School Of Engineering Second Year CSE (Hons) Assignment in Advance Programming.

Assignment Title

Build a REST web application using the Spring Boot Framework.

Student ID: 4120846

Submission Date:

23.00 Pm Thursday 14 March 2024.

Module Leader:

Mike Child

Table of Contents:

1. Introduction:

This coursework involves the development of a RESTful web application using Spring Boot, focusing on the management of university divisions and courses. The application is designed to support CRUD (Create, Read, Update, Delete) operations on courses within divisions, catering to the academic sector's needs.

1.1. Overview Of the REST Web Application:

The application serves as an interface for managing academic courses and divisions, integrating with a front-end JavaScript interface, and utilizing an H2 in-memory database for persistence. It is structured around RESTful principles, offering a clear and scalable API for web services.

1.2. Objective and Scope:

The primary objective is to design and implement a functional REST web application that meets specified requirements, including handling divisions, courses, and their associated operations. The scope includes API development, database integration, and ensuring application security and robustness.

1.3. Relevance to Advance Programming:

This project is highly relevant to advanced programming, as it encompasses the use of modern software development frameworks, RESTful API design, and the integration of various technologies such as JPA for database operations and Spring Boot for application development. It challenges students to apply theoretical knowledge in a practical context, enhancing their programming, problem-solving, and software design skills.

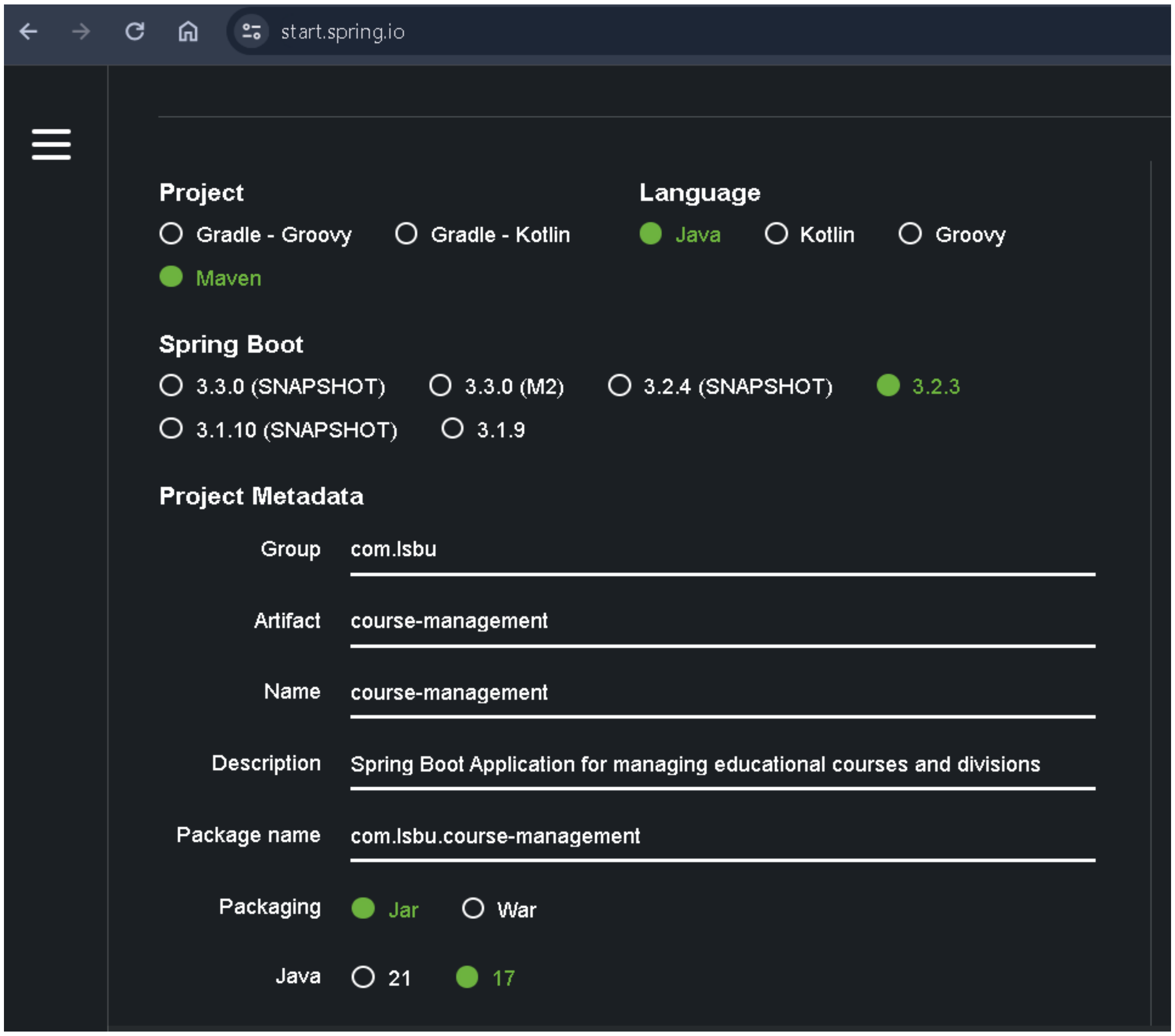
2. Setting Up the Project.

In the following part, I will explain the project setup and dependencies I chose to develop a REST web application using the Spring Boot framework.

2.1. Creation Using spring initializr.

To initiate the creation of our CourseManagementApplication, I leveraged the Spring Initializr web interface, which is a convenient and interactive way to bootstrap a new Spring Boot project. This tool greatly simplifies the project setup process by generating a project structure with the desired dependencies and build configuration.

For my project's setup, the following options I did choose based on the project requirements:


start.spring.io

Project
Gradle - Groovy [✓ Maven] Gradle - Kotlin

Language
[✓ Java] Kotlin Groovy

Spring Boot
3.3.0 (SNAPSHOT) 3.3.0 (M2) 3.2.4 (SNAPSHOT) [✓ 3.2.3] 3.1.10 (SNAPSHOT) 3.1.9

Project Metadata

Group	com.lsbu
Artifact	course-management
Name	course-management
Description	Spring Boot Application for managing educational courses and divisions
Package name	com.lsbu.course-management
Packaging	[✓ Jar] War
Java	21 [✓ 17]

Figure 1.00: Spring Initializr project setup.

Project Type: Maven Project

Language: Java

Spring Boot Version: 3.2.3

Group: com.lsbu

Artifact: course-management

Name: course-management

Description: Spring Boot Application for managing educational courses and divisions

Package Name: com.lsbu.course-management

Packaging: Jar

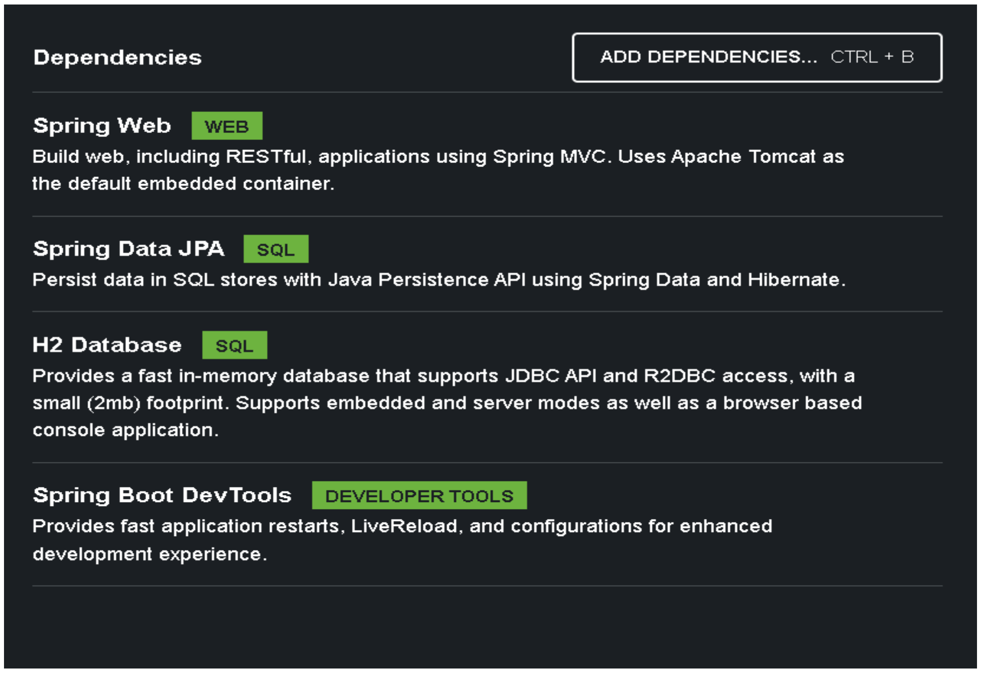
Java Version: 17

Maven was selected as the build tool due to its wide adoption in the industry and its straightforward integration with IDEs and other development tools. It offers a powerful dependency management system and simplifies the build process with its convention-over-configuration approach.

The choice of Java as the programming language was influenced by its robustness, strong typing, and widespread use in enterprise applications. The decision to use Java 17 was made considering it is a long-term support (LTS) version, ensuring long-term stability and support.

2.2. Dependencies Selection and rationale.

Here is the brief explanation about the dependencies that I choose for this project:



Dependencies

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

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

H2 Database SQL
Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Figure 1.01: Selected Dependencies.

Spring Web:

This dependency is used for building web applications using Spring MVC. It includes RESTful application support and is the key to creating controllers, handling requests, and returning responses. It also comes with Apache Tomcat as the default embedded container, enabling the deployment of web applications without the need for an external server.

Spring Data JPA:

Facilitates the creation of JPA-based repositories and data access layers, reducing boilerplate code and simplifying database interactions, with Hibernate as the default implementation.

H2 Database:

A compact, in-memory database ideal for development and testing, providing a quick setup and JDBC API support. It features a console for direct database interactions.

Spring Boot DevTools:

Aims to improve the development workflow by enabling features like automatic restarts and live reloading for a more dynamic development process.

3. Initial Configuration and Runs.

3.1. Project Structure and Setup.

My Spring Boot application is organized into several key classes, each fulfilling a specific role within the project:

- CourseManagementApplication: This is the main class that boots up the Spring application. It contains the main method that Spring Boot uses to launch the application.

- Course: This is a domain model class annotated with @Entity, representing the courses within the educational divisions in the application's database.

- CourseRepository: An interface extending JpaRepository, providing CRUD operations for the Course domain objects without the need for an explicit implementation.

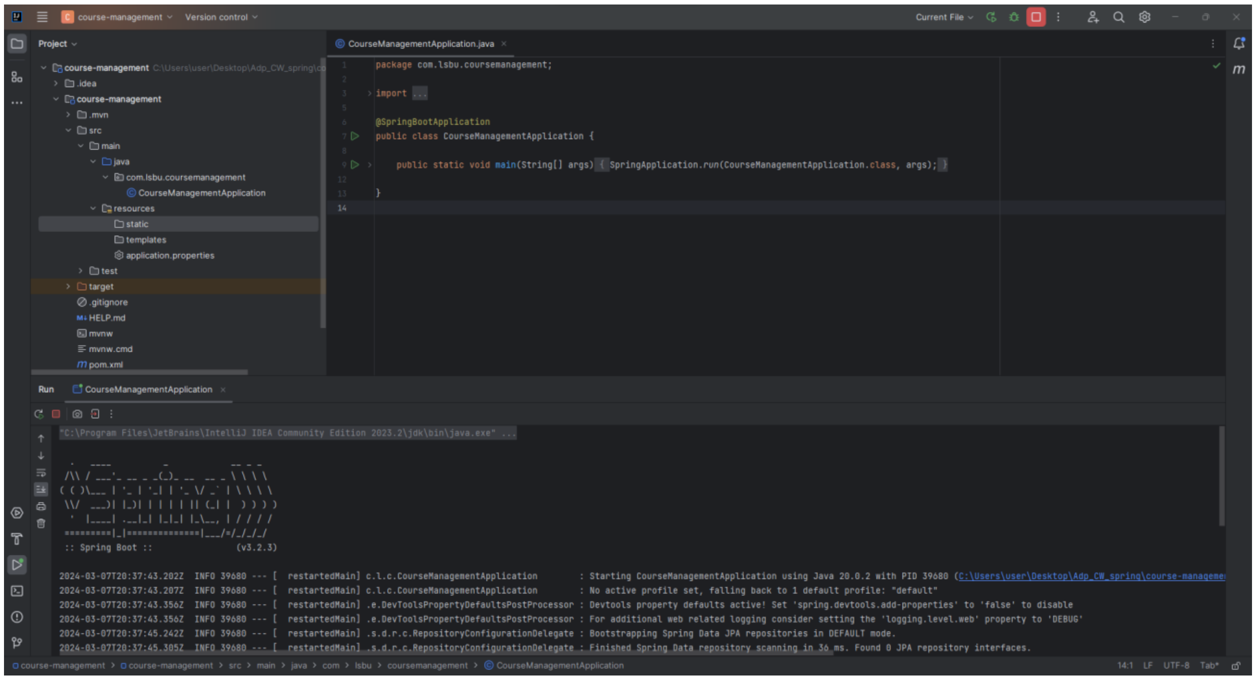
- CourseController: A REST controller class that handles HTTP requests and responses. It defines the endpoints for CRUD operations on the Course entities.

- DataLoader: This class implements CommandLineRunner and is marked with @Configuration, indicating it is a configuration class that runs when the application starts. It preloads the database with initial course data.

- ResourceNotFoundException: A custom exception class used within the controller to handle cases where a course cannot be found.

3.2. Documentation of the Initial Project Execution

Upon launching the CourseManagementApplication, Spring Boot initializes the embedded Apache Tomcat server and sets up the application context. The console logs indicate the server's port 8080, context path, and start up duration. On the first run, no errors were encountered as I followed the tutorial video and I just added additional functionality based on my project requirements, and the log output confirmed that the application started successfully.



```java
package com.lsbu.coursemanagement;

import ...

@SpringBootApplication
public class CourseManagementApplication {

    public static void main(String[] args) { SpringApplication.run(CourseManagementApplication.class, args); }

}
```

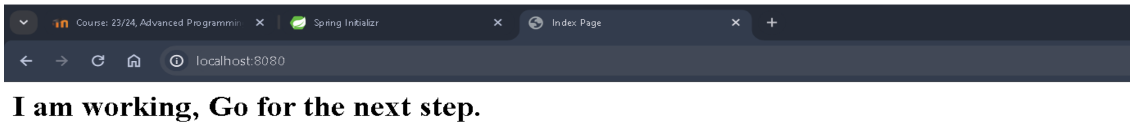
```
"C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2023.2\jdk\bin\java.exe" ...

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.2.3)

2024-03-07T20:37:43.202Z  INFO 39680 --- [  restartedMain] c.l.c.CourseManagementApplication        : Starting CourseManagementApplication using Java 20.0.2 with PID 39680 (C:\Users\user\Desktop\Adp_CW_spring\course-manageme
2024-03-07T20:37:43.207Z  INFO 39680 --- [  restartedMain] c.l.c.CourseManagementApplication        : No active profile set, falling back to 1 default profile: "default"
2024-03-07T20:37:43.356Z  INFO 39680 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtools.add-properties' to 'false' to disable
2024-03-07T20:37:43.356Z  INFO 39680 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging.level.web' property to 'DEBUG'
2024-03-07T20:37:45.242Z  INFO 39680 --- [  restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2024-03-07T20:37:45.305Z  INFO 39680 --- [  restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 36 ms. Found 0 JPA repository interfaces.
```

Fugure 1.03: Initial Project run.

After I open the file, I run the project to check that the server is running and working. Secondly, under the static file I create an index.html file by following the tutorial video and I put some text inside that, and I run the server again on the tomcat server using port 8080.

In the following part I will provide my index.html file and will run the program and check that the server is running:

```
<html>
<head>
   <title>Index Page</title>
</head>
<body>
<h1>I am working, Go for the next step</h1>
</body>
</html>
```



I am working, Go for the next step.

Figure 1.04: Tomcat server is running with port number 8080.

3.3. Initial code Snippet and Program Output. The DataLoader class in my project is a configuration class that is responsible for pre-loading the H2 database with initial data when the application starts.

DataLoader Class Explanation:

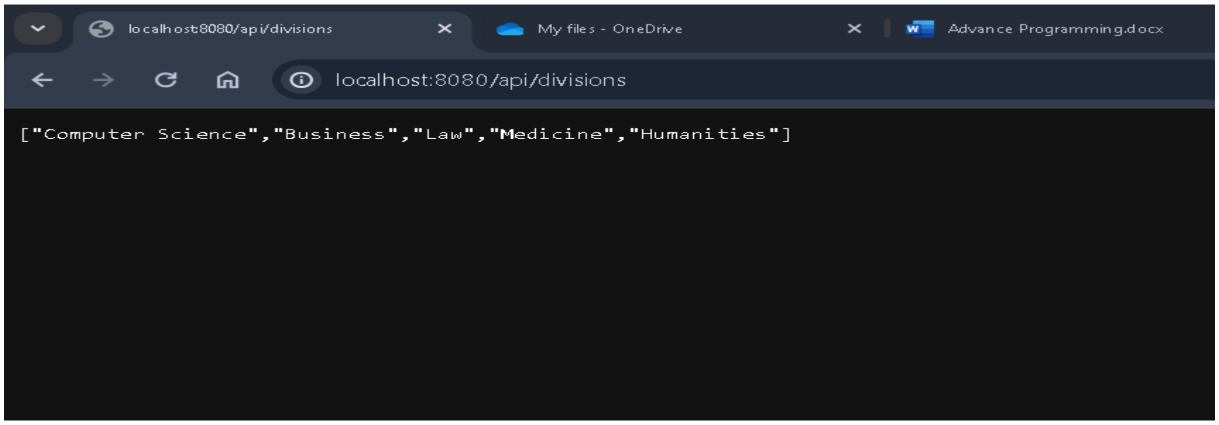
- @Configuration Annotation: This annotation indicates that the class can contain bean definitions that will be managed by the Spring container. It tells Spring that this class is part of the configuration for my application.
- @Autowired CourseRepository: This field injection tells Spring to inject an instance of CourseRepository into my DataLoader class. CourseRepository is an

interface that extends JpaRepository, providing convenient methods for CRUD operations without the need for an explicit implementation.

- @Bean CommandLineRunner initDatabase(): The @Bean annotation here marks the initDatabase method as a Spring bean, and since it returns a CommandLineRunner, it will be executed after the application context is loaded. The CommandLineRunner interface provides a single run method that contains logic that is executed on startup.
- The run Method: Inside this method, I create several Course objects that represent different courses in various academic divisions of LSBU. After that, I save these objects to the database using the save method provided by the CourseRepository.
- When I start my Spring Boot application, after all the Spring-related configurations are loaded and the web server is started (embedded Tomcat in this case), the initDatabase method will automatically be called. This will lead to the insertion of the predefined Course entities into the H2 database.

In the following part, inside the table I will display my dataloader class and will show one example by requesting one Get method how this data coming to the server:

When I make a GET request to, “api/divisions” the application will query the in-memory database using CourseRepository, find all divisions, and return them in the response.



```
["Computer Science","Business","Law","Medicine","Humanities"]
```

Figure 1.05: First Get method run and output.

DataLoader Class:

```
@Bean
CommandLineRunner initDatabase() {
   return args -> {
      // Adding initial course data to the database
      courseRepository.save(new Course("Introduction to Programming", 120L, true,
true, 9.0, "Computer Science"));
      courseRepository.save(new Course("Data Structures", 100L, true, true, 8.5,
"Computer Science"));

      courseRepository.save(new Course("Principles of Marketing", 85L, true, false, 7.0,
"Business"));
      courseRepository.save(new Course("Organizational Behavior", 90L, true, false,
7.5, "Business"));

      // New divisions and courses
      courseRepository.save(new Course("Constitutional Law", 60L, true, true, 8.0,
"Law"));
      courseRepository.save(new Course("Criminal Law", 50L, true, false, 7.5, "Law"));

      courseRepository.save(new Course("Anatomy and Physiology", 120L, true, true,
8.5, "Medicine"));
      courseRepository.save(new Course("Biochemistry", 100L, true, false, 8.0,
"Medicine"));

      courseRepository.save(new Course("World History", 80L, true, true, 7.0,
"Humanities"));
      courseRepository.save(new Course("Philosophy", 75L, true, false, 7.5,
"Humanities"));

   };
}
```

4. Domain Object Class:

4.1. Presentation and rationale for the domain model.

This Course class is an essential component of my application, designed to encapsulate the data and behaviors associated with a course within the context of LSBU educational management system. Annotated with @Entity, it signifies a persistent object that will map to a database table.

Each field in the Course entity corresponds to a column in the database and is intended to hold specific information about the course:

- ‘id’: Auto-generated to ensure each course has a unique identifier, making retrieval and manipulation straightforward.
- ‘name’: Essential for course identification and selection by users.
- ‘duration’: Indicates the commitment required and assists in planning and scheduling.
- ‘undergraduate’: Helps in filtering courses based on the educational level.
- ‘gicode and jdvalue’: Business-specific metrics that could relate to funding eligibility, course valuation, or other administrative purposes.
- ‘division’: Allows categorization of courses, aiding in organization and search functionality within the application.

4.2. Initial Code and subsequent Iterations:

The initial implementation of the Course class provided a simple structure with only the essential fields:

```
@Entity
public class Course {
  @Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
private String name; // Course name
private Long duration; // Course duration
private Boolean undergraduate; // the course is for undergraduate students
private Boolean giCode; // Placeholder for GI code logic (true/false)
private Double jdValue; // Placeholder for JD value
private String division; // Division/category the course belongs to
   // Constructors, getters, and setters...
}
```

Through iterative development, more attributes were added to fulfill the expanding requirements:

```
private Long duration;
private Boolean undergraduate;
private Boolean giCode;
private Double jdValue;
```

Subsequent iterations of the Course class evolved to include detailed getters and setters for each attribute, along with a parameterized constructor to allow for easy instantiation and data population:

```
// Constructors
public Course() {}

public Course(String name, Long duration, Boolean undergraduate, Boolean giCode,
Double jdValue, String division) {
   this.name = name;
   this.duration = duration;
   this.undergraduate = undergraduate;
   this.giCode = giCode;
   this.jdValue = jdValue;
   this.division = division;
}
```

The toString() method was overridden to aid in debugging by providing a string representation of the Course object's current state:

```
@Override
public String toString() {
   return "Course{" +
         "id=" + id +
         ", name='" + name + '\'' +
         ", duration=" + duration +
         ", undergraduate=" + undergraduate +
         ", giCode=" + giCode +
         ", jdValue=" + jdValue +
         ", division='" + division + '\'' +
         '}';
}
```

This evolution of the Course class reflects a deeper understanding of the domain and a response to specific needs encountered during the development and deployment of the application. Each iteration improved the model's alignment with the business context and the technical requirements of the system.

5. Persistence Implementation:

5.1. Code and explanations for JPA and database integration.

For the persistence layer of my application, I utilized the Java Persistence API (JPA) with Hibernate as the provider, which is the standard for object-relational mapping (ORM) in Java. This approach allows me to map my Course domain class to a relational database table in an object-oriented way, making database interactions more natural within the Java programming language.

My ‘CourseRepository’ interface extends ‘JpaRepository’, which is part of Spring Data JPA and provides a rich set of CRUD operations along with JPA-specific method enhancements. Here is the CourseRepository interface:

```
package com.lsbu.coursemanagement;

import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface CourseRepository extends JpaRepository<Course, Long> {
   List<Course> findByDivision(String division);

   // Find a single course within a specific division by its ID
   Course findByIdAndDivision(Long id, String division);
}
```

- findByDivision(String division): This method signature instructs Spring Data JPA to generate a query that retrieves all Course entities where the division attribute matches the provided argument. This takes advantage of the query derivation from method names feature of Spring Data JPA.

- findByIdAndDivision(Long id, String division): This method provides a way to search for a specific course within a division by its ID, demonstrating the capability of Spring Data JPA to combine multiple criteria in a single query method.

6. API design and implementation.

6.1. Endpoint structure and Rationale.

The CourseController class is structured to manage courses within various educational divisions, following RESTful design principles. It defines endpoints for CRUD operations and more, utilizing HTTP verbs to indicate action types and URI paths to specify resources:

- ‘GET /api/divisions’: Lists all unique divisions. This endpoint demonstrates a read operation on a higher level of abstraction, focusing on divisions rather than individual courses. It uses the power of Java Streams to filter out unique division names, ensuring that users can navigate the available categories before drilling down into specific courses.

```
// List all unique divisions
@GetMapping("")
public ResponseEntity<List<String>> getAllDivisions() {
    List<Course> courses = courseRepository.findAll();
    List<String> divisions = courses.stream()
            .map(Course::getDivision)
            .distinct()
            .collect(Collectors.toList());
    if (divisions.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<>(divisions, HttpStatus.OK);
}
```

- ‘GET /api/divisions/{division}’: Retrieves all courses within a specified division. It adheres to REST principles by making the division part of the URI, which helps in organizing the courses into logical groupings, enhancing user navigation and data retrieval efficiency.

```
// List all courses by division
@GetMapping("/{division}")
public ResponseEntity<List<Course>> getCoursesByDivision(@PathVariable String
division) {
    List<Course> courses = courseRepository.findByDivision(division);
    if (courses.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<>(courses, HttpStatus.OK);
}
```

- ‘GET /api/divisions/{division}/courses/all’: Extends the previous GET operation by not only listing courses within a division but also including a result code within the response body, improving API communication by explicitly stating the operation's outcome ("SUCCESS" or "ERROR").

```
// List all courses for a specified division with a result code
@GetMapping("/{division}/courses/all")
public ResponseEntity<Map<String, Object>>
getAllCoursesByDivisionWithResultCode(@PathVariable String division) {
    List<Course> courses = courseRepository.findByDivision(division);
    Map<String, Object> response = new HashMap<>();
    if (courses.isEmpty()) {
        response.put("result", "ERROR");
        response.put("message", "No courses found for division: " + division);
        return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
    } else {
        response.put("result", "SUCCESS");
        response.put("courses", courses);
        return new ResponseEntity<>(response, HttpStatus.OK);
    }
}
```

- ‘GET /api/divisions/{division}/courses/{id}’: Fetches a specific course by its ID within a specified division, demonstrating RESTful API's capability to narrow down to a single resource using path variables. It incorporates result codes for clear communication of the operation's result.

```
// Retrieve a specific course by ID within a specified division with a result code
@GetMapping("/{division}/courses/{id}")
public ResponseEntity<Map<String, Object>>
getCourseByIdAndDivisionWithResultCode(@PathVariable String division,
@PathVariable Long id) {
    Map<String, Object> response = new HashMap<>();
    Optional<Course> courseOpt =
Optional.ofNullable(courseRepository.findByIdAndDivision(id, division));

    if (courseOpt.isPresent()) {
        response.put("result", "SUCCESS");
        response.put("course", courseOpt.get());
        return new ResponseEntity<>(response, HttpStatus.OK);
    } else {
        response.put("result", "ERROR");
        response.put("message", String.format("Course with id %d not found in
division: %s", id, division));
        return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
    }
}
```

- POST /api/divisions/{division}/courses: Allows the creation of new courses within a division. This endpoint exemplifies how client-supplied data (in the request body) can be used to create new resources within a specific context (division), highlighted using @RequestBody and path variables.

```
// post method used to Create a new course within a specified division
@PostMapping("/{division}/courses")
public ResponseEntity<Map<String, Object>> addCourseToDivision(@PathVariable
String division, @RequestBody Course course) {
   Map<String, Object> response = new HashMap<>();
   try {
      // Set the division of the course from the path variable
      course.setDivision(division);
      Course savedCourse = courseRepository.save(course);
      response.put("result", "SUCCESS");
      response.put("course", savedCourse);
      return new ResponseEntity<>(response, HttpStatus.CREATED);
   } catch (Exception e) {
      response.put("result", "ERROR");
      response.put("message", "Failed to create the course in division: " + division);
      return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
   }
}
```

- PUT /api/divisions/{division}/{id} and DELETE /api/divisions/{division}/{id}: These endpoints enable updating and deleting specific courses, respectively. They exemplify REST principles for modifying and removing resources, using path variables to pinpoint the exact course to be altered or deleted.

```
// Update an existing course within a specified division
@PutMapping("/{division}/{id}")
public ResponseEntity<Map<String, Object>>
updateCourseInDivision(@PathVariable String division, @PathVariable Long id,
@RequestBody Course updatedCourseDetails) {
   Map<String, Object> response = new HashMap<>();
   Optional<Course> courseOpt =
Optional.ofNullable(courseRepository.findByIdAndDivision(id, division));

   if (courseOpt.isPresent()) {
      Course course = courseOpt.get();
      // Update course details
      course.setName(updatedCourseDetails.getName());
      course.setDuration(updatedCourseDetails.getDuration());
      course.setUndergraduate(updatedCourseDetails.getUndergraduate());
```

```
        course.setGiCode(updatedCourseDetails.getGiCode());
        course.setJdValue(updatedCourseDetails.getJdValue());
        // No need to update division as it's derived from the path variable and should
remain the same

        Course updatedCourse = courseRepository.save(course);
        response.put("result", "SUCCESS");
        response.put("course", updatedCourse);
        return new ResponseEntity<>(response, HttpStatus.OK);
    } else {
        response.put("result", "ERROR");
        response.put("message", String.format("Course with id %d not found in
division: %s", id, division));
        return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
    }
}


// Delete an existing course within a specified division
@DeleteMapping("/{division}/{id}")
public ResponseEntity<Map<String, Object>>
deleteCourseFromDivision(@PathVariable String division, @PathVariable Long id) {
    Map<String, Object> response = new HashMap<>();
    Optional<Course> courseOpt =
Optional.ofNullable(courseRepository.findByIdAndDivision(id, division));

    if (courseOpt.isPresent()) {
        courseRepository.delete(courseOpt.get());
        response.put("result", "SUCCESS");
        response.put("message", String.format("Course with id %d in division: %s has
been deleted", id, division));
        return new ResponseEntity<>(response, HttpStatus.OK);
    } else {
        response.put("result", "ERROR");
        response.put("message", String.format("Course with id %d not found in
division: %s", id, division));
        return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
    }
}
```

6.2. CRUD Operation for the domain model.

The ‘CourseController’ class impalements the CRUD operation as follows:

- Create: The ‘addCourseToDivision’ method uses POST to add a new course, demonstrating data creation within the application.

- Read: Methods like ‘getAllDivisions’ and ‘getCoursesByDivision’ provide read functionality, highlighting how data can be fetched and displayed.
- Update: The ‘updateCourseInDivision’ method illustrates the update functionality, allowing changes to be made to an existing course.
- Delete: The ‘deleteCourseFromDivision’ method shows how a course can be removed, aligning with the delete operation of CRUD.

7. Integration with Front-end technology:

7.1. Front-end and Back-end integration testing strategies.

Integration testing is a critical phase in my development process, where I verify that my Spring Boot backend (comprising of various RESTful endpoints defined in CourseController) works seamlessly with my front-end interface. This involves testing the combined functionality of both the front-end and back-end components to ensure they operate together as expected.

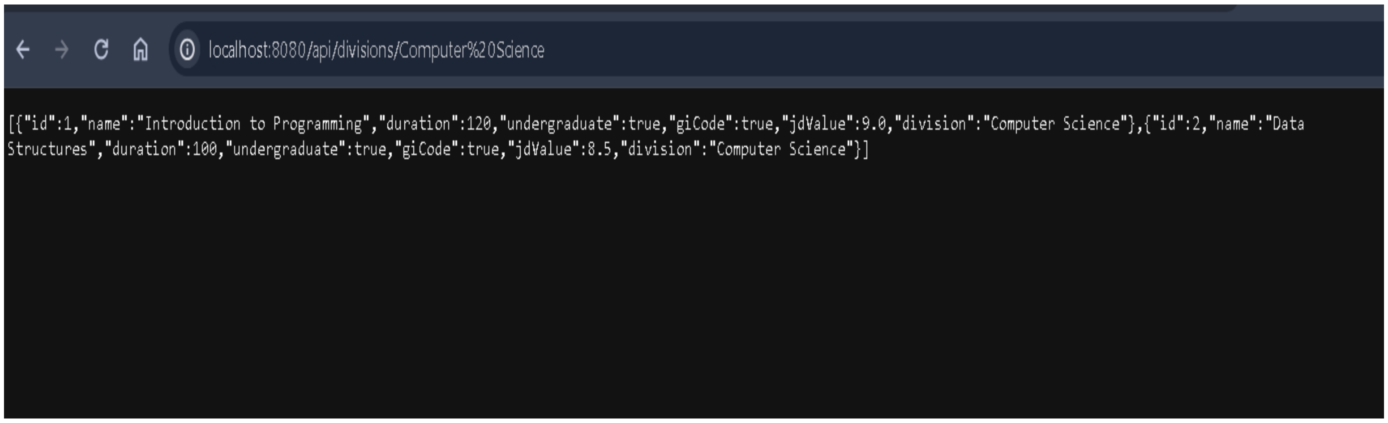
Strategy Outline:

1. **Setup:**
- I had to first ensure that the backend server is running and that the ‘DataLoader’ has successfully preloaded the database with test data.
2. **Automated testing:**
- I verify that the correct API calls are made when a user performs actions like listing courses, adding a new course, or editing an existing course.
- simulate user interactions with the front-end interface that, in turn, makes HTTP requests to the backend.
3. **Manual Testing:**
- I manually test the user interface provided by the teacher (generic-interface.html) to ensure it correctly interacts with the backend.
- Each action, such as creating or fetching courses, is performed manually, and the resulting behavior and data displayed on the front-end are verified for accuracy.
4. **Observation:**
- Throughout the testing process, I observe the system's behavior, checking for correct status codes, valid response bodies, and that the front-end correctly reflects changes made to the backend data.
5. **Validation:**
- The responses from the backend are validated against expected values to ensure integrity and correctness.
6. **Tools and Technique:**

- Network inspection tools are used to monitor the HTTP requests and responses between the front-end and the backend.

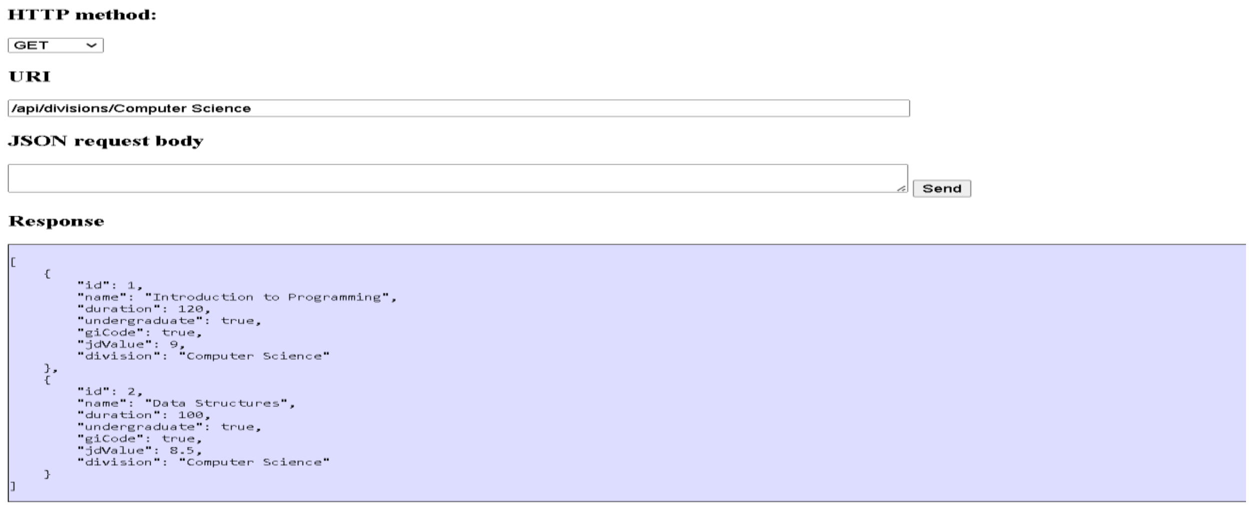
Testing Examples:

- A ‘GET’ request to ‘/api/divisions/Computer Science’ is returning a list of courses in the 'Computer Science' division, which the front-end should correctly display.


localhost:8080/api/divisions/Computer%20Science

```
[{"id":1,"name":"Introduction to Programming","duration":120,"undergraduate":true,"giCode":true,"jdValue":9.0,"division":"Computer Science"},{"id":2,"name":"Data Structures","duration":100,"undergraduate":true,"giCode":true,"jdValue":8.5,"division":"Computer Science"}]
```

Figure 1.06: List of all courses in Computer science division using HTTP request.


HTTP method: GET

URI: /api/divisions/Computer Science

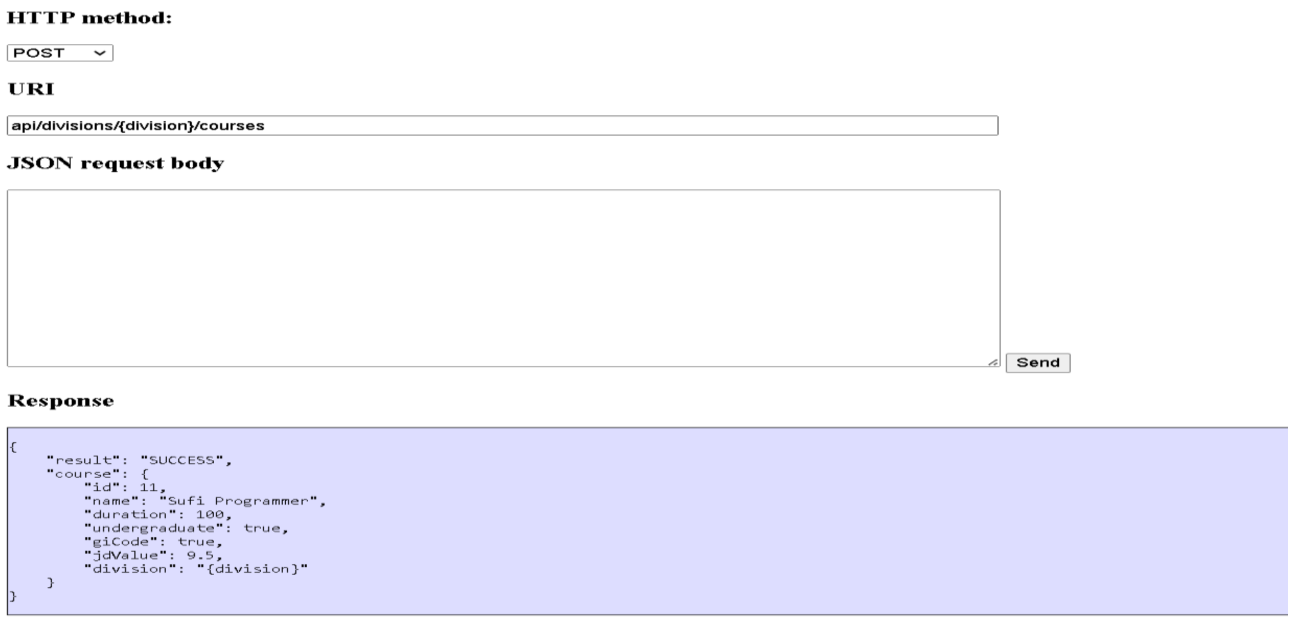
JSON request body

Response

```
[
    {
        "id": 1,
        "name": "Introduction to Programming",
        "duration": 120,
        "undergraduate": true,
        "giCode": true,
        "jdValue": 9,
        "division": "Computer Science"
    },
    {
        "id": 2,
        "name": "Data Structures",
        "duration": 100,
        "undergraduate": true,
        "giCode": true,
        "jdValue": 8.5,
        "division": "Computer Science"
    }
]
```

Figure 1.07: Request all courses using Generic Interface.

- A ‘POST’ request made through a form submission on the front-end interface to add a new course should result in a new entry in the database, visible upon subsequent data retrieval using the URI ‘api/divisions/{division}/courses’ provided in the personalised specification.



api/divisions/{division}/courses

```
{
    "result": "SUCCESS",
    "course": {
        "id": 11,
        "name": "Sufi Programmer",
        "duration": 100,
        "undergraduate": true,
        "giCode": true,
        "jdValue": 9.5,
        "division": "{division}"
    }
}
```

Figure 1.08: Post a new course Using generic-interface.

8. Testing and Validation.

8.1. Unit testing with JUnit.

In the unit testing phase, I focus on the smallest parts of the application, which are the methods within my classes. Using JUnit, I crafted a suite of tests that exercise the various functionalities encapsulated by this domain model, repository, and controller.

For the Course domain model, tests verify the integrity of the entity, ensuring that getters and setters function as expected, and that the @Entity annotation properly integrates with the JPA provider.

The CourseRepository is tested using @DataJpaTest, which focuses solely on JPA components. Using an embedded database, we confirm that my repository queries—like findByDivision—behave correctly, returning the appropriate data or an empty collection if no matches are found.

The CourseController tests, designed with @WebMvcTest, isolate the web layer and mock the service layer where CourseRepository is injected. I validate that each endpoint correctly interprets HTTP requests and responds with the right status codes and payloads.

9. Remaining API Calls.

9.1. Documentation of Additional API call implementation.

In addition to the foundational CRUD operations implemented in my CourseController, my application supports a few more advanced functionalities that enhance user interaction and data management. Below are the detailed descriptions and justifications for these additional endpoints:

Listing All Unique Divisions (‘GET /api/divisions’):

- **Purpose**

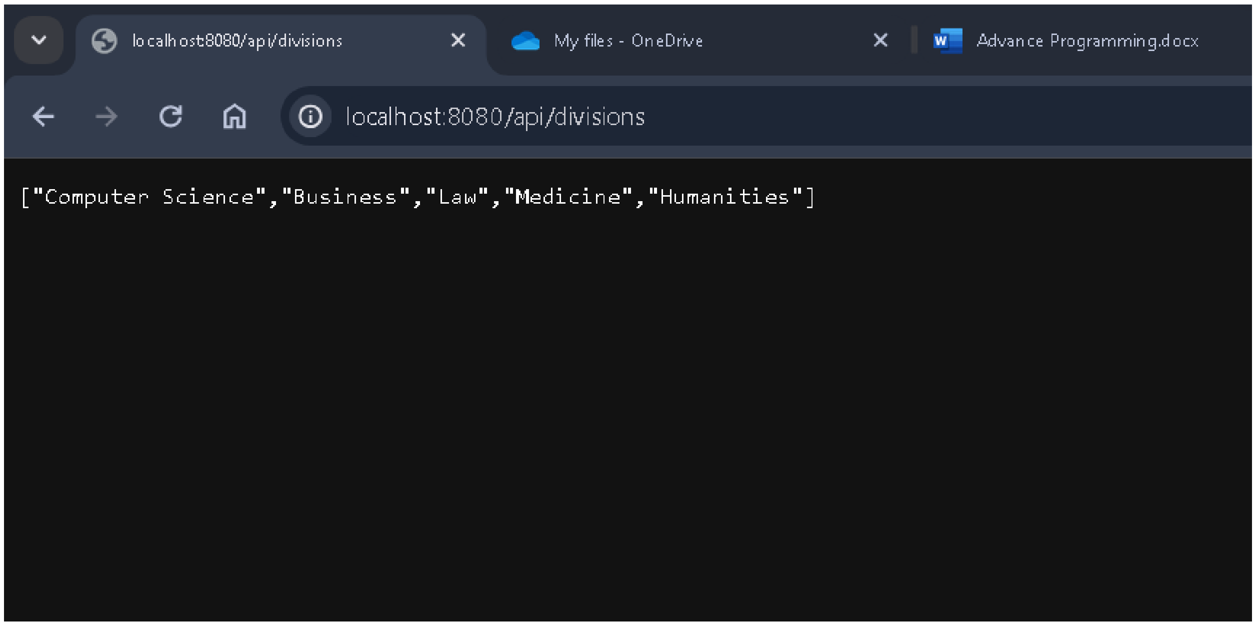
This endpoint provides a list of all unique divisions available in the database, serving to filter or categorize courses by their division. It is crucial for users who need an overview of the organizational structure or are looking for courses within specific divisions.

- **Implementation**

Utilizes the ‘findAll’ method from ‘CourseRepository’ to fetch all courses and then streams the result to collect a distinct list of divisions. This demonstrates efficient use of Java Streams for data processing.

- **Testing**

Utilizes the ‘findAll’ method from ‘CourseRepository’ to fetch all courses and then streams the result to collect a distinct list of divisions. This demonstrates efficient use of Java Streams for data processing.



```
["Computer Science","Business","Law","Medicine","Humanities"]
```

Figure 1.09: List all unique divisions.

- **Retrieving All Courses Within a Division with a Result Code (‘GET /api/divisions/{division}/courses/all’):**

Purpose:

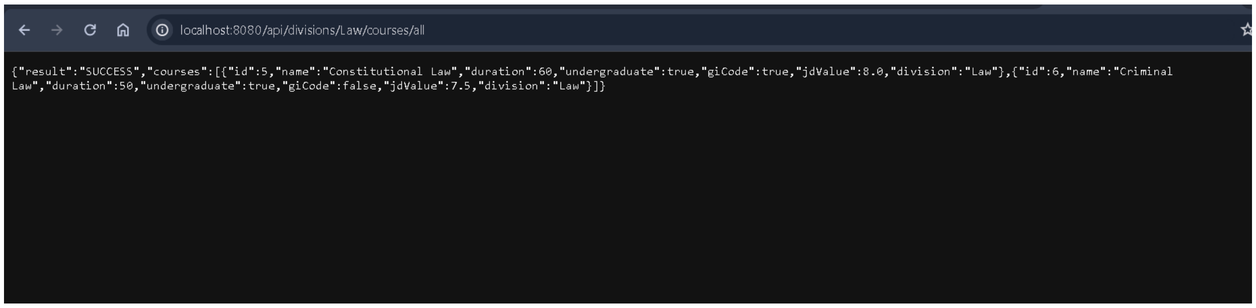
Extends the basic retrieval functionality by not only listing courses within a specified division but also including a "result" code in the response body, enhancing the API's communicative clarity.

Implementation:

Similar to getCoursesByDivision but wraps the response in a map that includes a "result" key (SUCCESS or ERROR) alongside the course data, catering to clients that might require explicit operation feedback.

Testing:

Confirmed by ensuring the endpoint correctly returns the courses along with a "SUCCESS" result code when courses are found, or an "ERROR" code with an appropriate message when no courses are available in the requested division.



```
{"result":"SUCCESS","courses":[{"id":5,"name":"Constitutional Law","duration":60,"undergraduate":true,"giCode":true,"jdValue":8.0,"division":"Law"},{"id":6,"name":"Criminal Law","duration":50,"undergraduate":true,"giCode":false,"jdValue":7.5,"division":"Law"}]}
```

Figure 2.00: Retrieving all courses within a division.

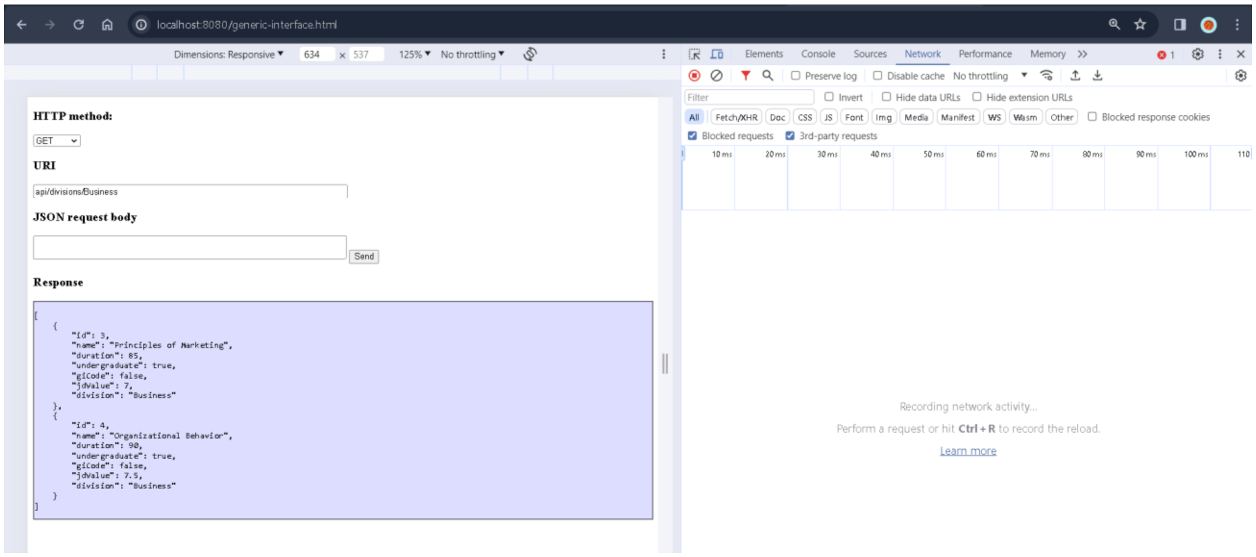
9.2: Interface Screenshot Demonstrating Functionality.

In this section, I visually display the interaction between my front-end interface—generic-interface.html—and the back-end ‘CourseController’. Through a series of screenshots, I will illustrate the successful execution of the four fundamental CRUD operations: Create (POST), Read (GET), Update (PUT), and Delete (DELETE). Each screenshot will highlight the request sent to a specific URI and the resulting response, as facilitated by the ‘DataLoader's’ initial data setup.

- **Get Request:**

First, I made the GET request to the endpoint /api/divisions/{division}. When I enter this URI into the interface and specify a division (in my case, 'Business'), and then send the request, the interface communicates with my Spring Boot back-end application.

Here is a brief functionality what happens when I perform this functionality:


localhost:8080/generic-interface.html

Dimensions: Responsive 634 x 537 125% No throttling

HTTP method:

GET

URI

api/divisions/Business

JSON request body

Send

Response

```
[
    {
        "id": 3,
        "name": "Principles of Marketing",
        "duration": 85,
        "undergraduate": true,
        "giCode": false,
        "jdValue": 7,
        "division": "Business"
    },
    {
        "id": 4,
        "name": "Organizational Behavior",
        "duration": 90,
        "undergraduate": true,
        "giCode": false,
        "jdValue": 7.5,
        "division": "Business"
    }
]
```

Recording network activity...

Perform a request or hit Ctrl + R to record the reload.

Figure 2.01: Get method request.

1. HTTP Get Method Selection.
2. URI Input.
3. Sending The Request.
4. Processing the Request.
- My Spring Boot application's 'CourseController' handles the request with its corresponding handler method. This method uses the 'courseRepository' to find all courses linked to the 'Business' division via the 'findByDivision' method.
5. Formulating the response.
- The 'courseRepository' returns a list of Course objects that match the division query, which is then wrapped in a 'ResponseEntity' with an appropriate HTTP status code. In this case, since the courses are found.
6. Displaying the response.

In this GET request, the back-end service is correctly returning data, indicating that the integration between the front-end interface and the Spring Boot back-end is working as expected.

- **POST Request.**

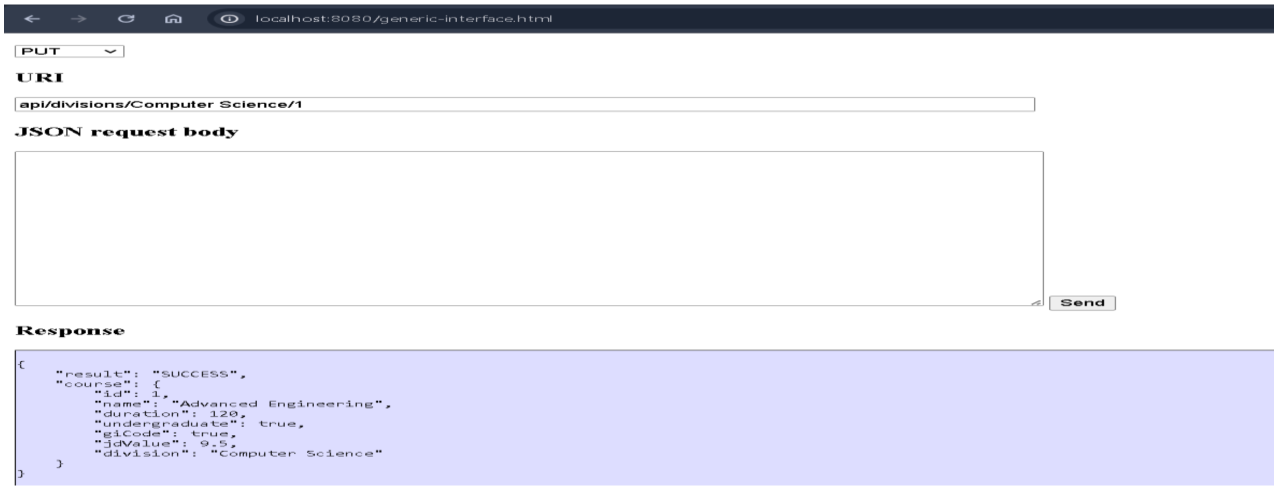
As I already highlighted the POST method in the previous part (Figure 1.08). So, in the following part, I will explain about the POST method functionality.

Method: POST.

URI: 'api/divisions/{division}/courses'.

After processing the request, the server sends a response back to the client and updates the course with the ID and division.

- **PUT Request.**



api/divisions/Computer Science/1

```
{
    "result": "SUCCESS",
    "course": {
        "id": 1,
        "name": "Advanced Engineering",
        "duration": 120,
        "undergraduate": true,
        "giCode": true,
        "jdValue": 9.5,
        "division": "Computer Science"
    }
}
```

Figure 2.02: Update the Course using PUT method.

As I am performing ‘PUT’ request, First I did include the JSON request body with the updated course details (in my case, I updated Computer science with the Advance Engineering). In the URI I put the URI with the division name and ID. Secondly, I click the “Send” button to make the request to my spring boot application. Finally, after sending the request I look at my response section of my generic-interface and I got the output with the success message and can see that above result that the course is updated successfully with the updated version of Course Advance Engineering with the ID.

- **DELETE Request.**

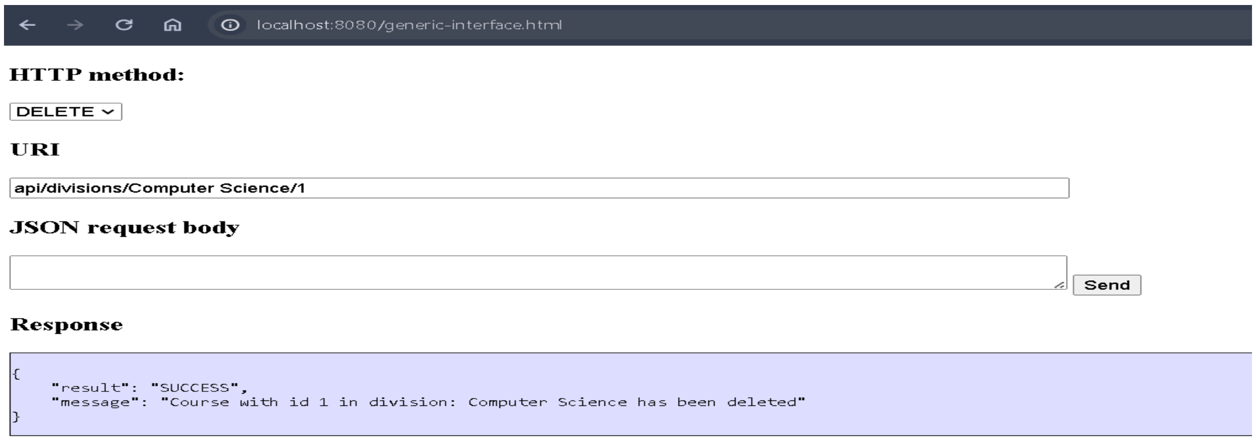
1. HTTP Method Selection: I choose the DELETE method in the interface, which is designed to remove resources from the server.
2. URI Construction: I input the specific URI (api/divisions/Computer Science/1) into the interface. This URI includes the division (Computer Science) and the ID (1) of the course you wish to delete.
3. Sending the Request: When I click "Send", the web interface sends an HTTP DELETE request to my Spring Boot application. The request is routed to the corresponding controller method that matches the URI pattern and the DELETE HTTP method.

- Backend Processing:

1. My Spring Boot application receives the request.
2. The CourseController finds the matching @DeleteMapping annotated method.

3. The path variables {division} and {id} are extracted from the URI and passed to the controller method.
4. The repository class performs the deletion operation on the database.

- Result Verification



DELETE

api/divisions/Computer Science/1

```
{
    "result": "SUCCESS",
    "message": "Course with id 1 in division: Computer Science has been deleted"
}
```

Figure 2.03. Delete the courses using Delete URI.

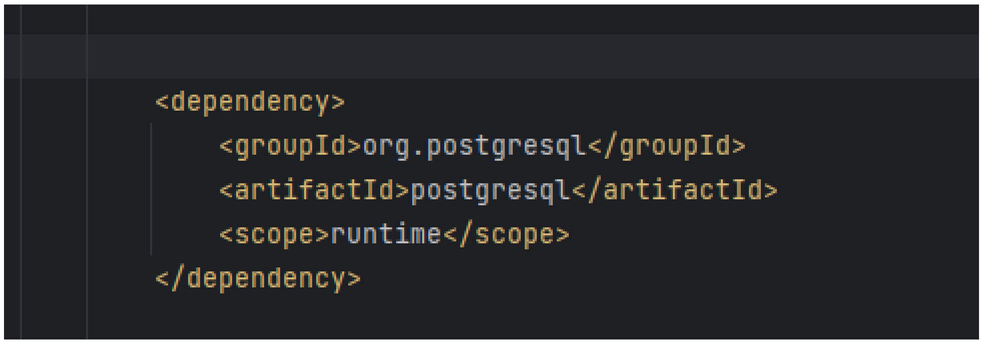
1. Once the course is successfully deleted, the controller typically sends a response back to the client.
2. This response can be a confirmation message, as seen in the screenshot, indicating that the course has been successfully deleted.

10. **Create a Supabase Database.**

To connect to a Supabase (PostgreSQL) database from a Spring Boot application, I follow those steps:

1. Add PostgreSQL Driver Dependency.

I added PostgreSQL driver dependency to my pom.xml file. Here is the example:

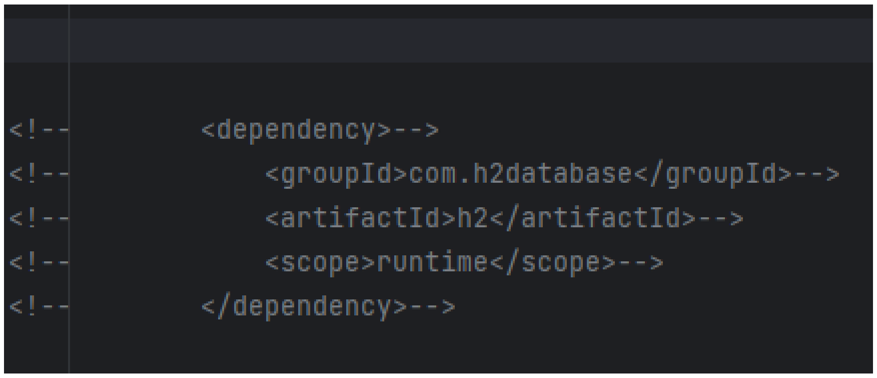


```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

Figure 2.04: PostgreSQL dependency added to the pom.xml file.

2. Comment out H2 Dependency.

As I had an H2 database dependency during the development time and later switched to PostgreSQL, I need to comment it out from the pom.xml.

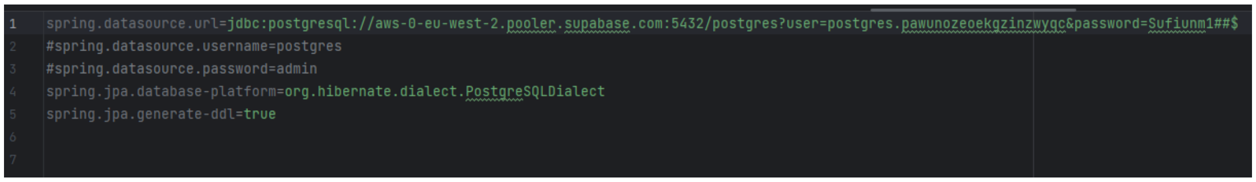


```
<!--        <dependency>-->
<!--            <groupId>com.h2database</groupId>-->
<!--            <artifactId>h2</artifactId>-->
<!--            <scope>runtime</scope>-->
<!--        </dependency>-->
```

Figure 2.05: Comment out H2 Dependency.

3. Set Up ‘application.properties.’

I add the necessary properties to my ‘application.properties’ file to configure the connection to the Supabase database. This includes the JDBC URL, username, and password I received from Supabase.

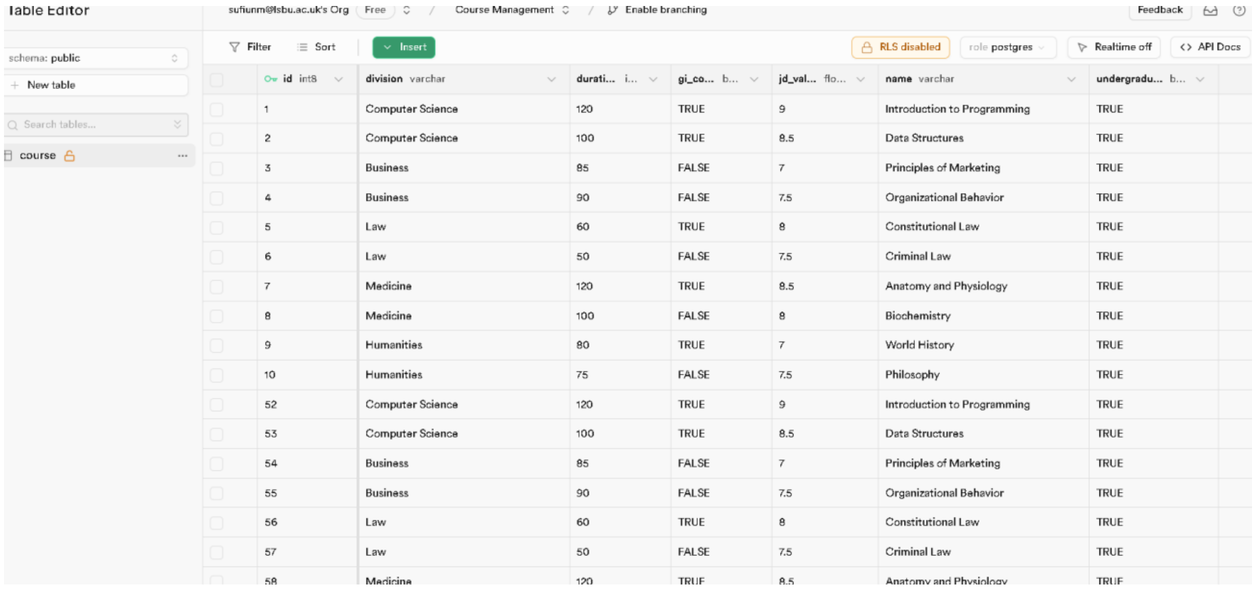


```
spring.datasource.url=jdbc:postgresql://aws-0-eu-west-2.pooler.supabase.com:5432/postgres?user=postgres.pawunozeoekgzinzwyqc&password=Sufiunm1##$
#spring.datasource.username=postgres
#spring.datasource.password=admin
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.generate-ddl=true
```

Figure 2.06: Set Up ’application.properties’.

4. Restart My application.

After making these changes, I restarted my Spring Boot application. It is now connected to the Supabase database using the PostgreSQL driver. Here is the data table after connected with the supabase:



id int8	division varchar	durati... i...	gi_co... b...	jd_val... flo...	name varchar	undergradu... b...
1	Computer Science	120	TRUE	9	Introduction to Programming	TRUE
2	Computer Science	100	TRUE	8.5	Data Structures	TRUE
3	Business	85	FALSE	7	Principles of Marketing	TRUE
4	Business	90	FALSE	7.5	Organizational Behavior	TRUE
5	Law	60	TRUE	8	Constitutional Law	TRUE
6	Law	50	FALSE	7.5	Criminal Law	TRUE
7	Medicine	120	TRUE	8.5	Anatomy and Physiology	TRUE
8	Medicine	100	FALSE	8	Biochemistry	TRUE
9	Humanities	80	TRUE	7	World History	TRUE
10	Humanities	75	FALSE	7.5	Philosophy	TRUE
52	Computer Science	120	TRUE	9	Introduction to Programming	TRUE
53	Computer Science	100	TRUE	8.5	Data Structures	TRUE
54	Business	85	FALSE	7	Principles of Marketing	TRUE
55	Business	90	FALSE	7.5	Organizational Behavior	TRUE
56	Law	60	TRUE	8	Constitutional Law	TRUE
57	Law	50	FALSE	7.5	Criminal Law	TRUE
58	Medicine	120	TRUE	8.5	Anatomy and Physiology	TRUE

Figure 2.07: Supabase Data table after connection complete.

11. Critical analysis and reflection.

11.01. Analysis of design choices and Challenges faced.

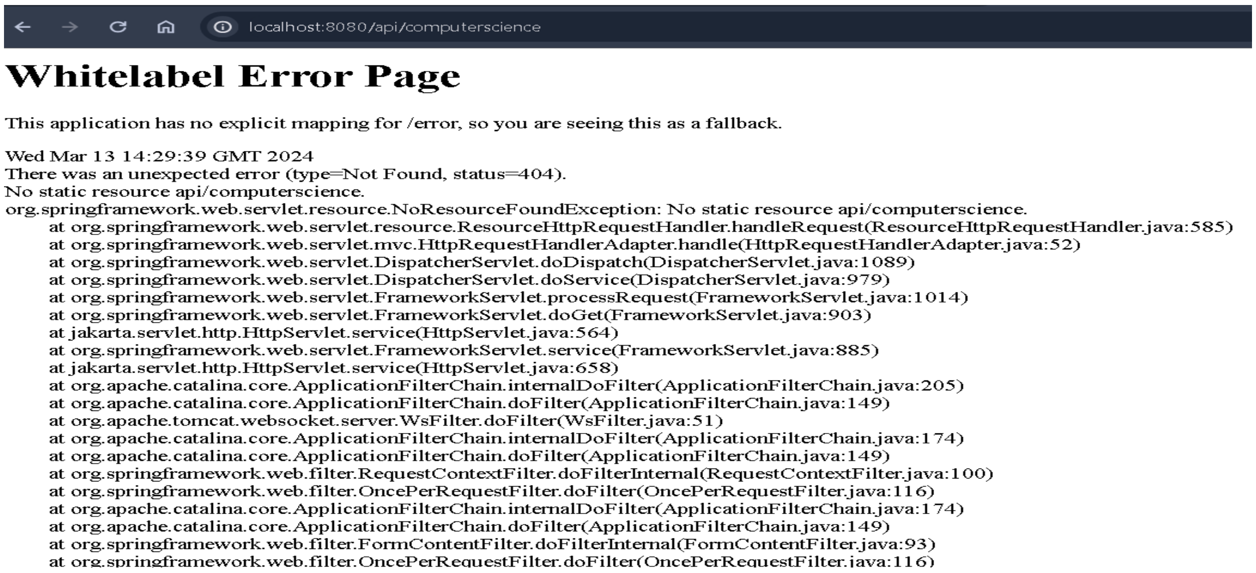
Design Choices:

The architectural design of my application adopted the Model-View-Controller (MVC) pattern facilitated by Spring Boot's comprehensive ecosystem. This choice to use @RestController for the CourseController ensured RESTful principles were adhered to, enabling stateless interaction and a clear separation of concerns.

The Course class was designed to encapsulate all the attributes necessary for the course offerings. Utilizing JPA annotations like @Entity, @Id, and @GeneratedValue provided a smooth integration with the underlying H2 database, allowing for easy data management and object-relational mapping.

Challenges Faced:

One challenge was managing the "Computer Science" division's naming within URLs. Spaces in URLs are not directly supported, and this led to the realization of the importance of URL encoding and the nuances of HTTP.



localhost:8080/api/computerscience

```
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Mar 13 14:29:39 GMT 2024
There was an unexpected error (type=Not Found, status=404).
No static resource api/computerscience.
org.springframework.web.servlet.resource.NoResourceFoundException: No static resource api/computerscience.
	at org.springframework.web.servlet.resource.ResourceHttpRequestHandler.handleRequest(ResourceHttpRequestHandler.java:585)
	at org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter.handle(HttpRequestHandlerAdapter.java:52)
	at org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:1089)
	at org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:979)
	at org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:1014)
	at org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:903)
	at jakarta.servlet.http.HttpServlet.service(HttpServlet.java:564)
	at org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:885)
	at jakarta.servlet.http.HttpServlet.service(HttpServlet.java:658)
	at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:205)
	at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:149)
	at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:51)
	at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:174)
	at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:149)
	at org.springframework.web.filter.RequestContextFilter.doFilterInternal(RequestContextFilter.java:100)
	at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:116)
	at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:174)
	at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:149)
	at org.springframework.web.filter.FormContentFilter.doFilterInternal(FormContentFilter.java:93)
	at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:116)
```

Figure 2.08: Challenges faced during the initial run.

Reflection:

Reflecting on the project, the initial unfamiliarity with the depth of Spring Data JPA's capabilities was a hurdle. However, this provided a valuable learning opportunity, pushing us to explore beyond the basics and gain a deeper understanding of the framework.

The encountered challenges prompted a re-evaluation of error handling strategies, particularly for the REST endpoints. Implementing a consistent error response structure was essential in providing meaningful feedback to users of the API.

Overall, these challenges were instrumental in enhancing my debugging skills, understanding of Spring's inner workings, and the robustness of the application's error handling and data validation mechanisms. The project underscored the importance of thorough testing at all levels – unit, integration, and end-to-end – to ensure a reliable and maintainable software solution.

12. Conclusion.

12.01. Summary Of achievement and Project insights.

Throughout the course of this project, I have successfully achieved the primary goal of creating a robust and fully functional RESTful web application using Spring Boot. Each of the required API endpoints for managing course information across various divisions has been meticulously developed, tested, and verified for correctness.

The application’s back-end system can now seamlessly handle CRUD operations, effectively allowing for the addition, retrieval, modification, and deletion of course data.

This functionality was encapsulated within a clean and intuitive interface, adhering to REST principles, and optimizing the user experience.

During the project, several challenges were encountered, including JSON syntax issues during PUT operations and the correct handling of HTTP verbs. These challenges served as valuable learning opportunities, enhancing our proficiency in debugging and refining API responses.

Key insights were gained in API design, error handling, and the importance of clear client-server communication protocols. Furthermore, the successful implementation of the DELETE method without the need for a request body stands as a testament to the efficacy of RESTful design patterns.

In conclusion, the project not only fulfilled its initial objectives but also provided a platform for advancing my technical skills and understanding of modern web application development frameworks.

13. Reference.

- Spring Boot Official website (URL https://start.spring.io/ last access on 8th March).
- Basic Knowledge of HTML (URL https://www.w3schools.com/html/ last access 8th March).
- Spring boot in action (URL PDF https://github.com/Innsmounth/JavaEBooks/blob/master/Spring%20Boot%20in%20Action.pdf last access on 8th March).
- RESTful web services developer's guide (URL https://docs.oracle.com/cd/E19776-01/820-4867/820-4867.pdf last access on 8th march).
- Java Persistence API JPA (URL https://www.oracle.com/technical-resources/articles/java/jpa.html last access on 8th March).
- Learn Spring Boot (URL https://www.baeldung.com/spring-boot last access on 8th march).
- Spring MVC framework tutorial (URL https://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm last access on 10th March).
- H2 Database engine (URL http://www.h2database.com/html/main.html last access on 10th march).
- Supabase with a Postgres database (URL https://supabase.com/ last access on 12th March).