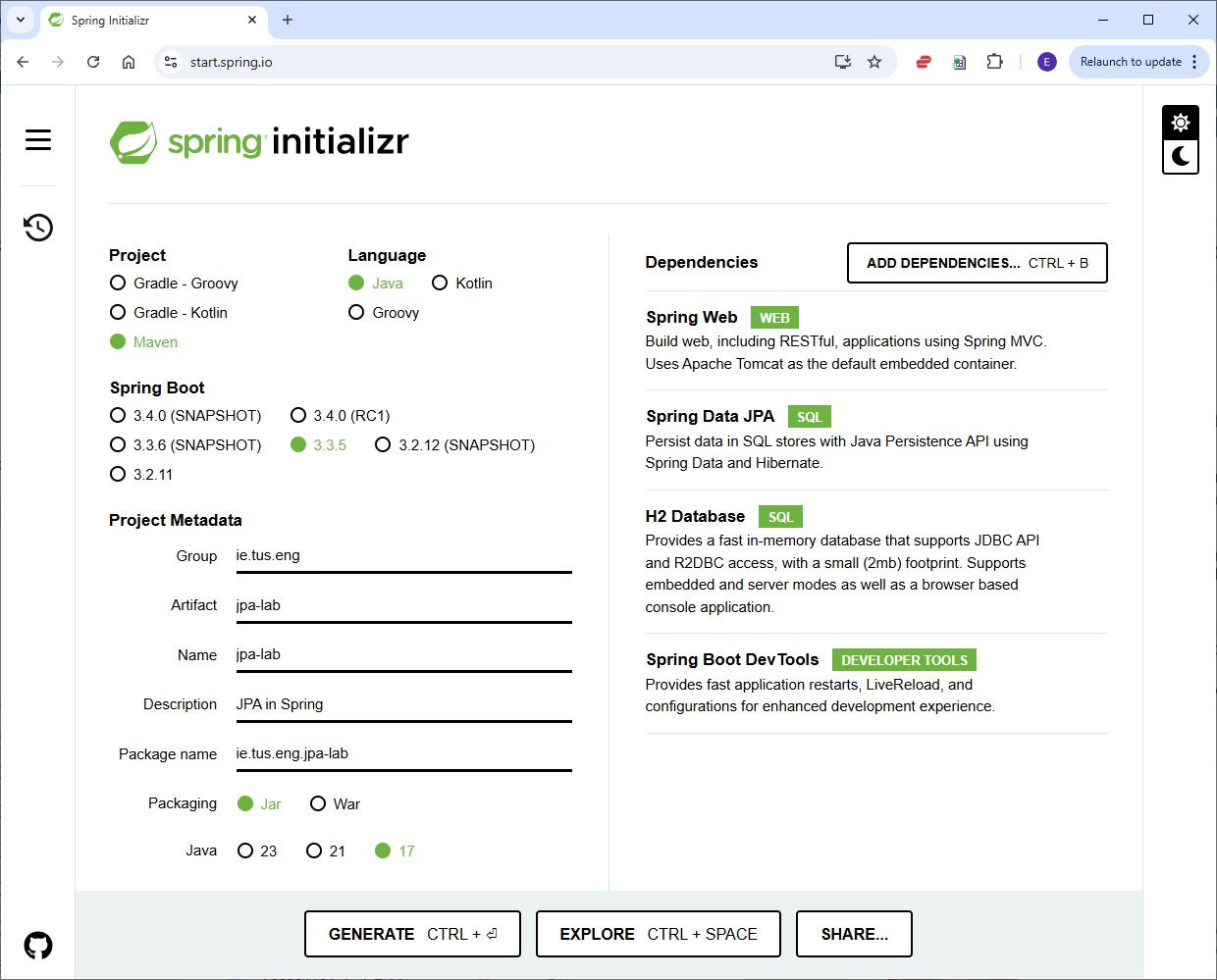
**JPA – Java Persistence API**

This lab introduces JPA (Java Persistence API) which is the Java standard for persistence and object/relational mapping. We’ll use JPA to simplify our code from the previous lab by doing the following:

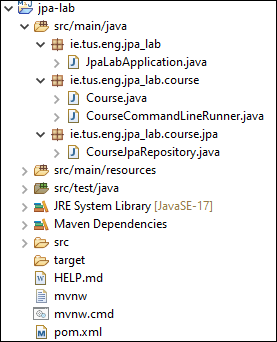
1. Database table will be automatically constructed based on the Course class properties. i.e. we don’t have to write a ‘create table course…’
2. We don’t need the JDBC style code to convert from a table to an object, so no need for ResultSet objects.

This lab achieves the same as the previous lab, but you’ll see that the code is much simpler.

Create and download the following project:



Create the following Java files taking care to get the package structure correct:



(Note: some of the code we’ll use is similar to the last lab, so you can copy and paste where appropriate)

**Spring Data JPA**

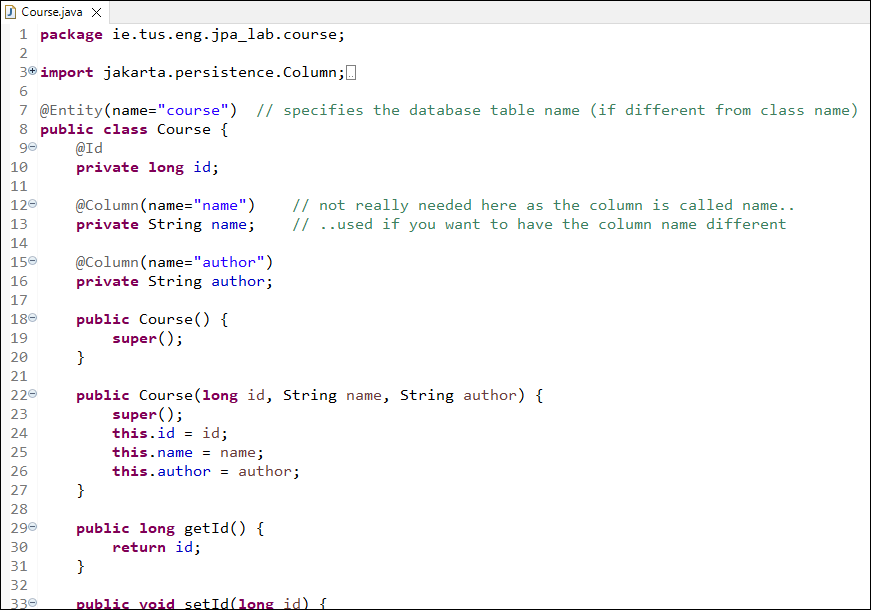
So far, we’ve seen how to use JDBC in Spring. Next, we want to look at JPA which maps objects (beans) to database tables.

**ORM (Object Relational Mapping) Framework:**

As mentioned above, JPA is a java standard for object/relational mapping i.e. it seamlessly stores a Java object in a relational database. ***Hibernate*** is an *implementation* of this standard and is widely used; hibernate is the implementation that comes with Spring. Hibernate is known as an ORM Framework.

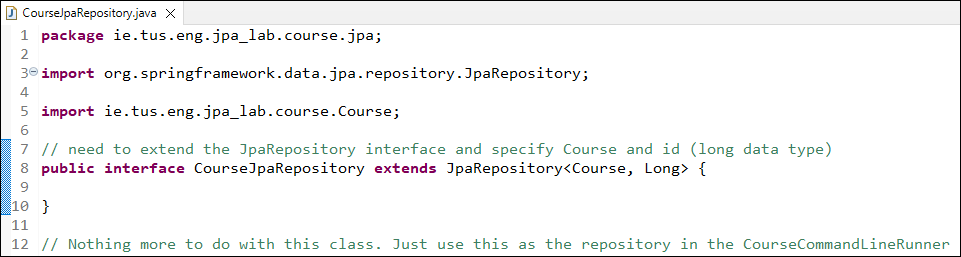
To enable the object/relational mapping, we first need to mark our Course class as an entity (table) with the @Entity annotation. We use the @Id annotation to mark the primary key, and the @Column annotation to mark the table columns.

Add the following annotations to Course.java (keep all the getters/setters as before):



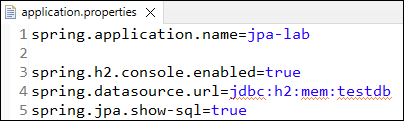
(Note: generate all getter/setters and a toString() method for the Course class.)

Create a new repository class (**interface** actually) as follows:

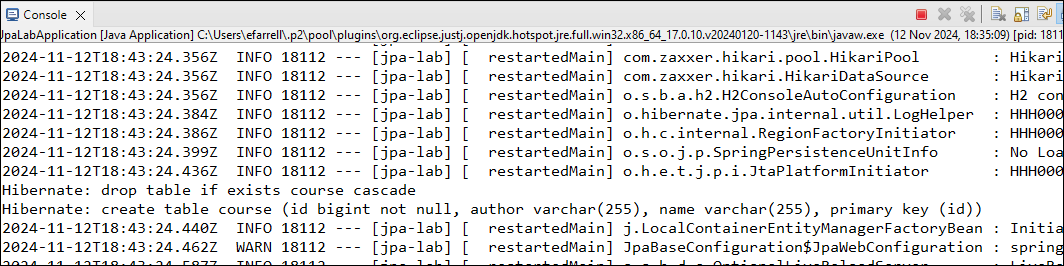


We only need to specify the interface here – Spring Data JPA uses a concept called dynamic proxy generation to create a proxy instance of this interface at runtime. The proxy object implements the CRUD operations (save, findAll, findById, etc) using the underlying ORM framework, Hibernate.

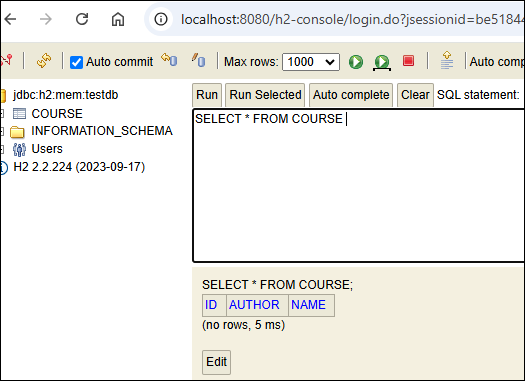
It can be useful to see what SQL is generated by our JPA commands. To do this we can add a line to the application.properties file (line 5):



Now, if we restart the server, we can see the SQL printed out in the log files.

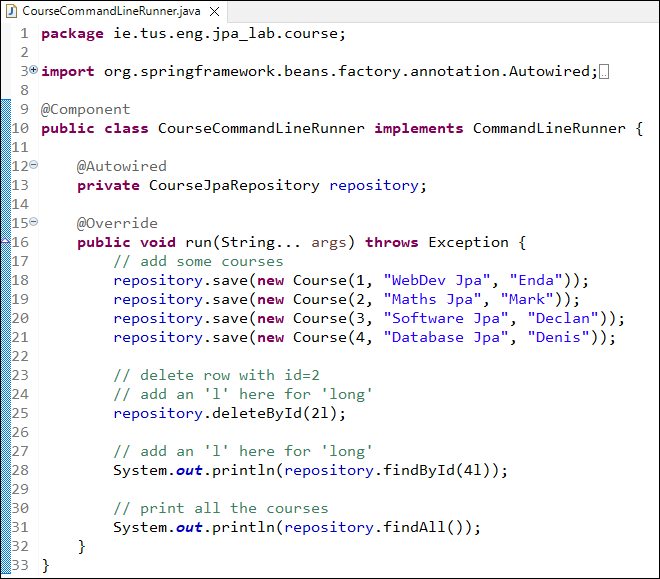


Open the h2-console:



Note that the table was created without having to write any DDL code. The table was created based on the Course class which we annotated with the @Entity, @Id and @Column annotations.

The CourseCommandLineRunner is as follows:



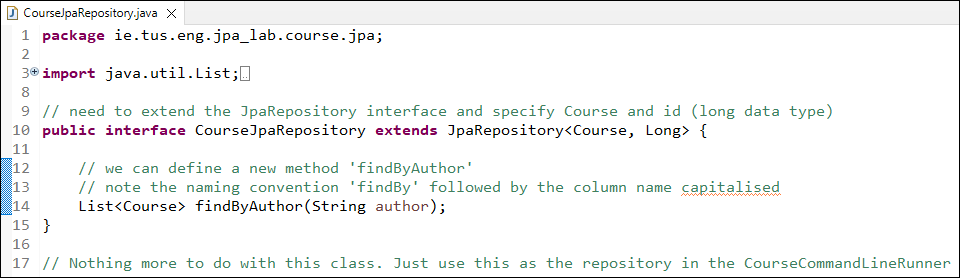
The main changes from the previous lab are:

* The repository is CourseJpaRepository
* The ‘insert’ commands change to ‘save’ (as Spring Data Jpa uses ‘save’)
* We add an ‘l’ to the numbers to show that they are ***long*** values

Rerun the server and it should work as before. Note that very little code needed to be written here.

**Other useful methods:**

If we want a method to search by another column e.g. author, we can specify a method in our interface as follows:



Note that there’s a naming convention that we use ‘findBy’ followed by the column name capitalised e.g. findByAuthor.

We specify that the method returns a list of Course objects and it takes a string as input. This is all we need to do. Now try searching by adding the following to ***CourseCommandLineRunner***

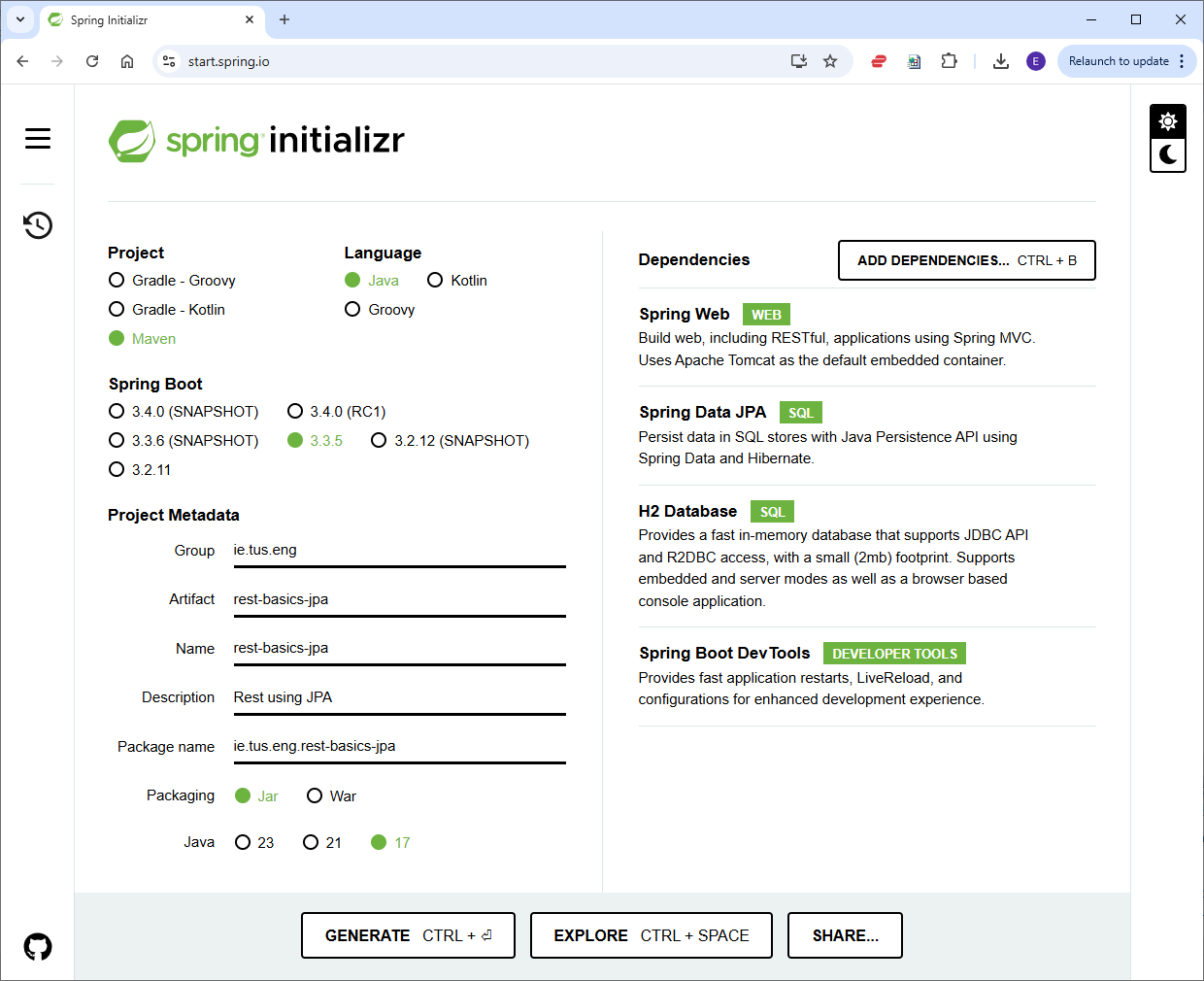


**Exercise:**

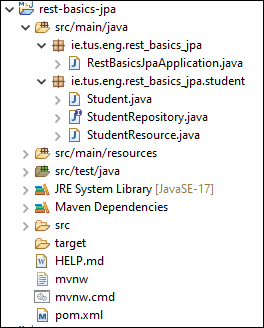
Make a new method to search the database by course name. Test the new method by calling from the CourseCommandLineRunner.

**Connecting the REST API to the database using JPA**

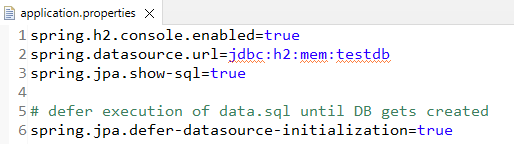
We’ll now return to our Student example and connect it to an H2 database using JPA.



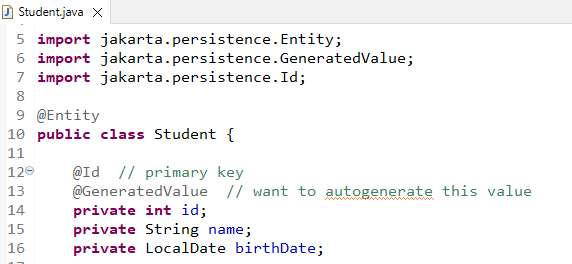
Create the following Java files taking care to get the package name correct.



The first step is to add some data settings to the **application.properties** file in **src/main/resources**. We’ve seen most of these in the previous section. Here we’ve added line 6, which defers running the script called data.sql (which inserts dummy data into the database – see below) until the database gets created. The database is created automatically with JPA after we add the @Entity and associated annotations to the Student class.

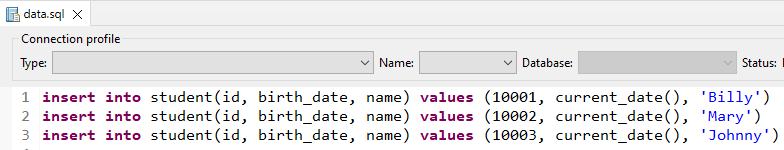


Now, we’ll provide the required annotations for the Student class. Add the @Entity, @Id and @GeneratedValue annotations as follows:



If you start the application and go to the h2-console, you should see the Student table is created.

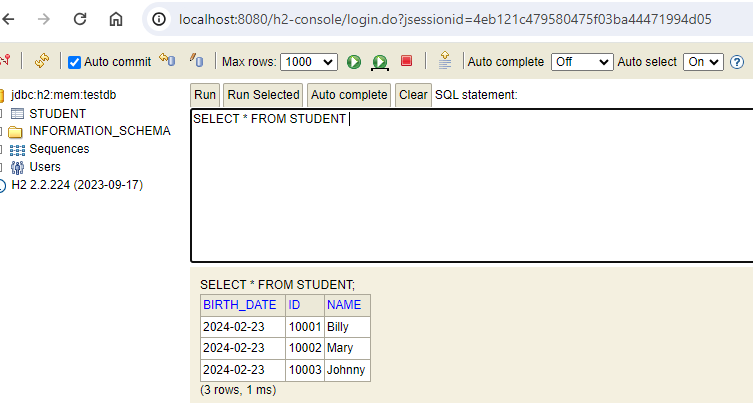
In order to insert dummy data into the database, create a new file in **src/main/resources** called **data.sql**. The **data.sql** file is automatically executed when the application starts. Enter the following insert statements into the file:



**Note:**

* JPA automatically converts camelCase to snake\_case. i.e. birthDate gets converted to birth\_date)
* We’re specifying ID values here for the dummy data. We have also specified that we want ID to be autogenerated; we’ll see the autogenerated values when we do some POST calls later.

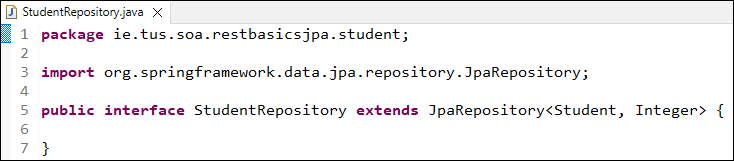
The database should now be created and have the dummy data installed:



**Part 2: Student Repository**

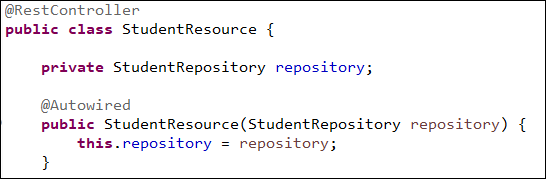
Previously, we had our StudentResource class set up to talk to the StudentDao, which in turn talks to the database. When using JPA, we’ll use a **StudentRepository** class instead of **StudentDao**.

Create a StudentRepository interface:



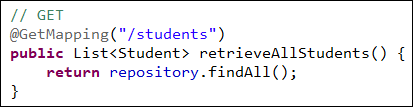
**Edit the StudentResource to use our StudentRepository:**

Our StudentResource class now uses the StudentRepository to interact with the database.

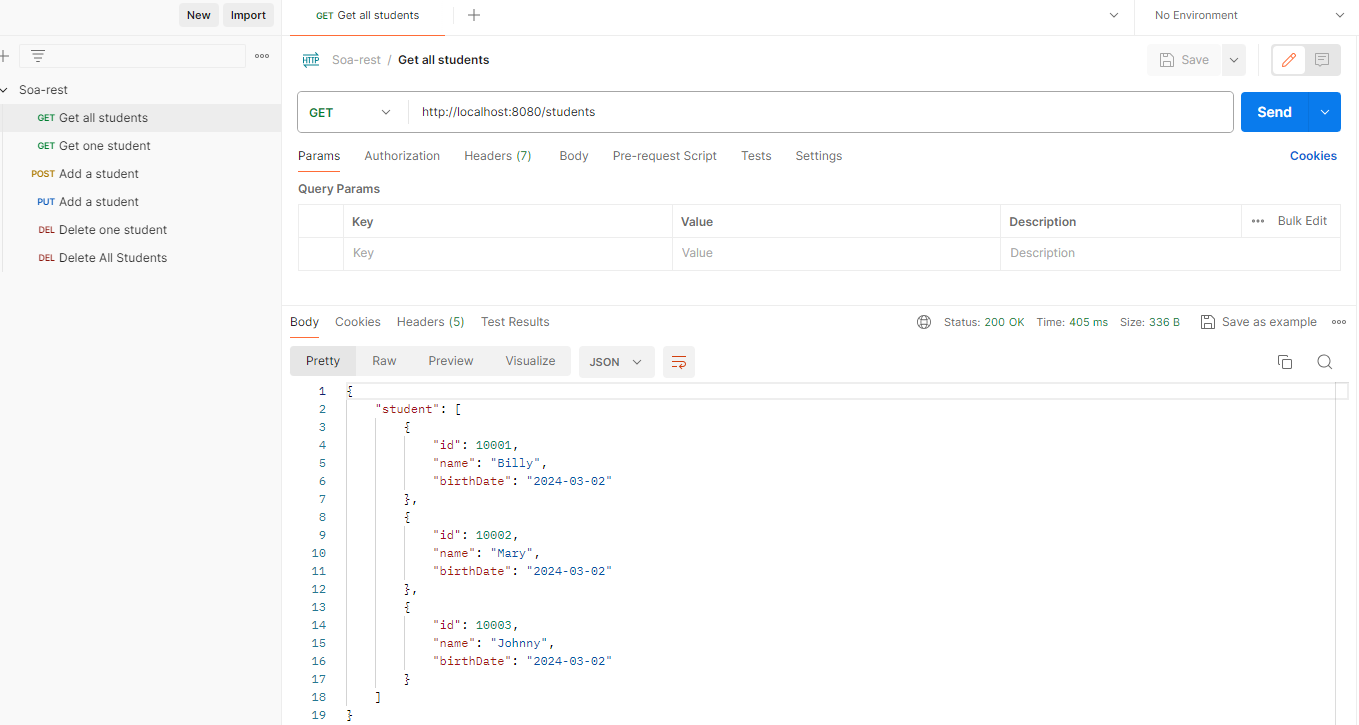


**Getting all students:**

The retrieveAllStudents() method uses StudentRepository as follows:



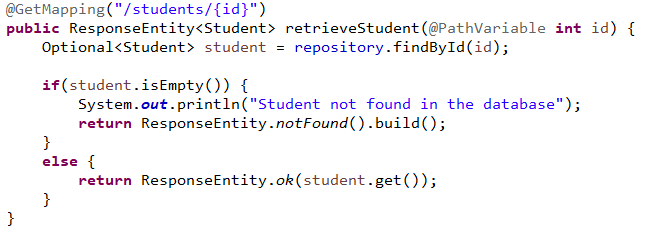
Now test in Postman. You should see the new values you previously added in the data.sql above.



**Getting one student:**

There’re a few changes needed in the retrieveStudent class. The repository returns **Optional<Student>** which deals with the situation where there’s no student with the specified id.

We can use **student.isEmpty()** to check if no student was found at the specified id. Then we need to use **student.get()** to retrieve the student.



**Exercise**:

Get the application working for the following methods below. The methods are all coded in the **StudentResource** class.

(Hint: Use the **StudentResource** class you completed in the **rest-basics** project for inspiration!)

1. Change the **deleteStudent** and test the change.
   1. You should check if a student with the specified id exists first;
      1. if it does return a 204 (No Content)
      2. if it doesn’t return a 404 (Not Found)
2. Change the **deleteAllStudents** method.
   1. Return a 204 (No Content)
3. Change the **createStudent** (POST) method and test the change.
   1. As before, return a 201, along with the location
4. Change the **editStudent** (PUT) method.
   1. First check to make sure the student exists;
      1. If it doesn’t, return a 404 (Not Found)
      2. If it does, use repository.save() to save the edited student details

Test all the changes using Postman – the tests you created previously should work.