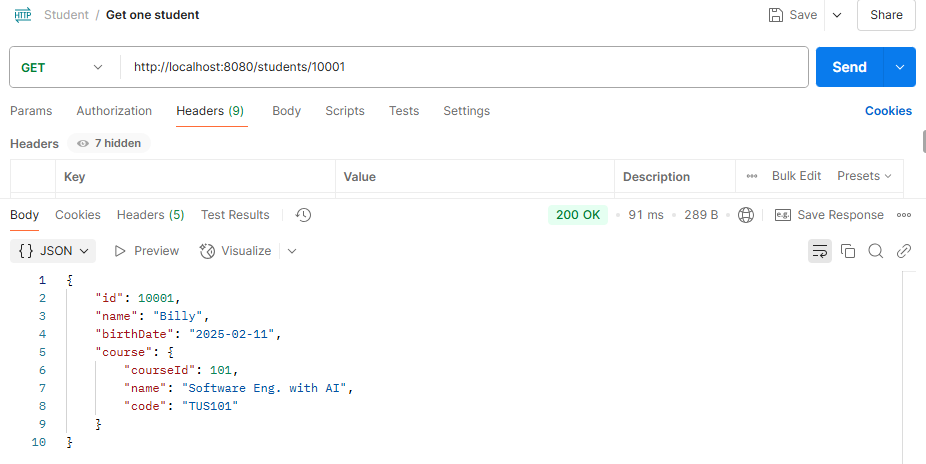
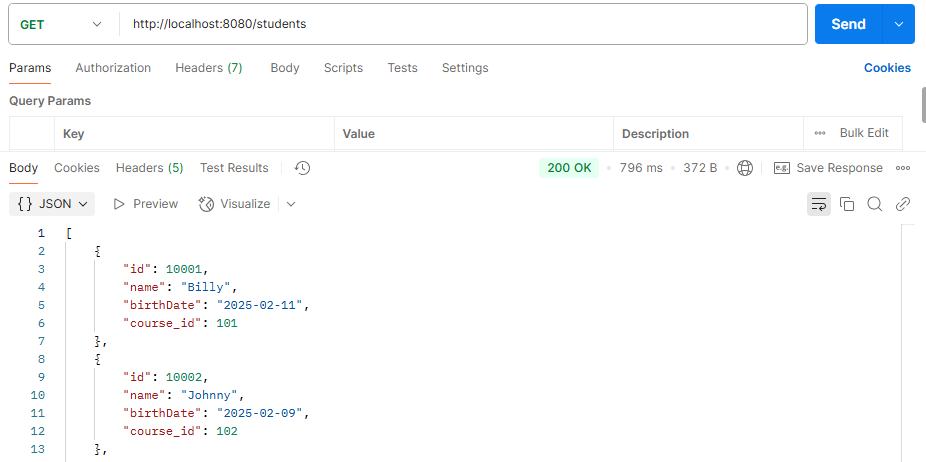
**~~Lab 10 – Connecting Microservices~~**

~~This lab focuses on connecting the~~ **~~Student~~** ~~and~~ **~~Course~~** ~~microservices. We’ll use the Synchronous Blocking style here. When we access the Student microservice, the~~ **~~Student~~** ~~service should query the~~ **~~Course~~** ~~service to get details about the~~ **~~Course~~** ~~the student is enrolled on:~~



~~Firstly, we’ll have to add a~~ **~~course\_id~~** ~~to the~~ **~~Student~~** ~~class to make the connection between the two services. When we get~~ **~~all students,~~** ~~we should see the following (note the added~~ **~~course\_id~~**~~):~~



~~We’ll also change our~~ **~~Course~~** ~~data a little to be more accurate e.g. Software Engineering (AI), Software Engineering (Game), Computer Engineering etc. (previously it was more like subject/module data).~~

**Part 1: Preparing the Services**

**Preparing the Student service:**

1. ~~Make a copy of the~~ **~~rest-basics~~** ~~project and call it~~ **~~student-service~~** ~~(right-click, copy and paste in Project Explorer).~~
2. ~~Change the~~ **~~Student~~** ~~class so that it has a~~ **~~course\_id~~** ~~attribute i.e. add a new private member variable in the~~ **~~Student~~** ~~class; then delete and recreate the constructors, getters and setters.~~
3. ~~Edit your~~ **~~data.sql~~** ~~file to add a~~ **~~course\_id~~** ~~for each mock student.~~
4. ~~Run the service and check with Postman to ensure everything is still working correctly.~~

**Preparing the Course service:**

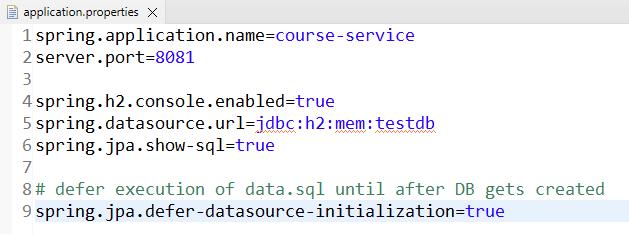
1. ~~Make a copy of the~~ **~~jpa-lab~~** ~~and call it~~ **~~course-service~~**
2. ~~Change the~~ **~~Course~~** ~~class to have the attributes course\_~~*~~id~~*~~,~~ *~~name~~* ~~and~~ *~~code~~* ~~(not author). You’ll have to delete and regenerate the constructors, getters and setters~~
3. ~~Write a~~ **~~data.sql~~** ~~file to insert data into the database (you may need to add to the configuration in~~ **~~application.properties~~** ~~to defer execution of the script).~~
4. ~~Write a resource class for the~~ **~~Course~~** ~~service. (i.e. the class with~~ *~~@RestController~~*~~). Only provide methods for the GETs and the POST.~~
5. ~~Run the service and test both GETs and the POST using Postman.~~

~~In the code files that follow, double check to make sure the data types and names match for the classes. E.g. in the~~ **~~Course~~** ~~class, I’ve used the~~ **~~long~~** ~~data type for the~~ **~~course\_id~~** ~~attribute; in the~~ **~~Student~~** ~~class I’ve used the~~ **~~int~~** ~~data type for the~~ **~~id~~** ~~attribute. It doesn’t matter what data types you use, or what you call the variables once they match. Just note that mine may be different!~~

**Part 2: Connecting the Services**

**~~Changes in the Course service for communication:~~**

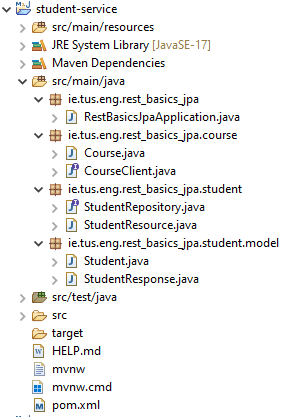
~~As both services by default run on port 8080, we need to change this. In the~~ **~~Course~~** ~~service’s~~ **~~application.properties~~** ~~file, add the line server.port=8081~~



~~Nothing else needs to be changed in the Course class.~~

**~~Changes in the Student service for communication:~~**

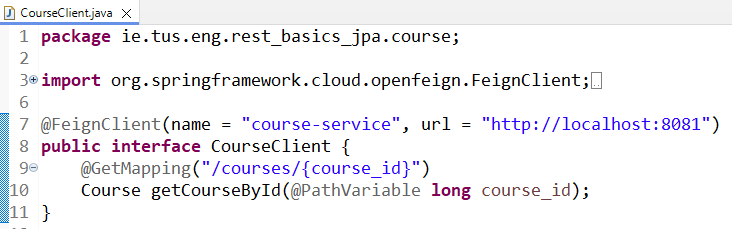
**~~The project will look as follows when we have all the necessary bits added:~~**



~~Make any of these files you don’t have; the code is provided in the following pages. Note the packages used to organise the files into a more coherent structure.~~

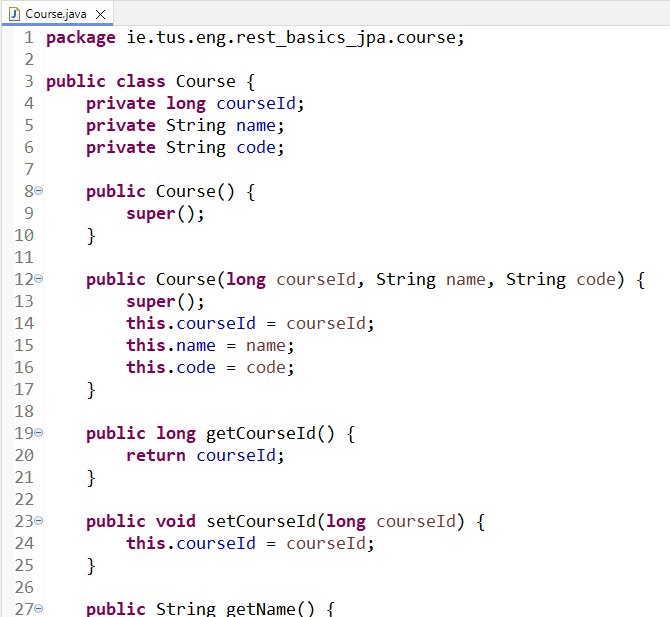
**~~FeignClient:~~**

~~We’ll use the FeignClient as a REST client to query the~~ **~~Course~~** ~~service. The FeignClient is a ‘declarative’ HTTP client in Spring Boot that simplifies making REST calls to other microservices. Feign makes use of annotations to define the interface, and Spring Boot automatically generates the implementation. Make the following file called~~ **~~CourseClient~~**~~:~~



~~The file specifies the interface with a single method –~~ **~~getCourseById()~~**~~. Spring Boot will automatically generate this method to contact the Course service on the specified port: 8081.~~

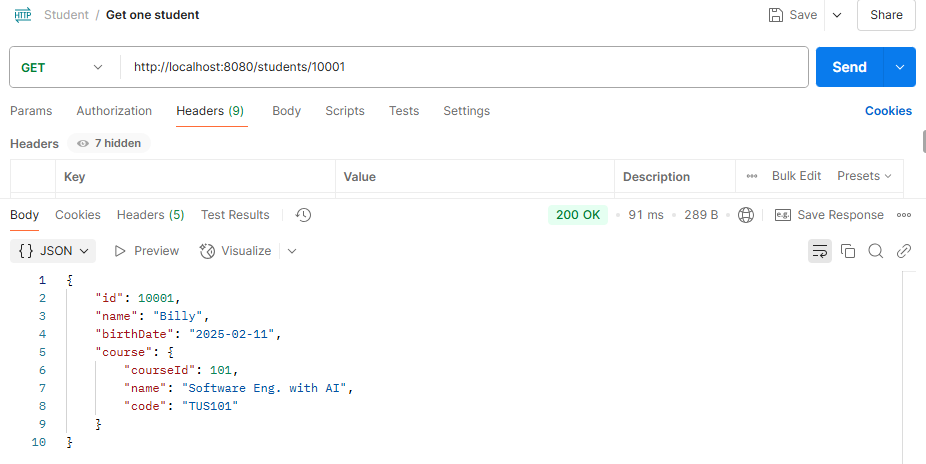
~~Note that this class uses the~~ **~~Course~~** ~~class; we have already written this in the~~ **~~Course~~** ~~service, but we’ll need to have it in the~~ **~~Student~~** ~~service too. When we create it in the~~ **~~Student~~** ~~service, make sure to remove the~~ *~~@Entity~~*~~,~~ *~~@Id~~* ~~and other JPA related tags; we just need a plain old Java object (POJO) with constructors, getters, setters, and toString methods:~~



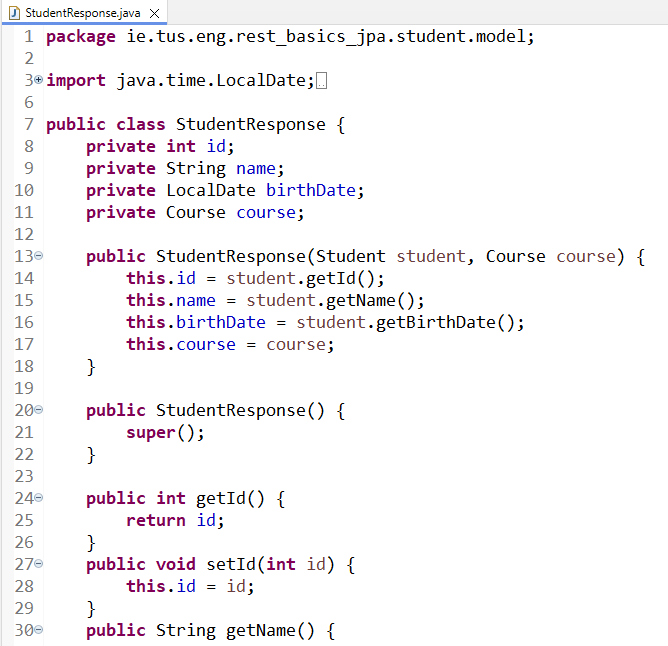
~~Finish the rest of the getters, setters and toString.~~

**~~Data Transfer Object (DTO) – StudentResponse.java~~**

~~Remember we want to return the following type of response to the user:~~



~~This is a mix of the Student and Course data. We’ll make a new object for this known as a Data Transfer Object (DTO). Its main job is to get data from the server to the client. Write the following class with the usual constructors, getters, setters and toString:~~

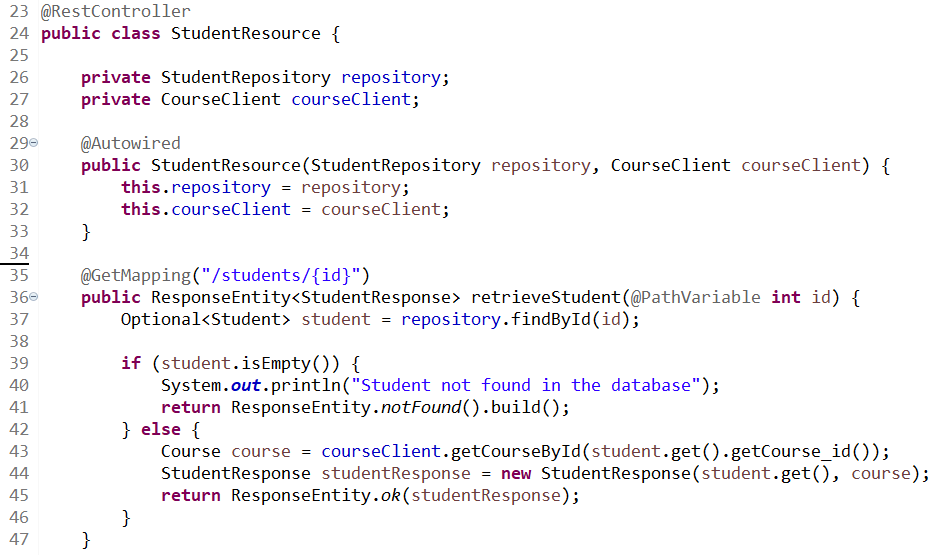


~~Provide all getters, setters, and toString.~~

~~Note we have id, name, birthdate and a Course object.~~

**~~Changing the resource (controller) file:~~**

~~Finally, we need to add the implementation for getting one student:~~



~~Note: the other @GET, @POST, @PUT etc are not shown.~~

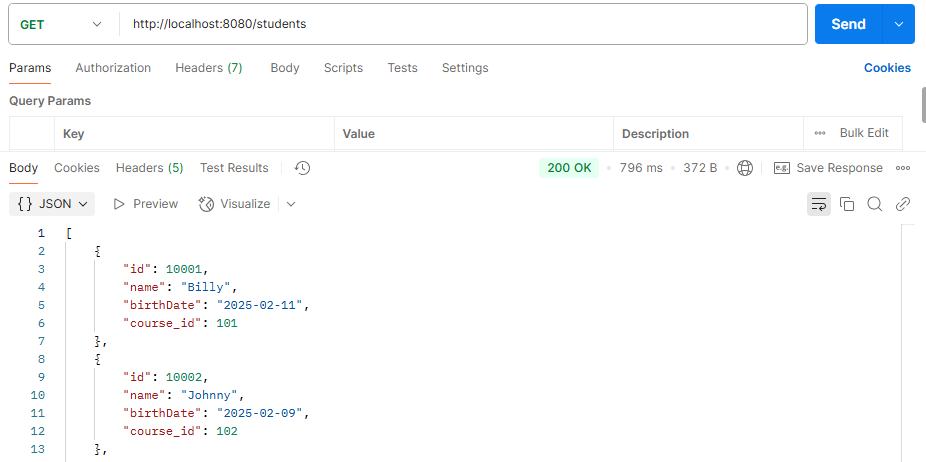
~~Referring to the above file, the main changes are as follows:~~

1. ~~The inclusion of the~~ **~~CourseClient~~** ~~(the FeignClient defined earlier).~~
2. ~~Line 43 calls the~~ **~~Course~~** ~~service and is returned a~~ **~~Course~~** ~~object.~~
3. ~~Line 44 creates a new~~ **~~StudentResponse~~** ~~object.~~
4. ~~The~~ **~~StudentResponse~~** ~~object is returned.~~

**~~Part 3: Running and testing the Services~~**

~~Run both services and then test to see if you can get the following screenshots. The same Postman tests as before should work. Debug as necessary!~~

**~~Getting all students:~~**



**~~Getting one student:~~**

