

# Dependency Injection and Spring RESTful API

## License



This work by Satish Kumar at Leeds Beckett University is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

## Contents

### [License](#)

### [Contents](#)

### [Introduction](#)

### [Spring Boot Application Architecture](#)

### [Creating Student-demo microservice](#)

### [Dependency Injection with Student-demo microservice](#)

### [Student API Testing with Postman](#)

### [Conclusion](#)

## Introduction

The aim of this week's lab is to gain an in-depth understanding of Dependency Injection. Through this lab, we aim to equip you with the knowledge of how to apply Dependency Injection effectively for creating loosely coupled RESTful API.

## Spring Boot Application Architecture

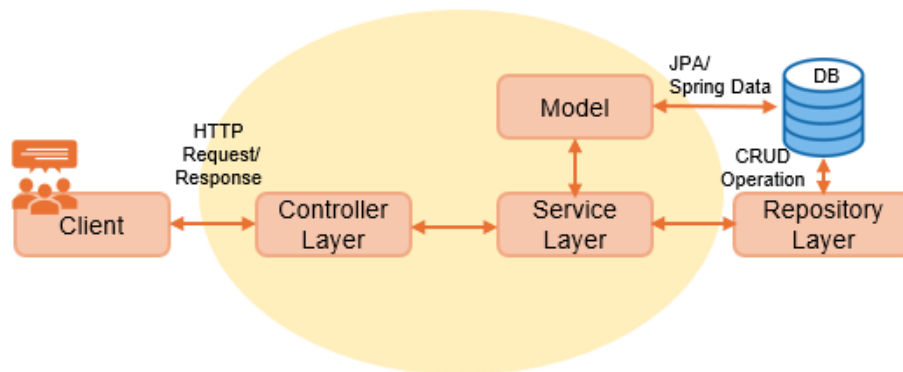


Figure 1: Spring Boot Application Architecture

The **Controller Layer** is responsible for handling incoming HTTP requests such as GET, POST, DELETE, and UPDATE. It receives these requests and interacts with the Service Layer to process them and returning the appropriate response back to the client.

**Server Layer:** This layer contains all the business logic specific to the application's use case. It facilitates communication between the Controller Layer and the Repository Layer (database), helping to organize, encapsulate, and manage business rules in a modular way.

A **Domain Model** in a Spring Boot application represents the real-world entities and their relationships within the application's business logic. It defines how data is structured, stored, and manipulated in the system.

The **Repository Layer** manages data access and is responsible for performing Create, Read, Update, and Delete (CRUD) operations, as well as executing custom queries by interacting with the database. Its main function is to simplify data persistence, allowing the Service Layer to retrieve, save, update, or delete data without directly dealing with the complexities of database operations.

## Student-demo microservice implementation

We are using spring initializr for creating a starter student-demo microservice and adding the following dependencies and clicking on the generate button to download this project. You can access spring initializr at <https://start.spring.io/>

- Spring Web
- Lombok

Project

☐ Gradle - Groovy
 ☐ Gradle - Kotlin
 ☒ Java
 ☐ Kotlin
 ☐ Groovy

Spring Boot

☐ 3.5.0 (SNAPSHOT)
 ☐ 3.5.0 (M1)
 ☐ 3.4.3 (SNAPSHOT)
 ☒ 3.4.2
 ☐ 3.3.9 (SNAPSHOT)
 ☐ 3.3.8

Project Metadata

Group

com.sesc

Artifact

student-service

Name

student-service

Description

Demo project for Spring Boot

Package name

com.sesc.unistudycircle.student\_service

Packaging

☒ Jar
 ☐ War

Java

☐ 23
 ☒ 21
 ☐ 17

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Lombok

DEVELOPER TOOLS

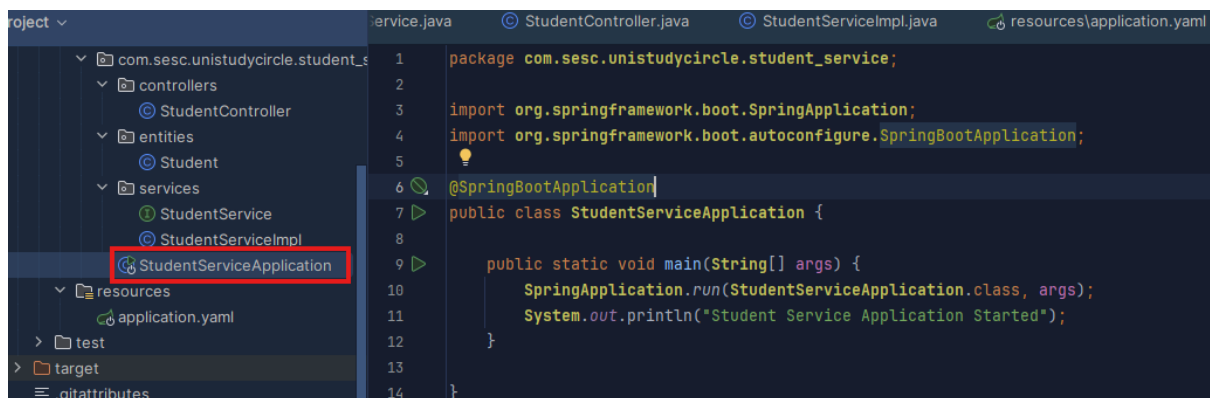
Java annotation library which helps to reduce boilerplate code.

GENERATE CTRL + G

EXPLORE CTRL + SPACE

...

The next step is to unzip student-demo project and save it on network P Drive for further use. Now, open this project with IntelliJ IDEA and explore the project structure. The **StudentServiceApplication** is the entry point class for executing student-demo service project.



```

1 package com.sesc.unistudycircle.student_service;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class StudentServiceApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(StudentServiceApplication.class, args);
11         System.out.println("Student Service Application Started");
12     }
13 }
  
```

## Spring Boot Annotation

Spring Boot annotations are metadata that provide instructions to the Spring framework about how to handle various components in a Spring Boot application. These annotations simplify configuration and help with dependency injection, bean management, and various other functionalities.

## @SpringBootApplication

The @SpringBootApplication annotation is a core of Spring Boot application that is designed to simplify the auto-configuration, component scan and be able to define extra configuration on the application class. It is a **meta-annotation**, meaning it combines multiple annotations as follows

- **@EnableAutoConfiguration**: enable Spring Boot's auto-configuration mechanism
- **@ComponentScan**: enable @Component scan on the package where the application is located.
- **@SpringBootConfiguration**: enable registration of extra beans in the context or the import of additional configuration classes. An alternative to Spring's standard @Configuration that aids configuration detection in your integration tests

## Domain Model for Student-demo application

The **Student** entity model represents the student's information including personal details, qualifications, university etc.

Now we are ready to create our first domain class. Let's start by creating a simplified version of Student.

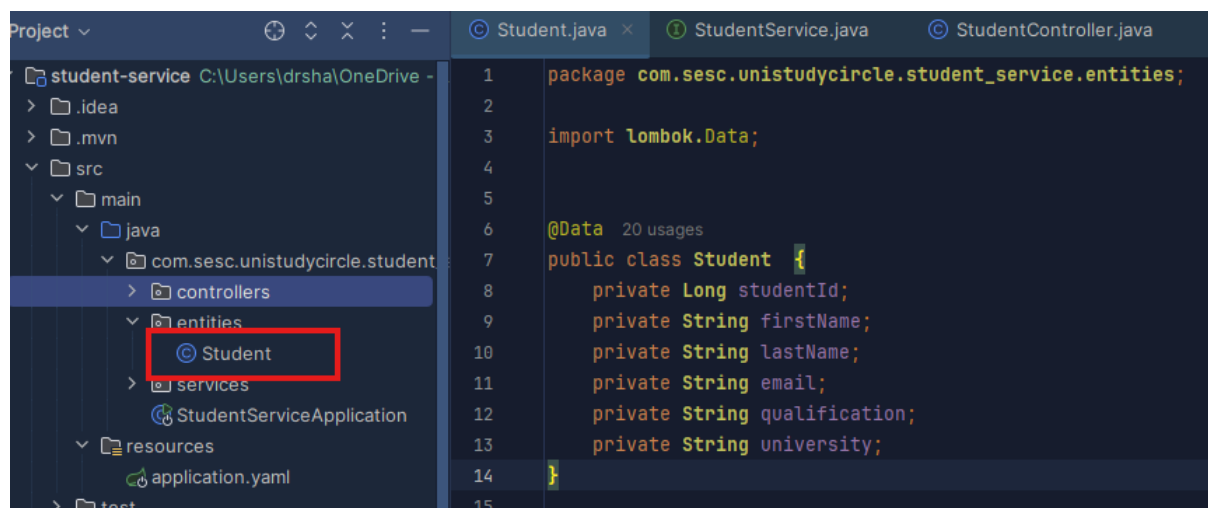
In your src/main/java directory, create a package called:

*com.sesc.unistudycircle.student\_service.entities*

This is where our domain model classes will live.

Create a class called Student in the package  
*com.sesc.unistudycircle.student\_service.entities*

Double click on the **Student** class (left side window) and write the following properties in the class and add @Data dependency on the class.



The **@Data** annotation is there for convenience. It uses the Lombok library to generate getters and setters for each field, as well as equals() and hashCode() implementations. If you prefer to write these by hand, you can do that instead and remove the Lombok dependency from your pom.xml file. It is a matter of personal choice, we think it looks much cleaner with the annotation

The next step is to implement the business logic for creating, storing and retrieving students information. Let's create interfaces and class for implementing students business logic.

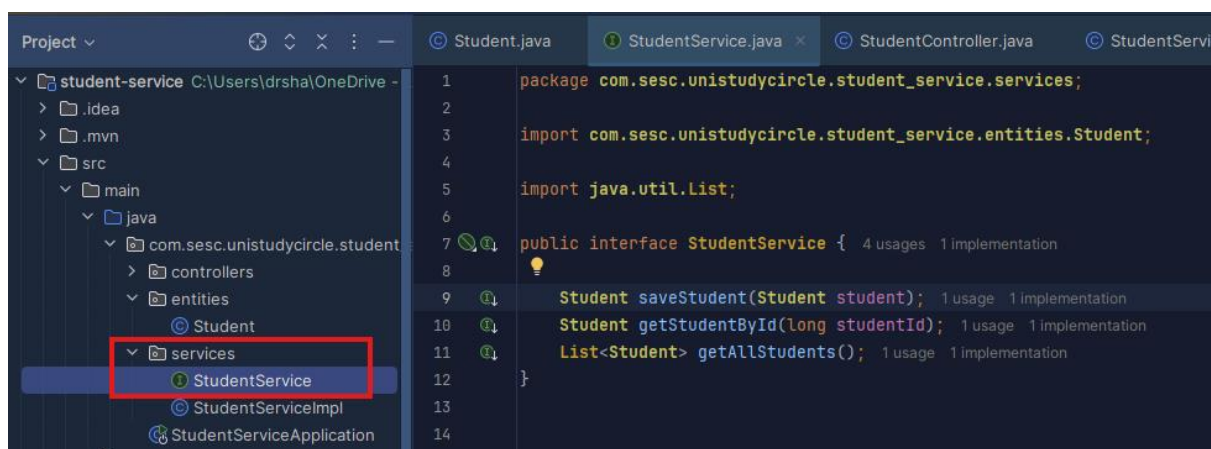
In your src/main/java directory, create a package called:

*com.sesc.unistudycircle.student\_service.services*

This is where our service classes and interface will live.

Create a interface called **StudentService** in the package *com.sesc.unistudycircle.student\_service.services*

Let's double click on the **StudentService** interface and declare following methods that will be implemented by another class to process relevant functionalities.



We now create a StudentServiceImpl class for implementing StudentService interface. As we know interface requirements, we will implement all the methods declared in the StudentService interface. For the sake of simplicity, we are using List data structure as a temporary storage (in-memory storage) for storing student details.

Create a class called StudentServiceImpl in the package *com.sesc.unistudycircle.service*

Double click on the **StudentServiceImpl** class (left-side window) and add @Service dependency on this class and implement StudentService interface as shown below,

```
9  @Service
10 public class StudentServiceImpl implements StudentService {
11
12
13  List<Student> students = new ArrayList<>(); 3 usages
```

We are now ready to implement following methods declared in the StudentService interface.

1. **saveStudent(Student student)** method: This method accepts a Student object (student details such as studentId, firstName, lastName etc.) and adds this Student object to the List, and returns the same object.
2. **getStudentById(String studentId)** : This method accepts a studentId as a String parameter and returns the Student object if the same studentId exists in the List.
3. **getAllStudents()** : This method returns all student records that exist in the list.

```
2  import com.sesc.unistudycircle.student_service.entities.Student;
3  import org.springframework.stereotype.Service;
4
5  import java.util.ArrayList;
6  import java.util.List;
7
8  @Service
9  public class StudentServiceImpl implements StudentService {
10
11      List<Student> students = new ArrayList<Student>(); 3 usages
12
13      @Override 1 usage
14      public Student saveStudent(Student student) {
15          students.add(student);
16          return student;
17      }
18
19      @Override 1 usage
20      public Student getStudentById(long studentId) {
21          return students.stream().filter(student -> student
22              .getStudentId().equals(studentId)).findFirst().get();
23      }
24
25      public List<Student> getAllStudents() 1 usage
26      {
27          return students;
28      }
29  }
```

## Controller Layer

Let's now create a student controller. The controller will be responsible for mapping the "/student" endpoint to the desired method.

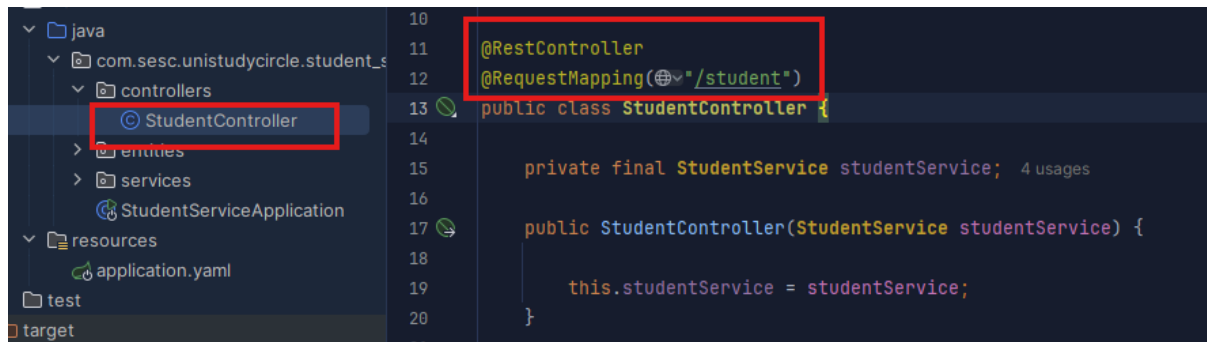
In your src/main/java directory, create a package called:

*com.sesc.unistudycircle.student\_service.controllers*

This is where our student controller class will live.

Create a class called `StudentController` in the package `com.sesc.unistudycircle.student_service.controllers`

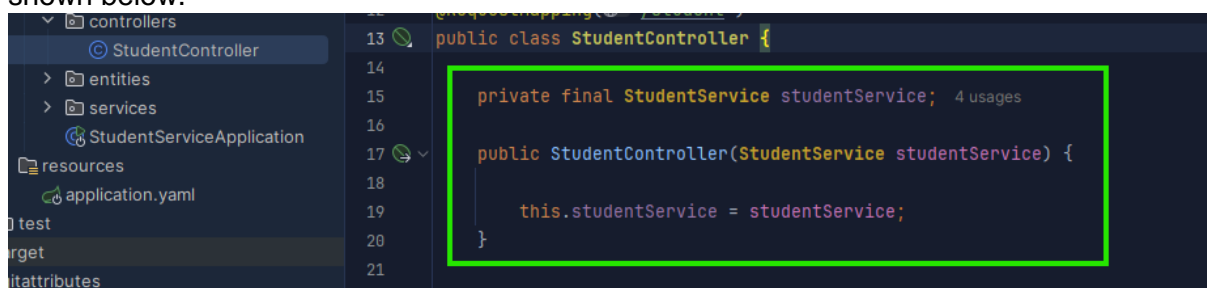
Double click on the `StudentController` class (left side window) and add `@RestController` and `@RequestMapping` annotations on this class as shown below.



The **@RestController** provides hints for Spring that the class plays a specific role. In this case, our class is a web @Controller, so Spring considers it when handling incoming web requests.

The **@RequestMapping** annotation provides “routing” information. It tells Spring that any HTTP request with the / path should be mapped to the home method. The @RestController annotation tells Spring to render the resulting string directly back to the caller.

Your controller should look similar to the one above. Here, we are going to use the concept of **dependency injection for injecting StudentService dependency** as a constructor argument shown below.



We are now ready to write our first method for creating student record in the `StudentController` class. This method will handle incoming HTTP POST request for creating a new student record.

Add a method called `createStudent()` and map the POST endpoint `"/student/create"` to your controller method.

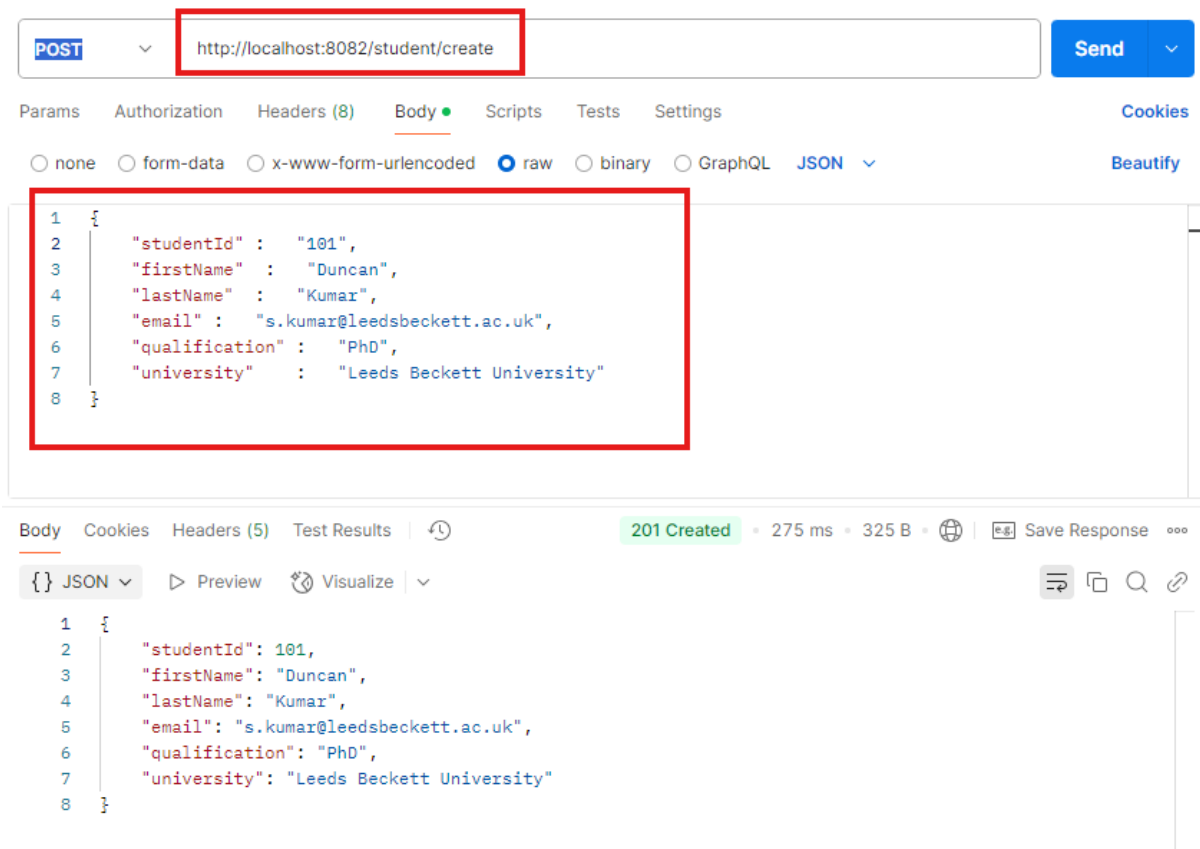
```

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

```

NOW, run the application and test the API endpoint.

Let's open Postman software and select the POST method and type the URL shown below. Further, you choose the select the Body option and type the student properties and their values as follows



Let's create one more student record in the list and then add a method called `getStudents()` to access all students record exist in the system (List).

Add a method called `getStudents()` and map the GET endpoint `"/student"` to your controller method.



```
@GetMapping
public ResponseEntity<List<Student>> getStudents() {
    List<Student> studentList = studentService.getAllStudents();
    return new ResponseEntity<>(studentList, HttpStatus.OK);
}
```

Implementation of getAllStudent() in student controller

select the GET method and type the URL shown below to retrieve all students records exists in the system (List)

The screenshot shows a REST client interface. At the top, the method is set to 'GET' and the URL is 'http://localhost:8082/student'. Below this, the 'Body' tab is selected, showing a JSON response. The response is a list of two student objects, each with fields: studentId, firstName, lastName, email, qualification, and university. The status bar indicates a '200 OK' response with a time of 11 ms and a size of 481 B.

```
{
  "studentId": 101,
  "firstName": "Duncan",
  "lastName": "Kumar",
  "email": "s.kumar@leedsbeckett.ac.uk",
  "qualification": "PhD",
  "university": "Leeds Beckett University"
},
{
  "studentId": 102,
  "firstName": "Alex",
  "lastName": "Mullier",
  "email": "a.mullier@leedsbeckett.ac.uk",
  "qualification": "PhD",
  "university": "Leeds Beckett University"
}
```

Add one more method in the student controller for accessing student record based on their ID.

Add a method called `getStudentById()` and map the GET endpoint `"/student/102"` to your controller method.

```
@GetMapping("/{studentId}")
public ResponseEntity<Student> getStudentById(@PathVariable long studentId) {
    Student student = studentService.getStudentById(studentId);
    return new ResponseEntity<>(student, HttpStatus.OK);
}
```

Implementation of `getStudentById()` in student controller

select the GET method and type the URL shown below to access student record based on supplied student ID in the URL.

The screenshot shows a REST client interface. At the top, the method is set to 'GET' and the URL is 'http://localhost:8082/student/101'. Below this, there are tabs for Params, Authorization, Headers (6), Body, Scripts, Tests, and Settings. The 'Body' tab is selected, and the request body is empty. The response is shown below, with a status of '200 OK', a time of '11 ms', and a size of '320 B'. The response body is a JSON object representing a student record, which is highlighted with a red box.

```
{
  "studentId": 101,
  "firstName": "Duncan",
  "lastName": "Kumar",
  "email": "s.kumar@leedsbeckett.ac.uk",
  "qualification": "PhD",
  "university": "Leeds Beckett University"
}
```

## Conclusion

At this point you have:

- gained theoretical and practical knowledge of implementing RESTful API with Spring Boot Framework.
- discovered how to leverage Spring IoC for injecting dependencies in RESTful API.
- worked independently to implement a full range of REST API methods (GET, POST).

Well done!

There were many new concepts covered this week, particularly if you are new to Java development. You should take the time to work through the examples carefully, and don't be afraid to look things up to gain a deeper understanding of what is happening in the background. Any questions, please feel free to tag us on Discord.