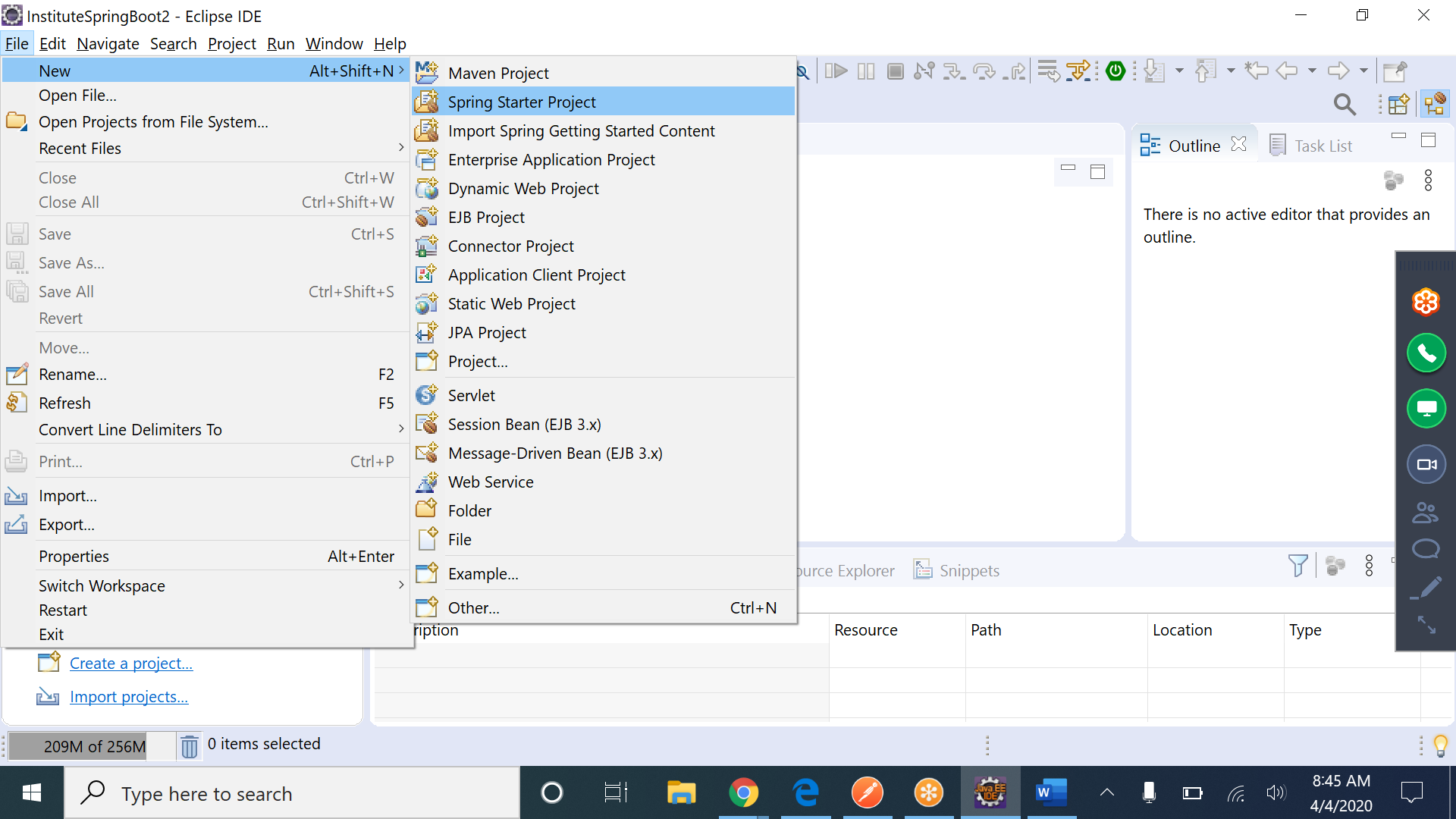
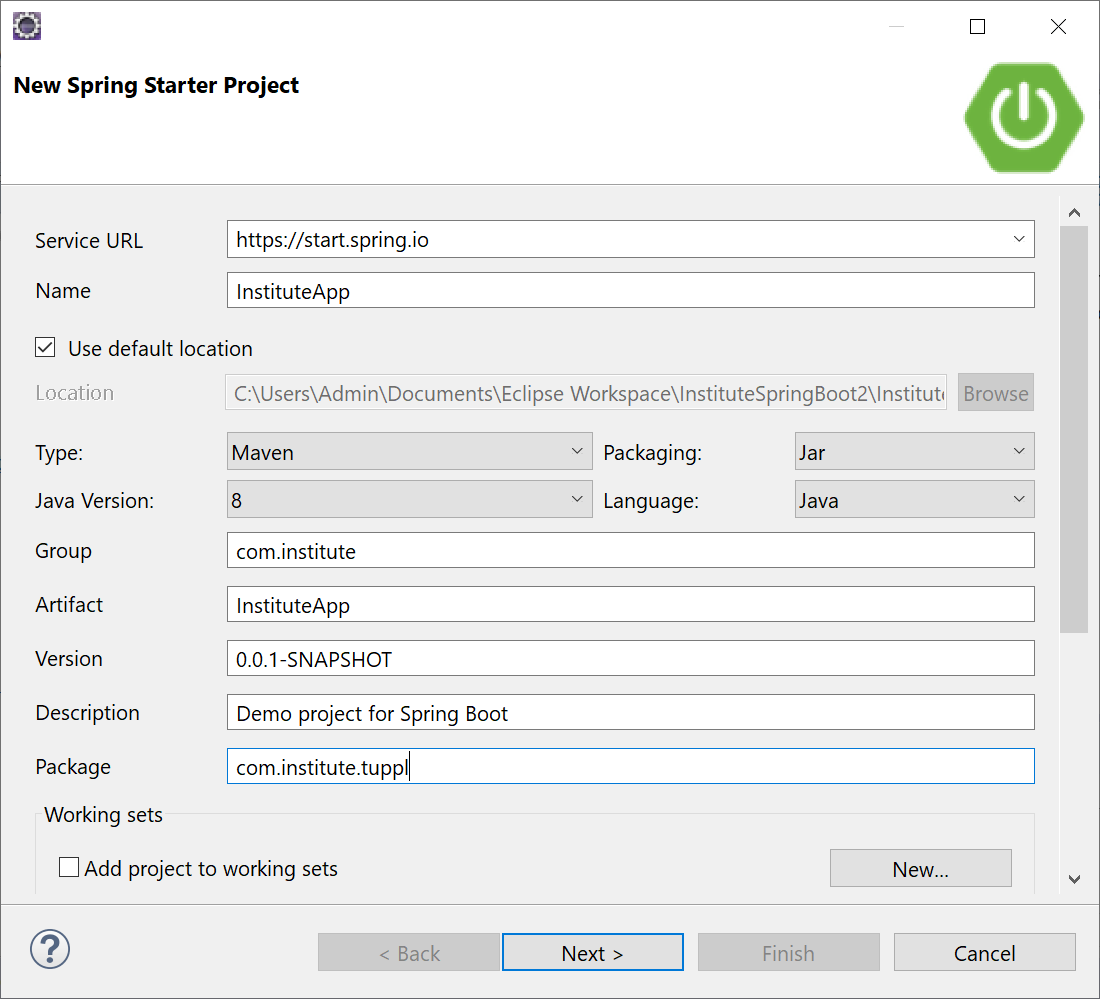
# Setup SpringBoot Workspace

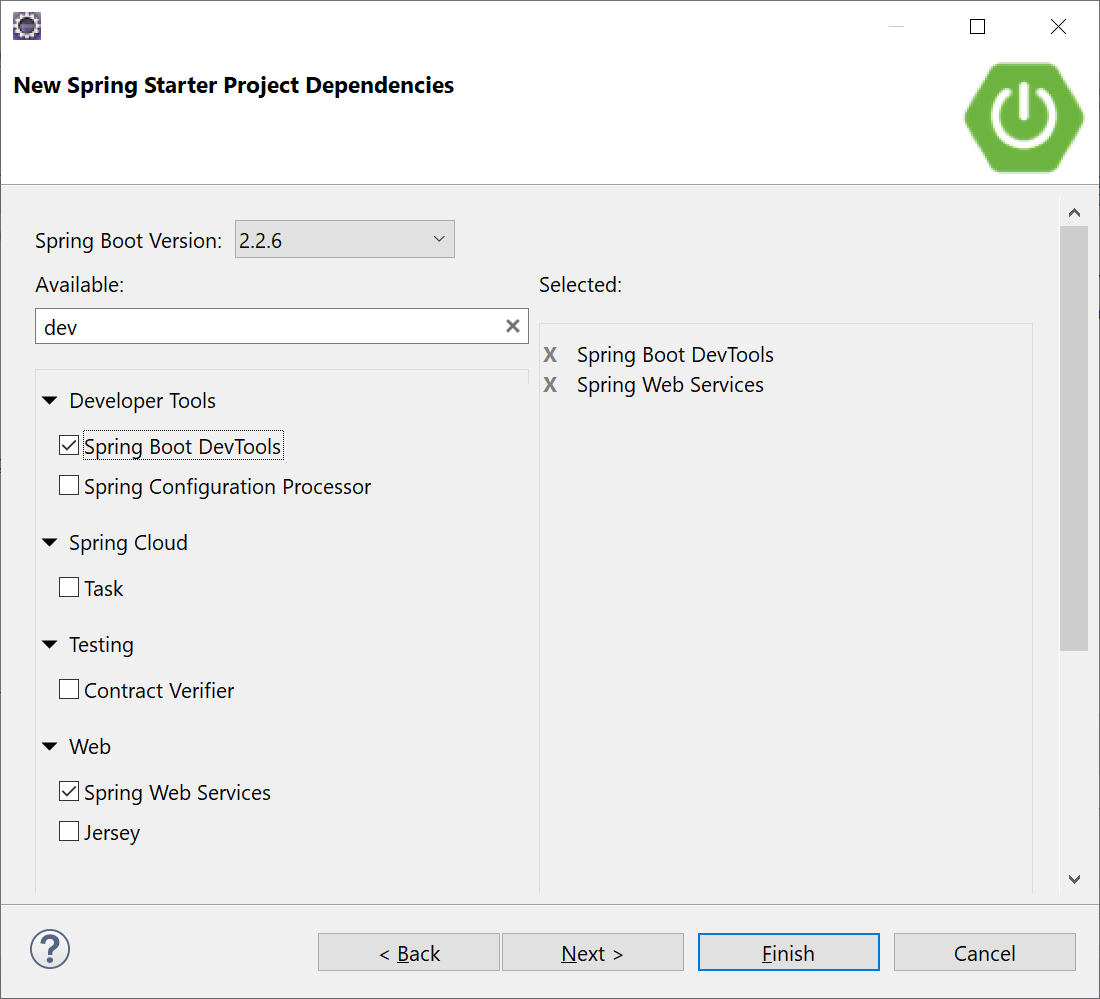
1. Create New Workspace.
2. Open Ecplise.
3. Click File -> New -> Spring Started Project.



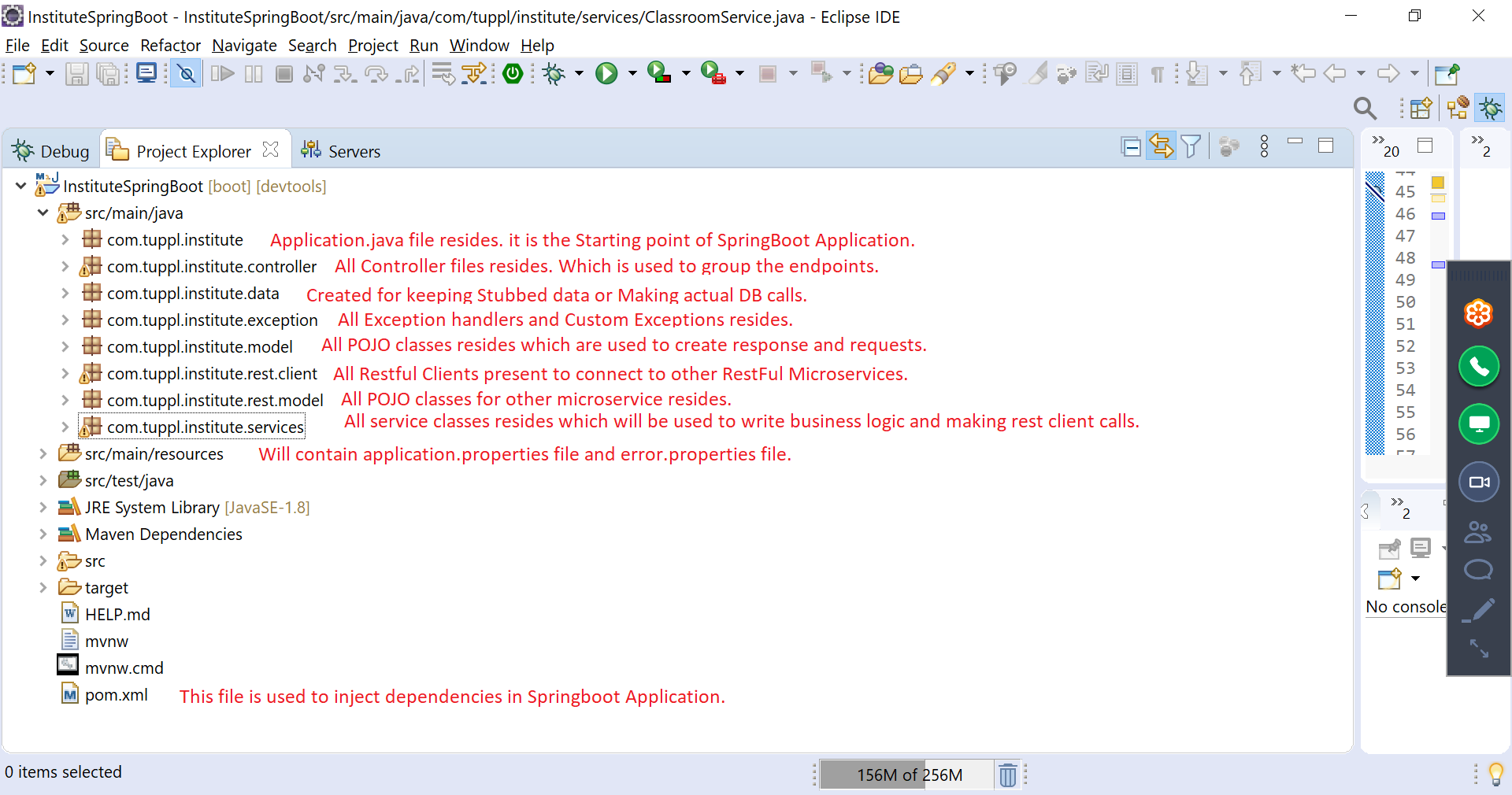
1. Enter Name, Group and Package and click Next



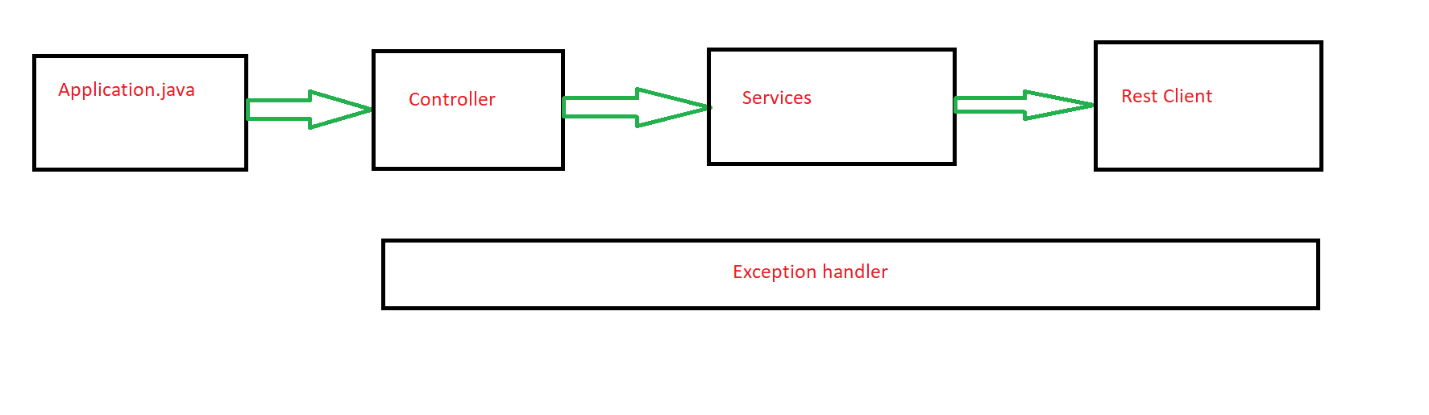
1. Select Dependencies like “Spring boot Devtools” and “Spring Web Services” and click Finish



# SpringBoot File Structure



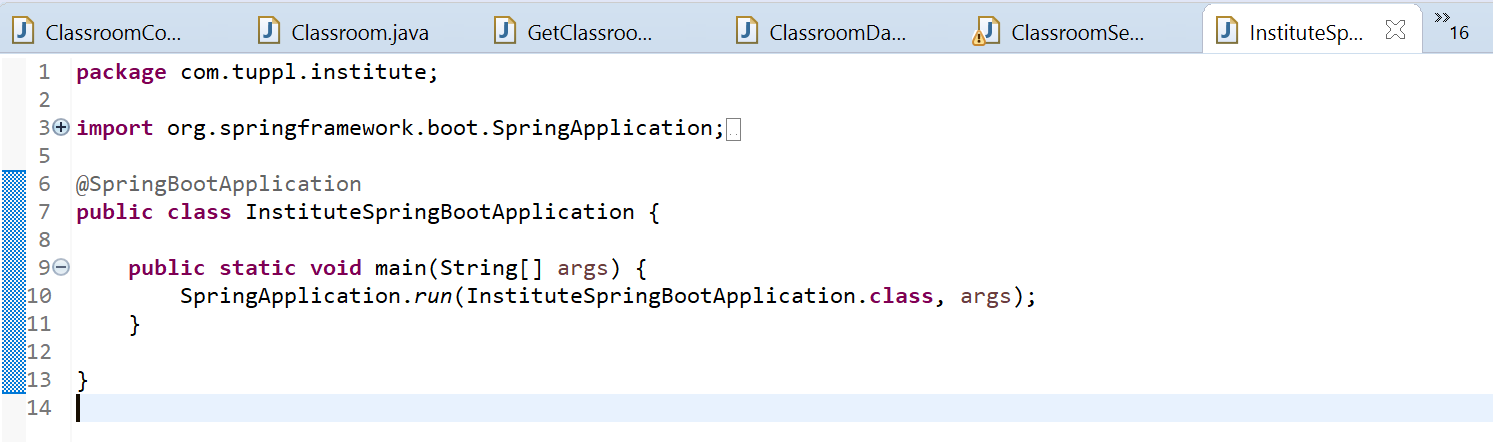
# Endpoint call Structure



# Files Explanation

## Application.java

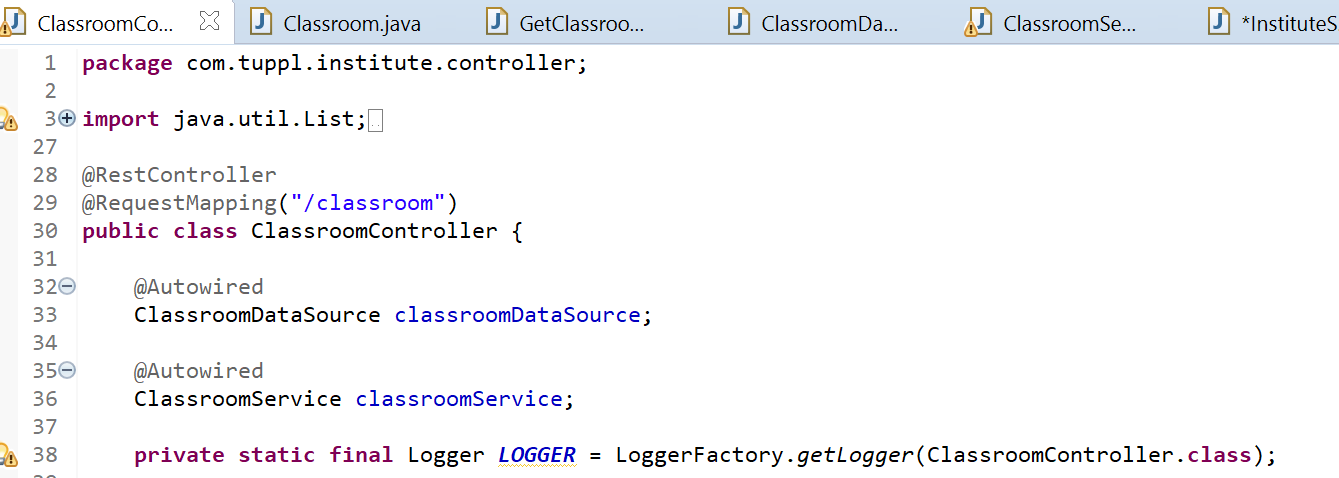
As Mentioned above, it’s the starting point of the application meaning all calls to this microservice will come to this file. We denote starting point with @SpringBootApplication annotation.



## Controller File

This will be used to group similar endpoints. We denote the class with @RestController annotation. If we add @RequestMapping("/classroom") annotation on class level then all endpoints will start from that url.

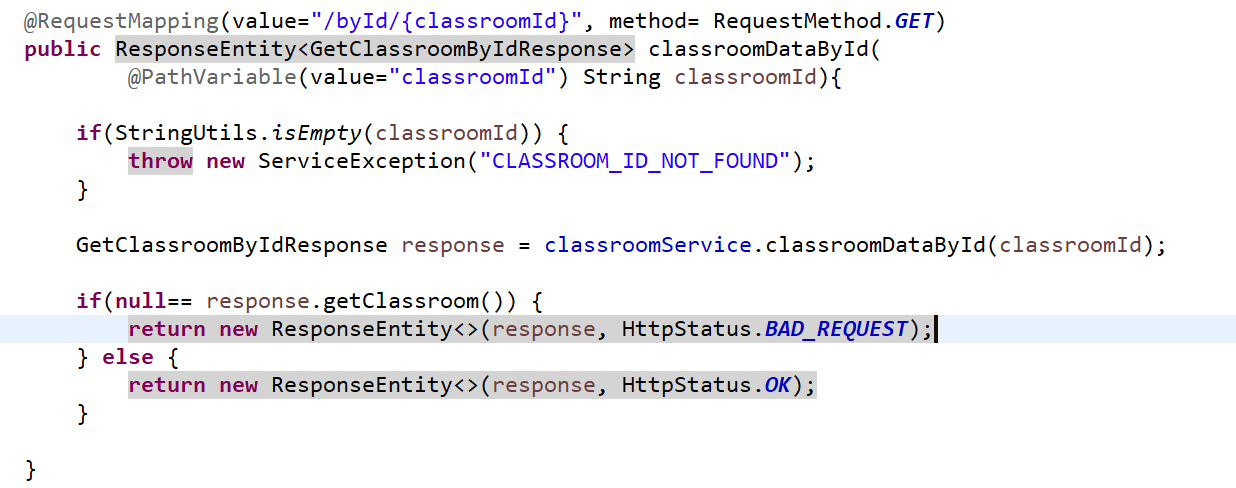
@Autowired annotation is used to inject dependency for other classes in a particular class.



Every Endpoint is defined as :

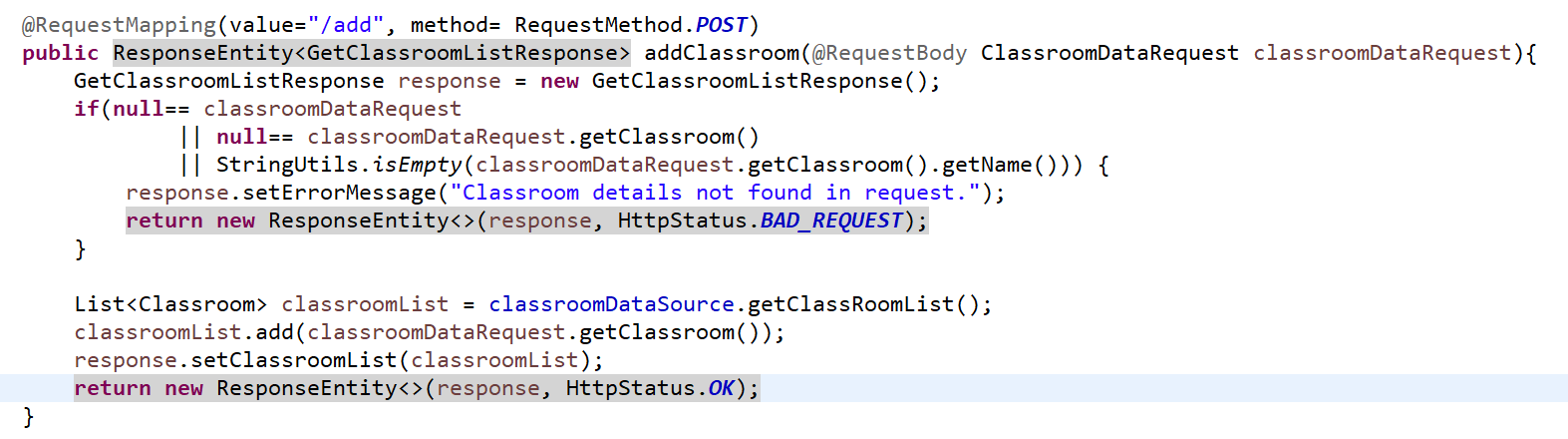
1. As a GET Method:

We denote endpoint url with @RequestMapping annotation which will have value as the url and method as GET.



1. As a POST Method:

We denote endpoint url with @RequestMapping annotation which will have value as the url and method as POST. POST request will always have @RequestBody.

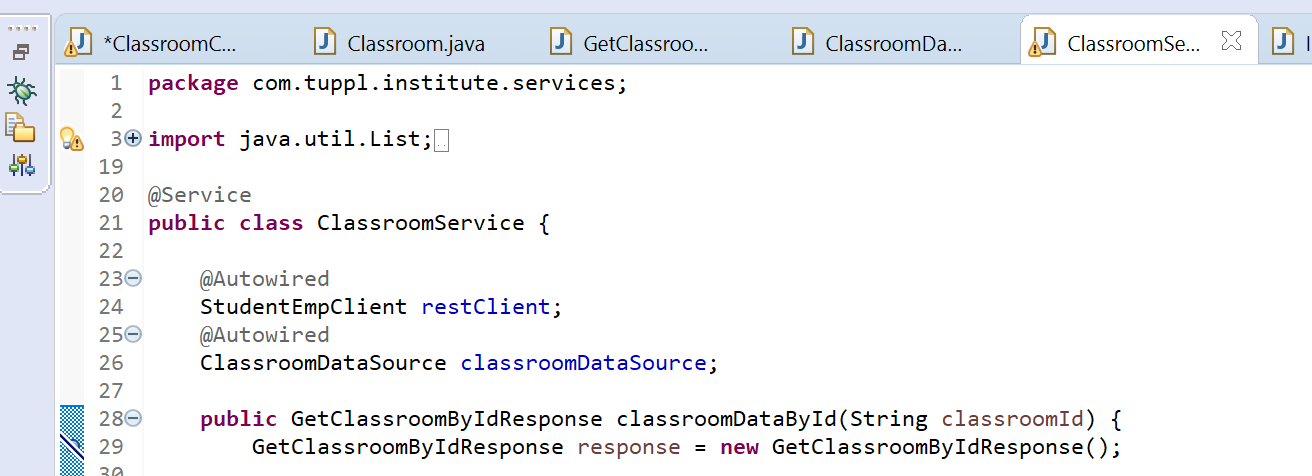


Notes:

1. We can use other methods as well like DELETE and PUT.
2. @PathVariable annotation will be used to provide dynamic value in an Endpoint url.
3. Response class is mapped inside ResponseEntity<> class to return response body, response status, headers and cookies.

## Service Class

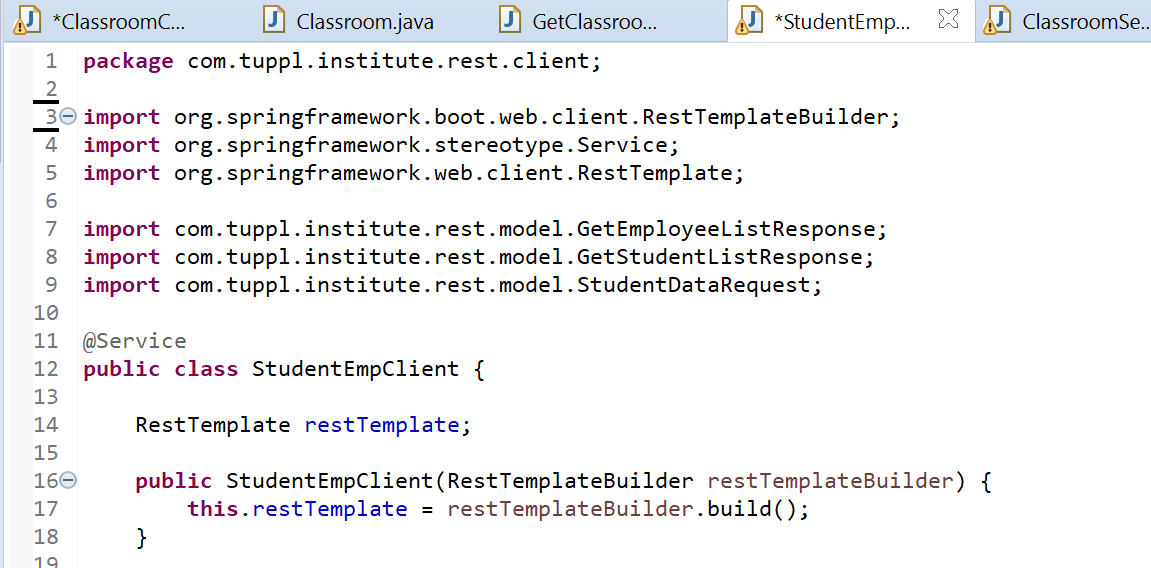
This file will be used to make Backend(or other microservices or connecting DB) calls and writing Business logic. We denote a service class with @Service annotation.



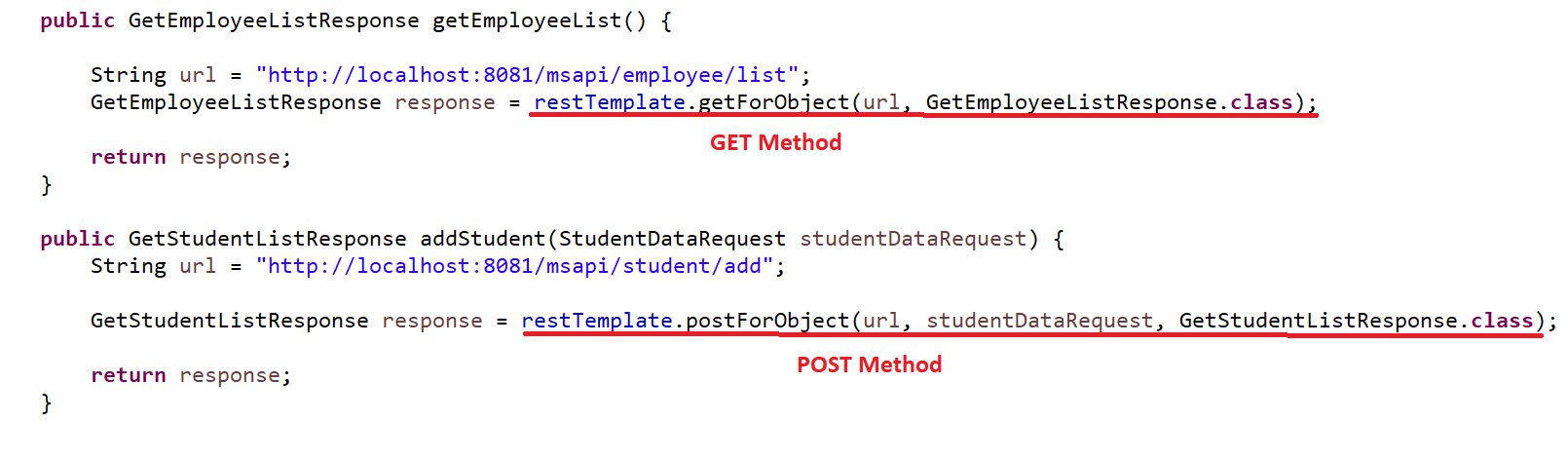
## Rest Client Class

Every rest client will be created for every Microservice and separate methods inside it will be used to call different endpoints within that Microservice.

To make rest calls, we need to create class level object for org.springframework.web.client.RestTemplate class and initialize it in constructor of the Rest Client class using org.springframework.boot.web.client.RestTemplateBuilder.



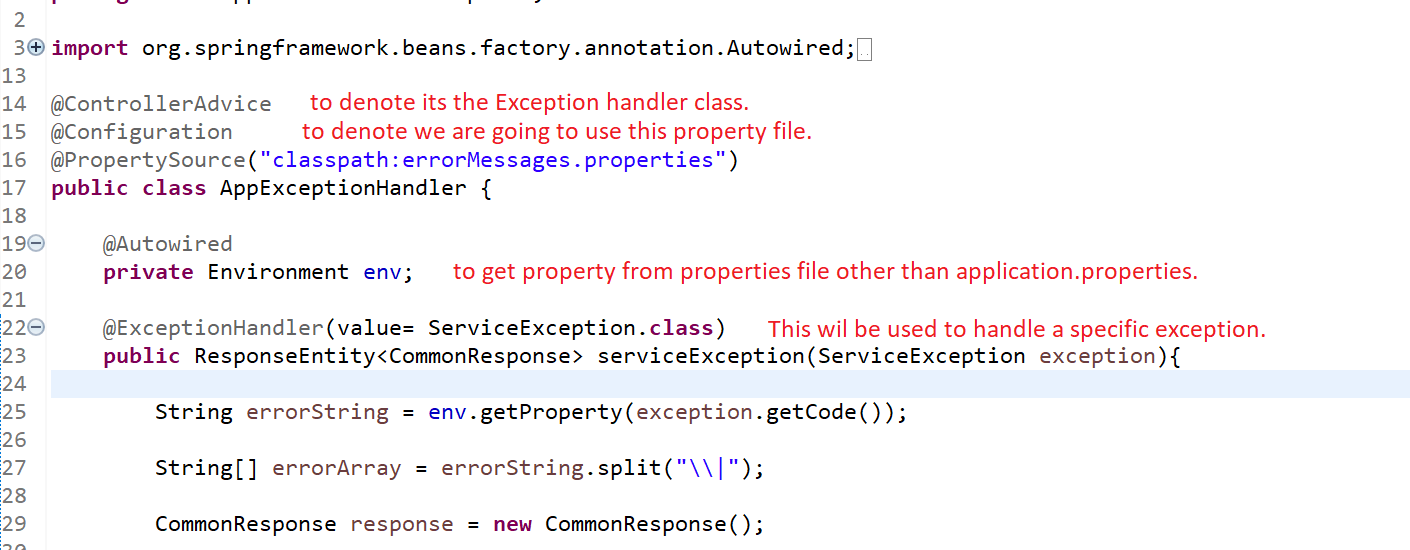
Below is the code to be used to make REST Calls:



## Exception handler Class

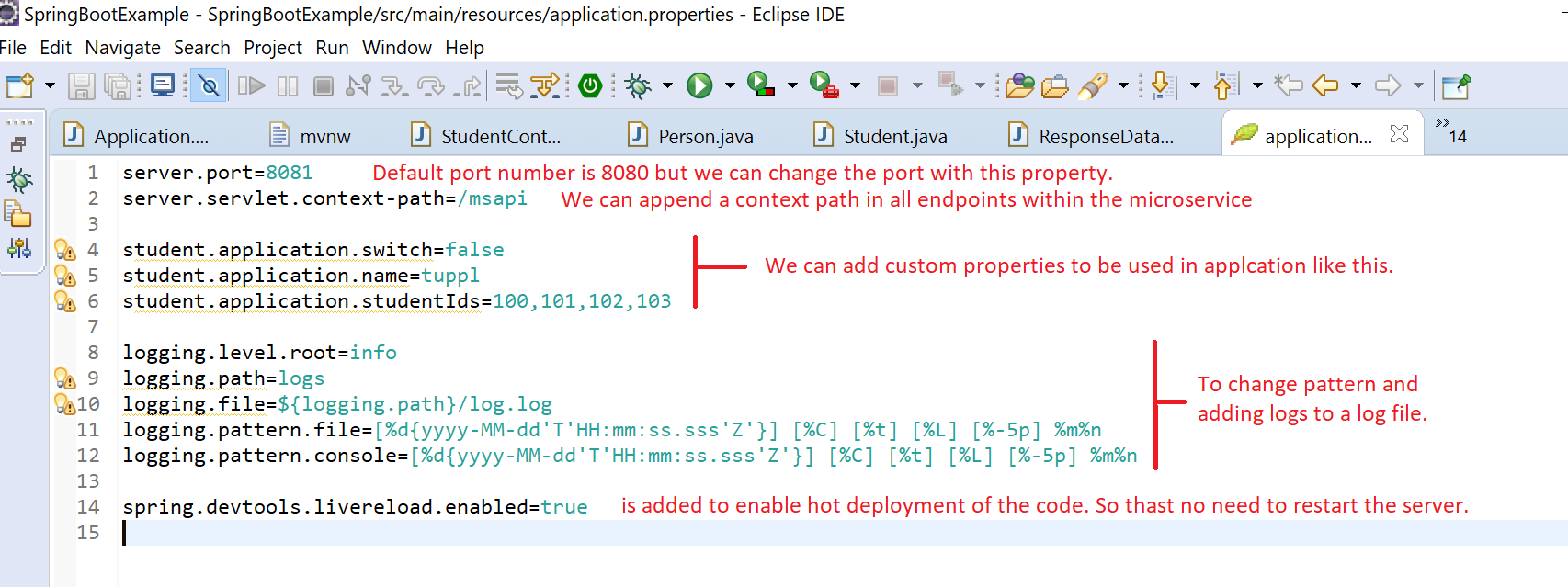
This class will be used to handle all exceptions within the Microservice. And it is denoted by @ControllerAdvice annotation. We can only create One Class like this in whole Microservice.

We can handle every exception via different method.

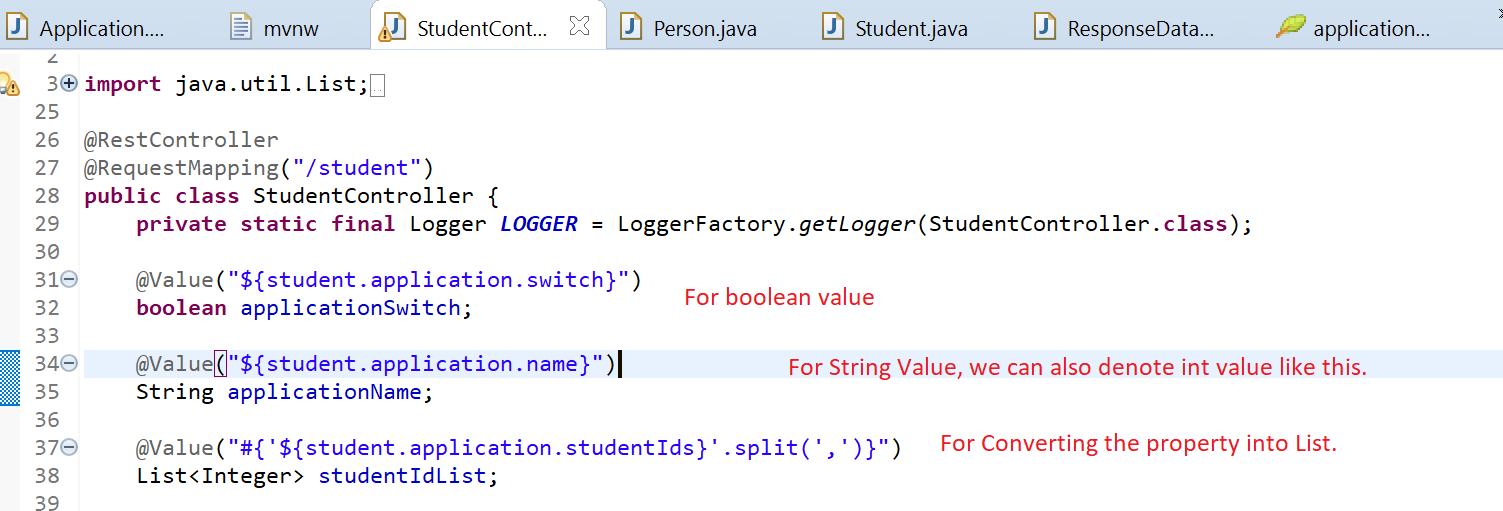


## Application.properties file

This file will be used to add properties which will be used by the microservice.

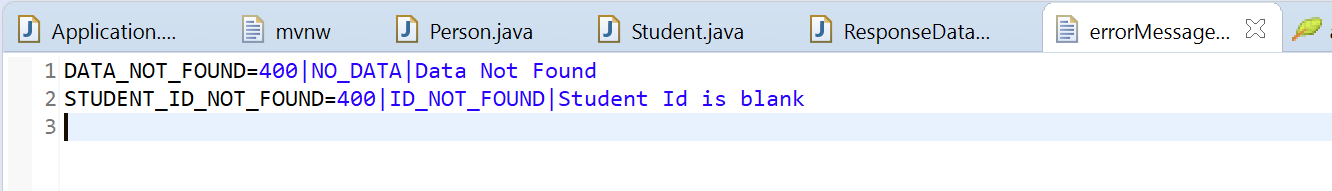


We can use properties in Class like this:



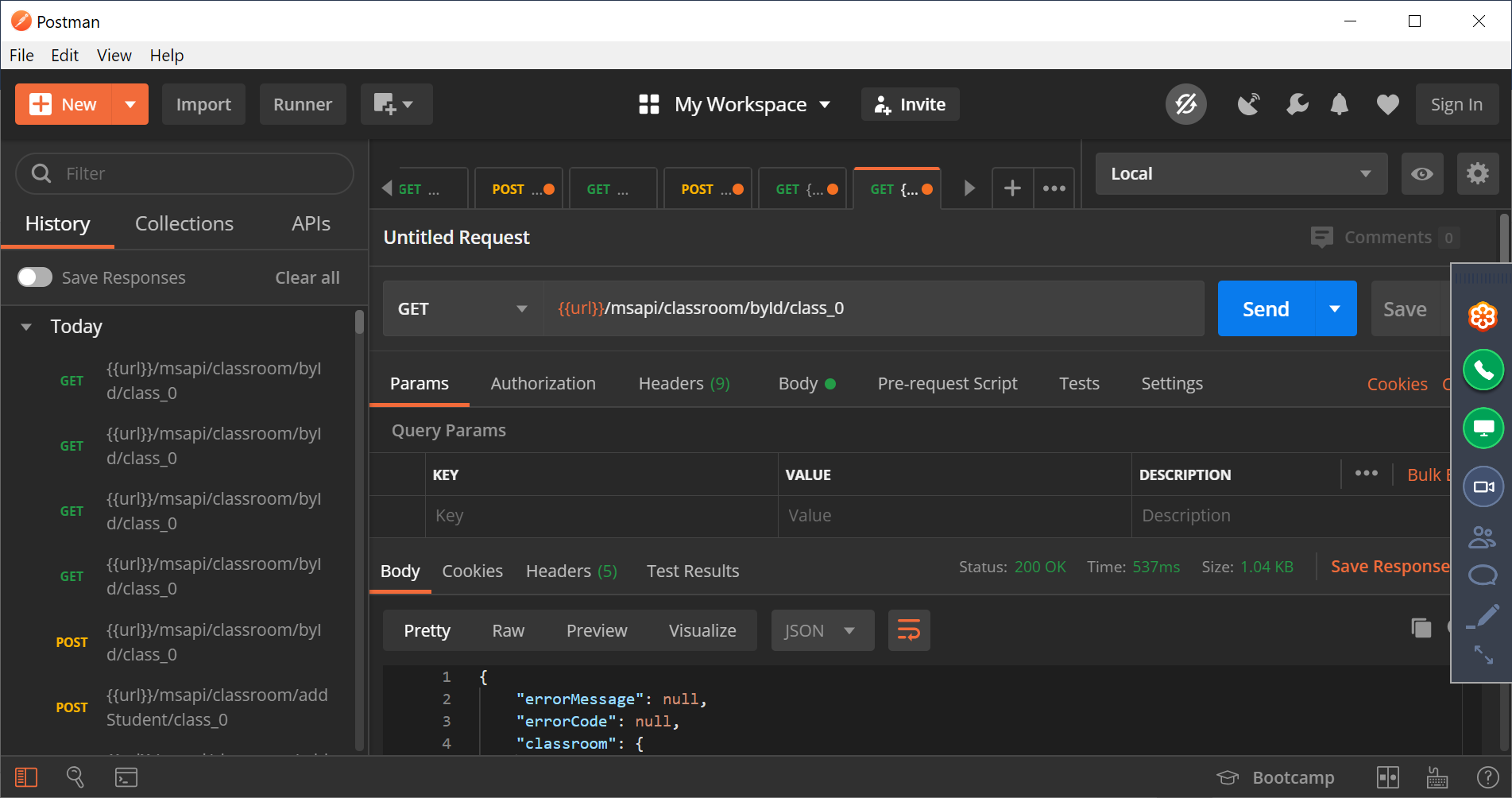
## ErrorMessages.properties file

This will be used to add error codes and their details.

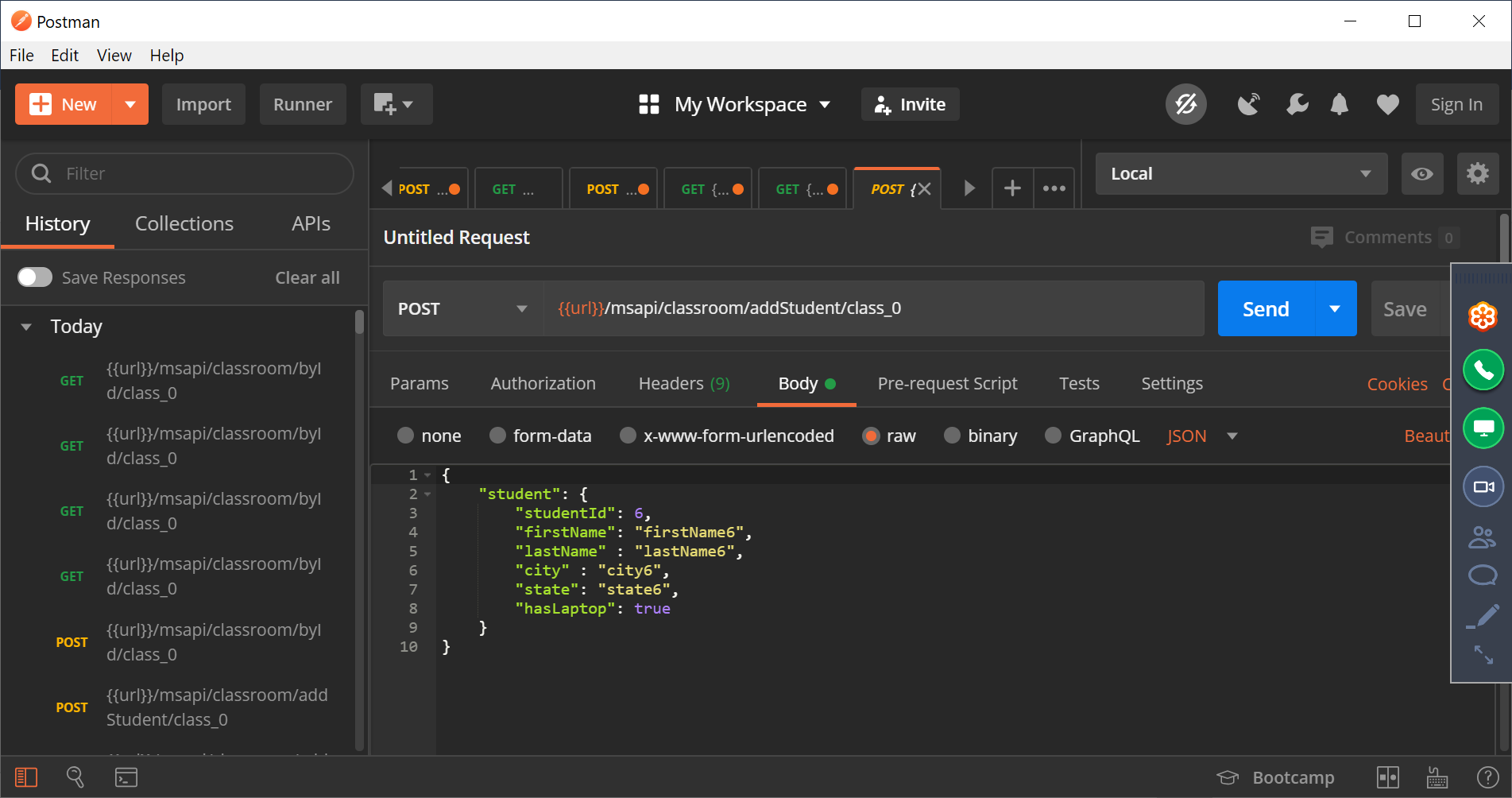


# POSTMAN

Example of GET:



Example of POST:



# **Integration with MySQL**

**Step1:** Add below dependency in Pom.xml and take Maven force update:

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>    <dependency>  <groupId>org.hibernate.javax.persistence</groupId>  <artifactId>hibernate-jpa-2.0-api</artifactId>  <version>1.0.1.Final</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <scope>runtime</scope>  </dependency> |

**Step2:** In MySQL, create Database and table:

|  |
| --- |
| mySql> Create Database instituteDatabase;  mySql> Use instituteDatabase;  mySql> CREATE TABLE Student\_tb (student\_id int NOT NULL,first\_name varchar(255),last\_name varchar(255),address varchar(255),zip\_code int,class\_id varchar(255),has\_laptop BOOLEAN,PRIMARY KEY (student\_id)); |

**Step3:** Add MySql DB connection details in application.properties:

|  |
| --- |
| spring.jpa.hibernate.ddl-auto=none  spring.datasource.url=jdbc:mysql://${MYSQL\_HOST:localhost}:3306/instituteDatabase  #For Mac  #jdbc.url=jdbc:mysql://localhost/instituteDatabase?characterEncoding=utf8&useUnicode=true&sessionVariables=storage\_engine%3DInnoDB&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC  spring.datasource.username=root  spring.datasource.password=root  spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver |

**Step4:** Add @EnableJpaRepositories in application.java class.

|  |
| --- |
| @SpringBootApplication  @EnableJpaRepositories  **public** **class** Application {  **public** **static** **void** main(String[] args) { |

**Step5:** Create Java Class which will map row in a Table with column name as attributes and getter/setters:

|  |
| --- |
| @Entity  **public** **class** StudentTb {    **public** StudentTb() {}    @Id  **private** **int** studentId;    **private** String firstName;    **private** String lastName; |

**Step6:** Create Repository Interface class to perform operations on table:

|  |
| --- |
| **public** **interface** StudentRepository **extends** CrudRepository<StudentTb, Integer>{    @Query(value="SELECT \* FROM StudentTb WHERE hasLaptop = true", nativeQuery = **true**)  List<StudentTb> findByHasLaptop();    @Query(value = "SELECT \* FROM StudentTb", nativeQuery = **true**)  List<StudentTb> findAllStudents();    } |

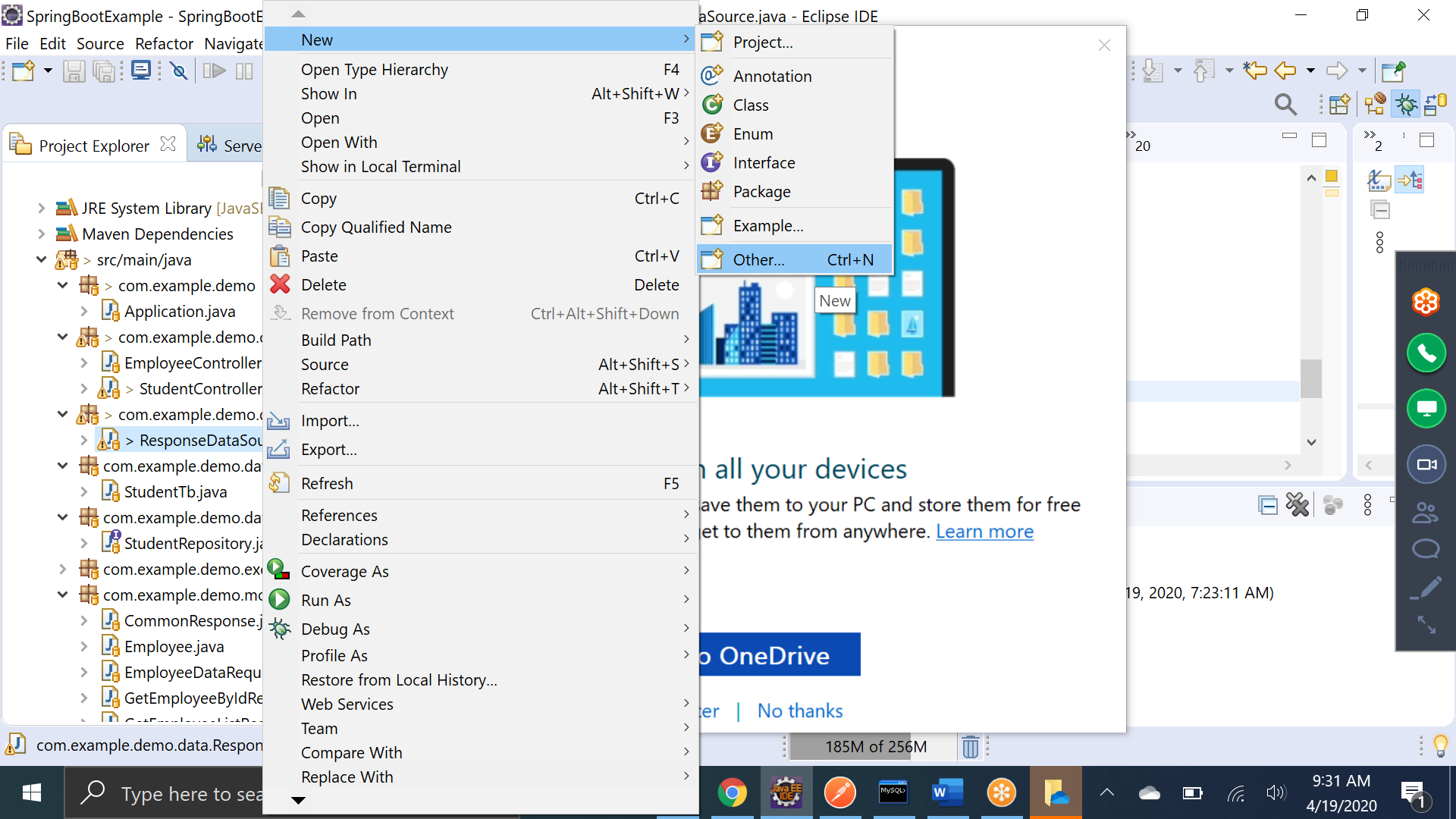
Note: If want to create special queries then use @Query annotation like above else common operations are already available.

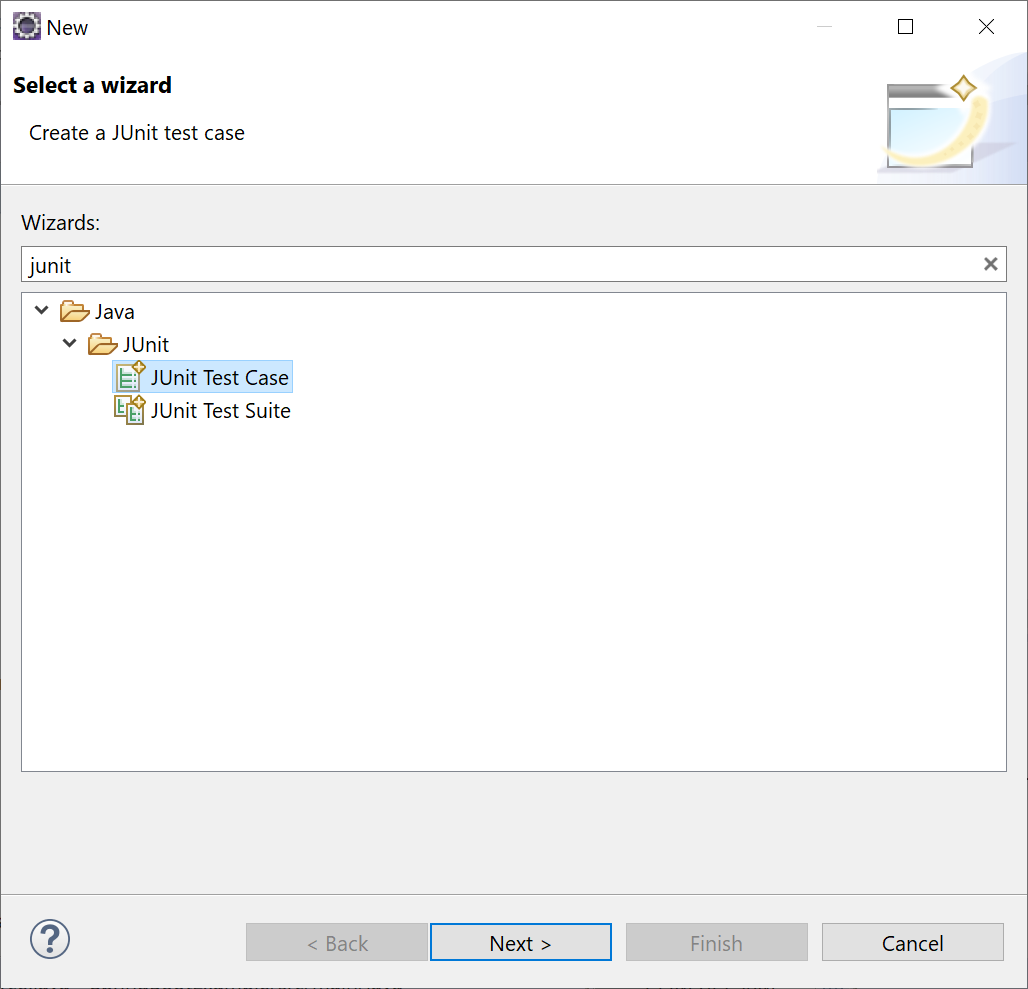
**Step7:** Use Repository class in code to perform work:

|  |
| --- |
| @Service  **public** **class** StudentService {  @Autowired  StudentRepository studentRepo;    **public** Iterable<StudentTb> getAllStudents() {  **return** studentRepo.findAll();  }    **public** **void** addStudent(StudentTb studentTb) {  studentRepo.save(studentTb);  }    **public** Optional<StudentTb> findById(Integer id) {  **return** studentRepo.findById(id);  }  } |

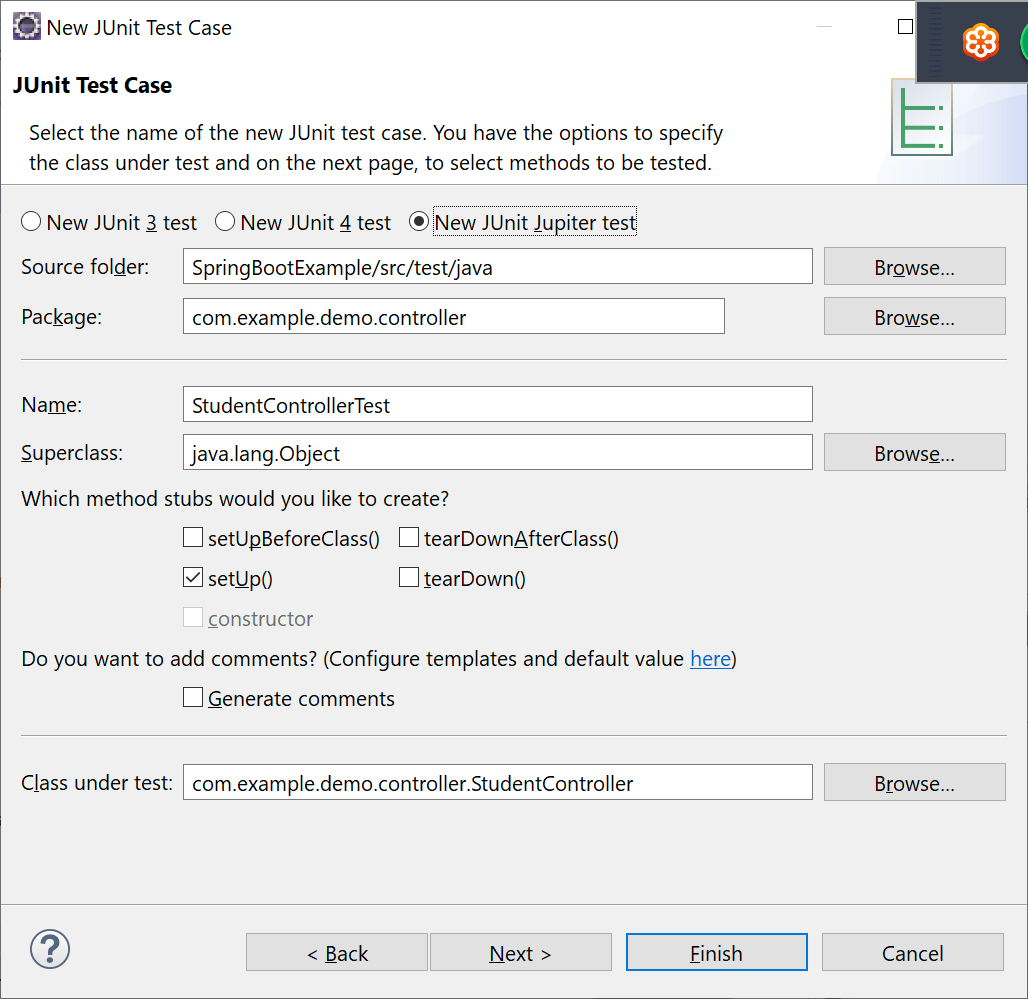
# Writing JUNIT test cases

Step 1: Right click on the class for which you want to write JUNIT and select New -> other



Step2: Select “JUNIT test case” and click Next()

Step3: checkbox setUp() and click Finish



Step 4: add @RunWith(SpringRunner.**class**) annotation on the top of the test class and create object of class which needs to be tested with @InjectMocks annotation. Also, inject dependency of all the autowired classes with @Mock annotation.

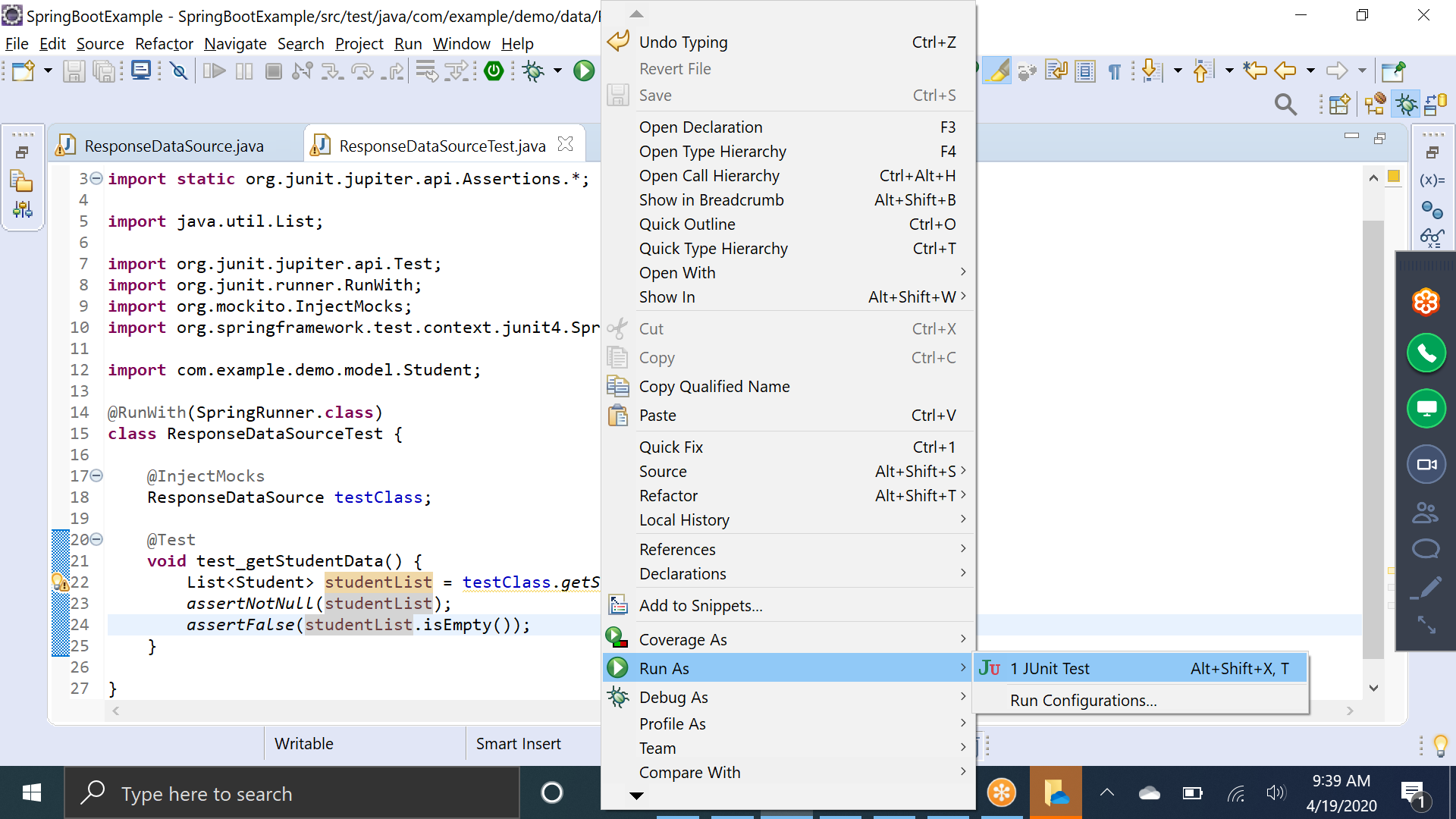
|  |
| --- |
| @RunWith(SpringRunner.**class**)  **class** ResponseDataSourceTest {    @InjectMocks  ResponseDataSource testClass;    @Mock  StudentRepository studentRepo;  /\*\*  \* **@throws** java.lang.Exception  \*/  @BeforeEach  **void** setUp() **throws** Exception {  // to initialize all the mock values  MockitoAnnotations.*initMocks*(**this**);    } |

Step5: Add the test case:

|  |
| --- |
| @Test  **void** test\_getStudentData() {  List<Student> studentList = testClass.*getStudentData*();  *assertNotNull*(studentList);  *assertFalse*(studentList.isEmpty());  } |

|  |
| --- |
| @Test  **void** test\_getStudentDataFromDB\_empty() {  List<StudentTb> studentTbList = **new** ArrayList<>();    Iterable<StudentTb> iterable = () -> **new** Iterator<StudentTb>() {  **private** **int** index = 0;  @Override  **public** **boolean** hasNext() {  **return** studentTbList.size() > index;  }  @Override  **public** StudentTb next() {  **return** studentTbList.get(index++);  }  };  Mockito.*when*(studentRepo.findAll()).thenReturn(iterable);  List<Student> studentList = testClass.getStudentDataFromDB();  *assertNotNull*(studentList);  *assertTrue*(studentList.isEmpty());  } |

Step6: Run the test case by right clicking on the test class then run As -> JUnit Test



Step7: To debug the test class, right click -> Debug As -> Junit Test.

Step8: Check the code coverage, right click -> Coverage As -> Junit test

