SPRING BOOT STUDENT REGISTRATION SYSTEM WITH CRUD OPERATIONS

A Project Report

Submitted by:

ISHITA JENA (2141010031)

in partial fulfillment for the award of the degree

Of

BACHELOR OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Faculty Of Engineering And Technology, Institute of Technical Education and Research

SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY

Bhubaneswar, Odisha, India

December 2024

CERTIFICATE

This is to certify that the project report titled "**SPRING BOOT STUDENT REGISTRATION SYSTEM WITH CRUD OPERATIONS**" being submitted by **ISHITA JENA (CSE-M)** to the Institute of Technical Education and Research, Siksha 'O' Anusandhan (Deemed to be) University, Bhubaneswar for the partial fulfillment for the degree of Bachelor of Technology in Computer Science and Engineering is a record of original confide work carried out by them under my supervision and guidance. The project work, in my opinion, has reached the requisite standard fulfilling the requirements for the degree of Bachelor of Technology. The results contained in this project work have not been submitted in part or full to any other University or Institute for the award of any degree or diploma.

(Name and signature of the Project Supervisor)
Department of Computer Science and Engineering
Faculty of Engineering and Technology
Institute of Technical Education and Research
Siksha 'O' Anusandhan (Deemed to be) University

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my mentor, Prof. Amul Chourasia, for his guidance and constructive feedback throughout the development of this project, "**SPRING BOOT STUDENT REGISTRATION SYSTEM WITH CRUD OPERATIONS**". His support and expertise was invaluable in shaping the project and ensuring its successful completion.

I am also thankful to Siksha 'O' Anusandhan for providing the resources and infrastructure needed to carry out this work. A special thanks to my peers and family, whose encouragement and inputs motivated me to deliver my best efforts. This project is the result of their collective support and my learning journey.

Place: Siksha 'O' Anusandhan (ITER), Bhubaneswar

Signature Of Student

Date: 26-12-2024

DECLARATION

I declare that this written submission represents my ideas in my own words and where other's ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/fact/source in my submission. I understand that any violation of the above will cause for disciplinary action by the University and can also evoke penal action from the sources which have not been properly cited or from whom proper permission has not been taken when needed.

Signature Of Student With Registration Number
Date : 26-12-2024

REPORT APPROVAL

This project report titled "**SPRING BOOT STUDENT REGISTRATION SYSTEM WITH CRUD OPERATIONS**" submitted by **ISHITA JENA** is approved for the degree of *Bachelor of Technology in Computer Science and Engineering.*

Examiner(s)

Supervisor

Project Coordinator

PREFACE

This project, “Spring Boot Student Registration System With CRUD Operations”, is developed using Spring Boot, a powerful Java-based framework that simplifies the creation of robust, production-ready applications. The primary objective of this system is to demonstrate the effective implementation of CRUD (Create, Read, Update, Delete) operations for managing student data in a structured and user-friendly manner. This system incorporates various modern web development principles, including Model-View-Controller (MVC) architecture, dependency injection, and database integration, providing an exemplary demonstration of how these principles converge in real-world software applications.

The application is designed to maintain a database of student records, including attributes such as the student’s name, course, and fee structure. Built using the Spring Boot framework, the backend logic leverages the capabilities of Spring Data JPA for seamless interaction with a MySQL database. The ‘Student‘ entity class models the data, while the repository interface extends JpaRepository to facilitate efficient database operations such as saving, retrieving, updating, and deleting student records.

The application’s frontend, developed with Thymeleaf templates, provides an interactive user interface for administrators to manage student records. The homepage displays all registered students in a tabular format, with options to add new students, update existing records, or delete them. The form validation and responsiveness are enhanced by integrating Bootstrap, ensuring a visually appealing and user-friendly experience.

The system employs a layered architecture to promote separation of concerns. The service layer encapsulates the business logic, interfacing with the repository layer to perform database operations. The controller layer, implemented using Spring MVC, handles HTTP requests and directs them to the appropriate views or services. This design ensures modularity, maintainability, and scalability of the application.

Additional configuration is handled in the ‘application.properties‘ file, which includes database connection details, server port configuration, and Hibernate settings for managing the database schema. The Maven build tool is employed for dependency management, ensuring seamless integration of necessary libraries such as Spring Boot Starter Web, Thymeleaf, and MySQL Connector.

This project showcases Spring Boot’s capabilities in building scalable, maintainable web applications. The Student Registration System serves as an accessible introduction to Java-based web development, focusing on simplicity and best practices like modular design, JPA, and Thymeleaf. It’s both a functional app and a valuable learning resource for aspiring developers.

INDIVIDUAL CONTRIBUTIONS

Ishita Jena	Back-end (Coding), Documentation
Pratik Swarup Mishra	Back-end (Coding), Documentation
Soumya Shree Dash	Front-end (HTML, CSS, Bootstrap) Documentation
Gayatri Sahu	Front-end (HTML, CSS, Bootstrap), Documentation

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

The ‘Student Registration System’ is a comprehensive web application developed to simplify and streamline the process of managing student records. Built using Spring Boot, a widely recognized Java framework, this system serves as a practical example of leveraging modern software development methodologies to solve real-world challenges. The primary purpose of this application is to provide an intuitive and efficient platform for administrators to manage student data, including details such as names, courses, and fees. By implementing fundamental CRUD (Create, Read, Update, Delete) operations, the system demonstrates how a database-driven application can be built and maintained with ease.

Spring Boot is chosen as the backbone of this project due to its ability to simplify application development while promoting scalability and maintainability. The framework’s built-in features, including dependency injection and Spring Data JPA, enable seamless interaction with databases, while its support for Thymeleaf templates facilitates the creation of dynamic web interfaces. The system is further enhanced by the integration of Bootstrap, which ensures responsive design and a visually appealing user experience.

The core architecture of the system is grounded in the Model-View-Controller (MVC) design pattern. The model layer defines the ‘Student‘ entity, encapsulating essential attributes, while the controller layer manages HTTP requests and routes them to the appropriate views or services. The service layer implements business logic and acts as an intermediary between the controller and repository layers, which handle data persistence using MySQL. This layered architecture ensures modularity and simplifies maintenance and future enhancements.

The application begins with a homepage displaying all registered students in a tabular format, with options to add, update, or delete records. Each function is accompanied by forms for data entry and validation, ensuring accuracy and consistency in the database. The use of Spring Data JPA simplifies database operations by abstracting complex queries into straightforward method calls, while Hibernate manages database schema generation and ensures smooth data interactions.

The project not only provides a functional Student Registration System but also offers a valuable learning experience in modern web development. It covers key concepts like Maven dependency management, database configuration, Thymeleaf templates for dynamic content, and Bootstrap for styling. By integrating these technologies, the project demonstrates the practical application of Java-based web development principles.

SYSTEM ARCHITECTURE

The Student Registration System is structured using the Model-View-Controller (MVC) design pattern. This architecture divides the application into distinct layers, where each layer plays a crucial role in the application's functionality, contributing to its overall efficiency and ease of use. Below is a detailed explanation of each layer:

Model Layer

The model layer represents the core business entities of the application, encapsulating the data structure and behavior. In the context of the Student Registration System, the Student class serves as the primary model.

Attributes:

- The `Student` class includes attributes such as `id`, `name`, `course`, and `fee`. These attributes correspond to columns in the database table, defining the structure of student records.

Annotations:

- `@Entity`: Marks the class as a JPA entity to be mapped to a database table.
- `@Id` and `@GeneratedValue`: Designate the primary key and enable automatic generation of unique IDs for each student.
- `@Column`: Used to map class attributes to specific database columns, with options to specify constraints such as `nullable` and `length`.

Repository Layer

The repository layer acts as the bridge between the application and the database, managing data access and persistence. It is implemented using Spring Data JPA, which simplifies CRUD operations.

Key Features:

- `StudentRepository` extends `JpaRepository`, inheriting methods like `save`, `findById`, `findAll`, `deleteById`, etc., for seamless database interactions.
- By leveraging Spring Data JPA, developers can avoid writing boilerplate code for SQL queries, as methods are dynamically generated based on method names.

Database Interaction:

- This layer interacts directly with the MySQL database configured in the application's properties file. Hibernate, as the underlying ORM tool, manages the schema creation and query execution.

Service Layer

The service layer contains the business logic of the application, acting as an intermediary between the controller and repository layers. It ensures that the core application logic is encapsulated and reusable.

Responsibilities:

- Fetching and manipulating student data from the repository.
- Implementing additional business rules, such as input validation or data formatting, before saving or retrieving records.
- Providing meaningful error messages or exceptions if operations fail.

Implementation:

- The `StudentService` class includes methods like `getAllStudents()`, `saveStudent(Student student)`, `getStudentById(Long id)`, and `deleteStudentById(Long id)`.
- These methods wrap repository operations to provide a consistent interface for the controller layer.

Controller Layer

The controller layer handles user requests, processes input, and determines the appropriate responses. It serves as the application's entry point and connects the service layer with the view.

Key Components:

- The `StudentController` class is annotated with `@Controller`, signifying its role in handling web requests.
- Methods are mapped to specific URLs using `@RequestMapping` or `@GetMapping` annotations.

Functionalities:

- Homepage Display: The method `showAllStudents()` retrieves student data from the service layer and passes it to the view for display.
- Form Handling: The `saveStudent()` method processes form submissions for adding or updating records, invoking service methods to perform these actions.

- Redirection: After completing an operation, the controller redirects users to appropriate views or pages.

View Layer

The view layer is responsible for presenting data to the user through a web interface. In the Student Registration System, Thymeleaf templates and Bootstrap are used to create responsive and user-friendly views.

Thymeleaf:

- Enables dynamic content rendering, such as displaying a list of students or prepopulating forms with existing data.
- Directives like `th:text`, `th:each`, and `th:action` make it easy to bind data and manage interactions.

Bootstrap:

- Ensures the views are visually appealing and responsive across different devices.
- Provides pre-designed components, such as tables, buttons, and forms, to enhance the user experience.

Templates:

- The homepage template (`students.html`) displays a tabular view of student records with options for adding, editing, or deleting entries.
- Forms for adding or updating students are styled for simplicity and clarity, minimizing user errors.

Interaction Flow

- **User Request:** A user action, such as accessing the homepage or submitting a form, triggers an HTTP request.
- **Controller Handling:** The controller processes the request, interacts with the service layer, and determines the appropriate view to render.
- **Business Logic:** The service layer applies any necessary logic or validation and interacts with the repository layer for data access.
- **Database Interaction:** The repository layer retrieves or modifies data in the database using JPA methods.
- **Response Generation:** The controller passes data to the view layer, which renders the final output to the user.

FUNCTIONALITIES

The Student Registration System provides multiple endpoints to handle core functionalities like viewing, adding, updating, and deleting student records. Below is a description of each endpoint and its role:

- **Homepage:** `/students`
  - Role: Displays a list of all registered students.
  - Controller Method: `showAllStudents()`.
  - Description: Fetches student data from the service layer and renders it in a tabular format. Includes options for editing and deleting records.
- **Add Student:** `/students/new`
  - Role: Displays a form to add a new student.
  - Controller Method: `showAddForm()`.
  - Description: Renders a form where users can input details for a new student record.
- **Save Student:** `/students/save`
  - Role: Processes form submissions for adding or updating a student.
  - Controller Method: `saveStudent(Student student)`.
  - Description: Saves the submitted student data via the service layer and redirects to the homepage.
- **Edit Student:** `/students/edit/id`
  - Role: Displays a form pre-filled with existing student details for editing.
  - Controller Method: `showEditForm(Long id)`.
  - Description: Fetches the student record by ID and passes it to the view for editing.
- **Delete Student:** `/students/delete/id`
  - Role: Deletes a student record based on the provided ID.
  - Controller Method: `deleteStudent(Long id)`.
  - Description: Removes the specified student from the database and redirects to the homepage.

TECHNOLOGIES USED

- **Spring Boot**
  - Acts as the backbone of the project, simplifying the development of the Student Registration System with pre-configured settings.
  - It handles dependency injection, ensuring seamless integration of various components.
  - Provides an environment for creating REST APIs to manage operations such as adding, retrieving, updating, and deleting student records.
- **Thymeleaf**
  - Provides a server-side template engine for rendering dynamic HTML content.
  - Facilitates the seamless integration of form submissions and data display within the registration system.
- **MySQL**
  - Serves as the database management system to store, retrieve, and manage student records, ensuring reliable data persistence and querying capabilities.
- **Spring Data JPA (Java Persistence API)**
  - Automatically maps Java objects to relational database tables, bridging the gap between the application and MySQL.
  - Offers built-in CRUD operations, reducing the need for boilerplate SQL queries.
  - Supports custom query creation for more advanced data operations.
- **Spring Boot Starter Web**
  - This dependency provides the necessary tools for creating and managing RESTful web services.
  - It ensures smooth communication between the client and the server by handling HTTP requests and responses.
  - Manages URL routing and supports JSON and XML serialization for data exchange.
  - Plays a crucial role in exposing the functionality of the application to users and external systems.
- **Spring Boot Starter Thymeleaf**

  - Integrates Thymeleaf with the Spring Boot application, supporting the development of interactive web pages.
  - Powers the HTML views displayed in the browser.

- **Spring Boot Starter Data JPA**
  - Includes Hibernate as the default ORM tool to manage data persistence.
  - Automates common database tasks such as inserting, updating, and deleting records.
  - Works with Spring Data JPA to provide repository support, enabling the application to interact with the database efficiently.
  - Facilitates advanced features like pagination, sorting, and custom queries for better data management.

- **MySQL Connector**
  - The MySQL Connector acts as a critical bridge between the Spring Boot application and the MySQL database.
  - Establishes a secure and optimized connection to execute database operations.
  - Handles authentication and ensures that queries from the application are executed seamlessly on the database side.

- **Spring Boot DevTools**
  - Spring Boot DevTools enhances the development experience by enabling features like hot-reloading and automatic application restarts.
  - Reflects changes made to the codebase immediately, eliminating the need to restart the server manually.

- **HTML, CSS, and Bootstrap**
  - HTML forms the skeleton of the application, structuring content such as student lists, forms, and navigation elements.
  - CSS styles the application, ensuring consistent and appealing design elements like colors, fonts, and layouts.
  - Bootstrap enhances responsiveness, ensuring the application is accessible and user-friendly on devices of all sizes.

IMPLEMENTATION

StudentCrudApplication.java

- Entry point for the Spring Boot application.
- Consists of the `@SpringBootApplication` meta-annotation that combines three key Spring annotations:
  - `@Configuration`: Marks the class as a source of bean definitions for the Spring container.
  - `@EnableAutoConfiguration`: Enables Spring Boot's auto-configuration feature to configure the application based on dependencies and properties.
  - `@ComponentScan`: Automatically scans for components (e.g., `@Controller`, `@Service`, `@Repository`) within the package and its sub-packages.
- This annotation makes the application ready to run as a Spring Boot application without requiring manual configuration

Student.java

- Domain model that represents a student entity in the application. It is designed to map to a database table and encapsulate the attributes and behaviors of a student.
- `@Entity` annotation indicates that the `Student` class is a JPA entity and should map to a table in the database.
- It encapsulates student data (`id`, `studentname`, `course`, `fee`) and defines getters and setters to interact with this data. Each field corresponds to a column in `student` table.
- `id` is annotaed with `@Id` to specify that this field is the unique identifier (primary key) and `@GeneratedValue(strategy= GenerationType.IDENTITY)` to indicate that the value will be auto-generated by the database.

StudentRepository.java

- Repository interface in the Spring Data JPA framework. It is responsible for interacting with the database to perform CRUD (Create, Read, Update, Delete) operations on the `Student` entity.
- `@Repository` annotation marks the interface as a Spring Repository.

- By extending `JpaRepository`, the `StudentRepository` interface inherits many ready-to-use methods for interacting with the database

StudentService.java

- Acts as an intermediary between the controller layer and the repository layer, encapsulating the business logic and ensuring that database operations are executed correctly.
- `@Service` annotation marks this class as a service layer component.
- `@Autowired` annotation is used to inject the `StudentRepository` dependency into this service class which allows the service to use the methods provided by `StudentRepository` for interacting with the database
- Provides methods to:
  - Retrieve all students: Calls the `findAll()` method of `JpaRepository`, which returns a list of all Student entities.
  - Save or update a student: Calls the `save()` method of `JpaRepository` to add a new student or update an existing student's information
  - Get a student by ID: Calls the `findById()` method of `JpaRepository` to find the student with the specified id
  - Delete a student by ID: Calls the `deleteById()` method of `JpaRepository`, which removes the student with the given id from the database.

StudentController.java

- Manages HTTP requests and responses, serving as a bridge between the user interface (views) and the business logic (service layer). It defines endpoints for performing CRUD operations on `Student` records.
- Annotations used in the class:
  - `@Controller`: Marks this class as a Spring MVC controller to handle web requests.
  - `@RequestMapping`: Maps HTTP requests to specific handler methods.
  - `@GetMapping`: Maps HTTP GET requests to specific methods.
  - `@ModelAttribute`: Binds request data to a model object.
  - `@PathVariable`: Extracts values from the URL path.

- It provides end-to-end flow.
  - Display All Students: User visits `/`, `viewHomePage()` fetches all students, and returns the index view with a list of students.
  - Add a New Student: User clicks “Add New”, `add()` opens the new view, user submits the form, `saveStudent()` saves data, and redirects to `/`.
  - Edit a Student: User clicks “Edit”, `showEditStudentPage()` fetches the student, opens the pre-filled form, user updates data, and `saveStudent()` saves updates.
  - Delete a Student: User clicks “Delete”, `deleteStudent()` deletes the student, and redirects to `/`.

index.html

- Displays the list of all students in the table.
- Provides “Edit” and “Delete” options for each student.
- Includes a button to add a new student.

new.html

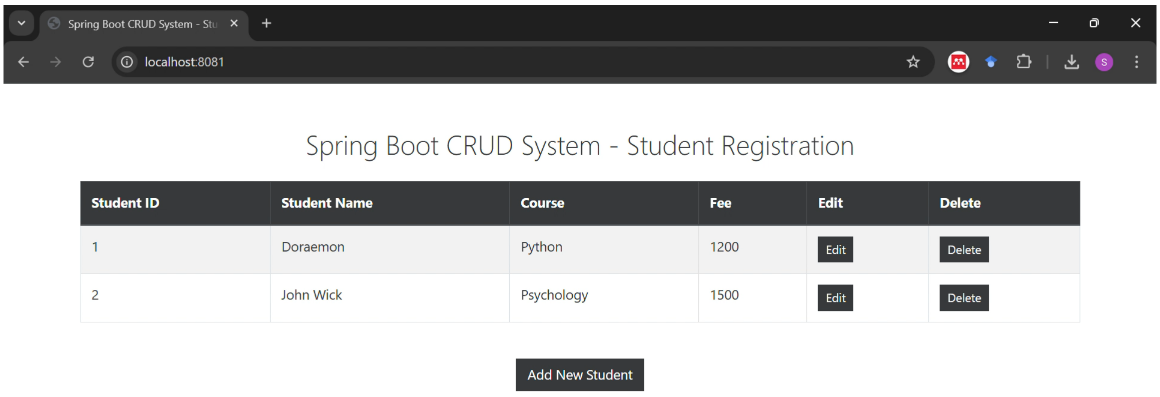
- Displays a form for creating a new or editing an existing student record.
- When inserting a new student, a blank form is displayed.
- When editing the details of an existing student, a pre-filled form containing the student’s details is displayed.

application.properties

- Specifies the server port, database and Java Persistence API (JPA) configurations.
- Spring Boot application runs on port 8081.
- All database information like the database name, username, password is extracted from here.
- Determines how Hibernate handles the database schema

RESULTS & INTERPRETATION

Starting The Application



Spring Boot CRUD System - Student Registration

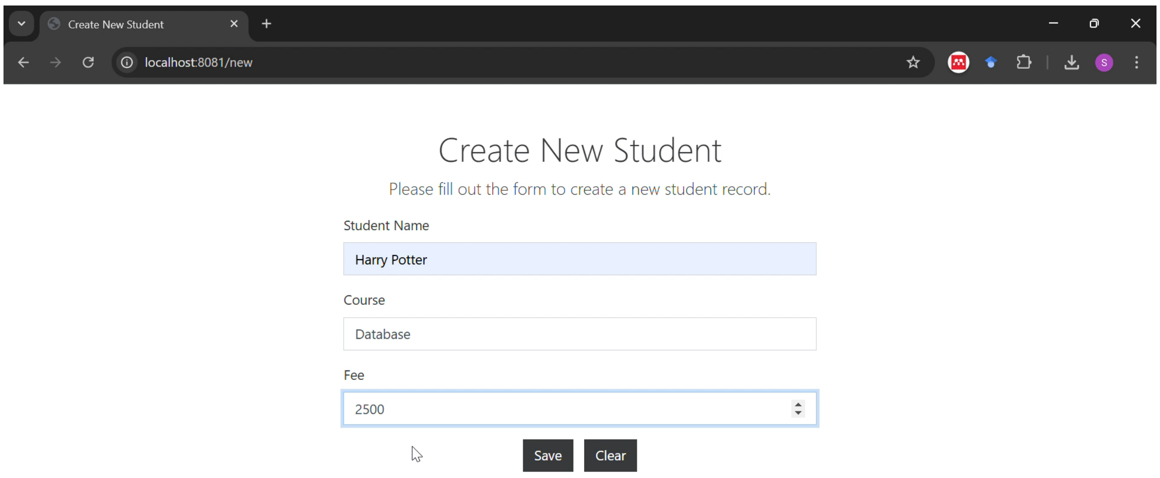
Student ID	Student Name	Course	Fee	Edit	Delete
1	Doraemon	Python	1200	Edit	Delete
2	John Wick	Psychology	1500	Edit	Delete

- **Spring Boot Application Initialization**
  - When the `StudentCrudApplication.java` is run, `@SpringBootApplication` combines three key annotations: `@Configuration`, `@EnableAutoConfiguration`, `@ComponentScan`.
  - The `SpringApplication.run(StudentCrudApplication.class, args)` method initializes the Spring application context, which starts the embedded server (e.g., Tomcat by default) and sets up all the beans, configurations, and components.
- **Auto-Configuration and Dependency Injection**
  - Spring Boot detects dependencies (e.g., JPA, Thymeleaf, MySQL connector) and configures necessary beans automatically.
- **Database Configuration**
  - Spring Boot reads the database connection details from `application.properties`.
- **Entity and Repository Setup**
  - The `@Entity` annotation maps the `Student` class to a database table, and `StudentRepository` extends `JpaRepository` for automatic CRUD operations.
  - Spring Data JPA scans and creates the `StudentRepository` implementation at runtime.
- **Controller and View Resolution**
  - The `@Controller` annotation marks `StudentController`, which is detected by Spring's component scanning to handle HTTP requests and return views.

- **Server Startup**
  - The embedded Tomcat server starts, as configured by the `spring-boot-starter-web` dependency and the server listens on port 8081.
- **Thymeleaf Integration**
  - The `viewHomePage` method fetches the student list via `StudentService` and passes it to the model, which is rendered by `index.html` using Thymeleaf.
- **User Interaction**
  - Accessing `http://localhost:8081/` triggers the `viewHomePage` method, which fetches student data via `StudentService` and renders it in a table on `index.html` using Thymeleaf.

Adding New Student

- **Frontend Interaction**
  - On the `index.html` page, when the "Add New Student" is clicked, it loads the `new.html` which contains a form for adding a new student.



Create New Student

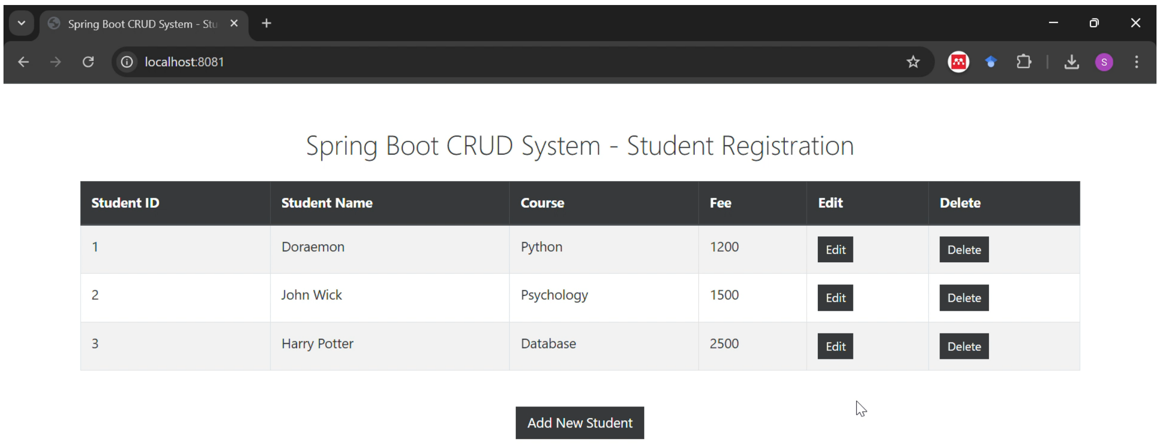
Please fill out the form to create a new student record.

Student Name	Harry Potter
Course	Database
Fee	2500

- **Controller Layer**
  - The `@GetMapping("/new")` method in the controller serves the form and when the form is submitted, the `@PostMapping("/save")` method processes the request.
- **Service Layer**
  - The `save` method in the `StudentService.java` interacts with the repository
- **Repository Layer**

- The `StudentRepository` interface, which extends `JpaRepository`, provides the `save` method.
- JPA executes an `INSERT` SQL query to add the new record to the database. `INSERT INTO student (course, fee, name) VALUES (?, ?, ?);`

- **Post-Save Flow**
  - After saving, the application redirects to `/`, which triggers the `viewHomePage` method in the `StudentController`.
  - The updated list of students is fetched from the database, now including the newly added record.
  - The `index.html` page renders the updated list of students, showing the new student.



Spring Boot CRUD System - Student Registration

Student ID	Student Name	Course	Fee	Edit	Delete
1	Doraemon	Python	1200	Edit	Delete
2	John Wick	Psychology	1500	Edit	Delete
3	Harry Potter	Database	2500	Edit	Delete

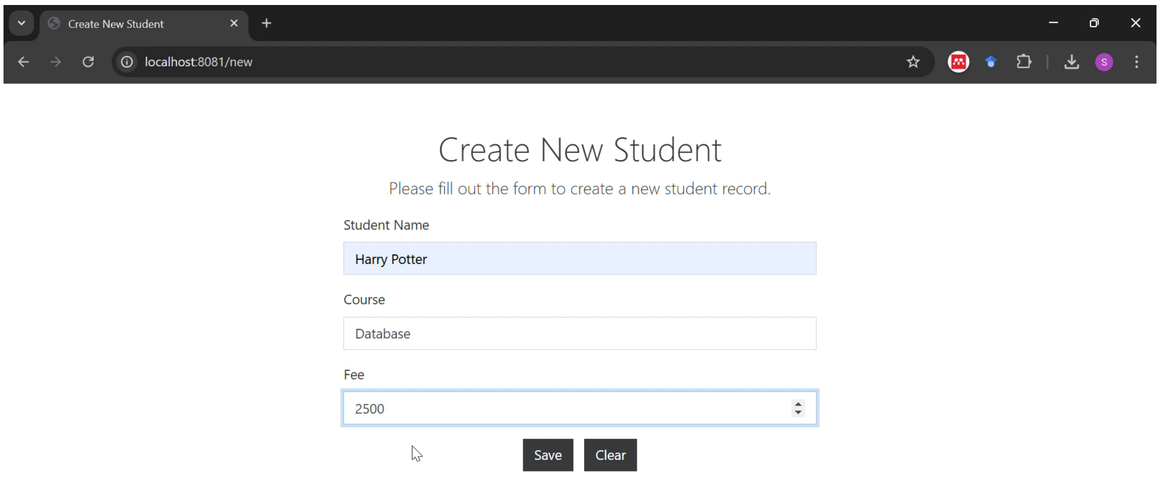
Edit Existing Student

- **Frontend Interaction**
  - On the `index.html` page, when the "Edit" is clicked, Thymeleaf syntax (`th:href`) to dynamically generate a URL like `/edit/id`, where `id` is the `id` of the student.
- **Controller Layer**
  - The `@RequestMapping` method in the `StudentController.java` is triggered.
  - The `id` in the URL is extracted and mapped to the method parameter `id` using `@PathVariable(name = "id")`.
  - The `StudentService` fetches the record from the database using the `get` method.

  - The `StudentRepository` interface, which extends `JpaRepository`, performs the database query (`SELECT * FROM student WHERE id = ?`).
  - The retrieved `Student` object is added to the `ModelAndView` object with the view name `new`.

- **Service Layer**
  - The service calls the repository method `findById`.
  - The repository interacts with the database via JPA to fetch the record with the corresponding `id`.

- **View Rendering**
  - The `new.html` page is rendered with the `Student` object pre-filled into the form fields.



Create New Student

Please fill out the form to create a new student record.

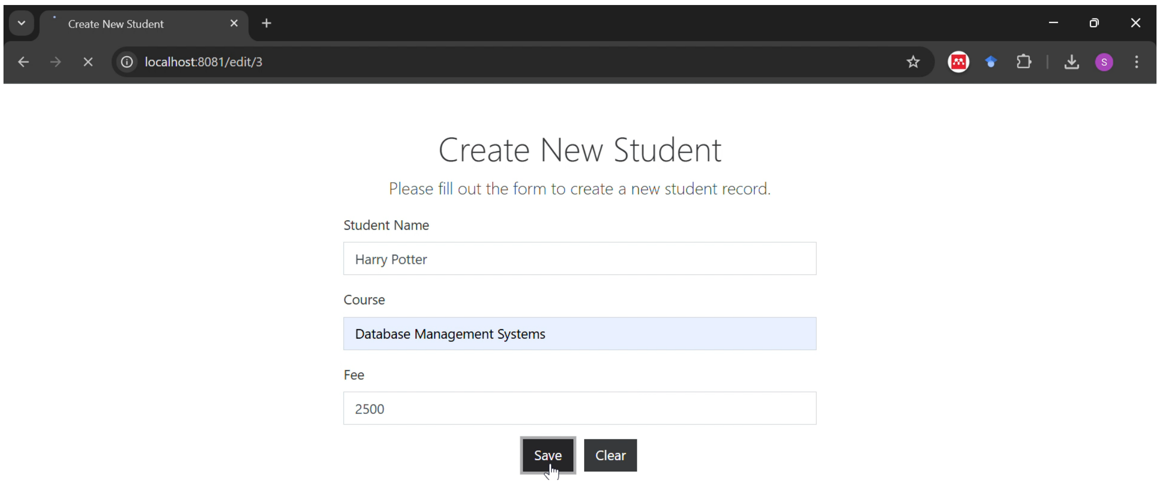
Student Name: Harry Potter

Course: Database

Fee: 2500

- **Updating Record**
  - After making changes, when the form is submitted, the data is sent via a POST request to `/save`.
  - The `saveStudent` method in the `StudentController` is triggered.
  - The `@ModelAttribute` maps form data to the `Student` object.
  - The `Student` object now contains updated information along with the existing id.
  - The `save` method in the `StudentService` is called.
  - If the `Student` object has an `id` that exists in the database, JPA treats it as an update rather than an insert.
  - The repository uses the `save` method to execute an `UPDATE` query. (`UPDATE student SET studentname = ?, course = ?, fee = ?  WHERE id = ?;`)

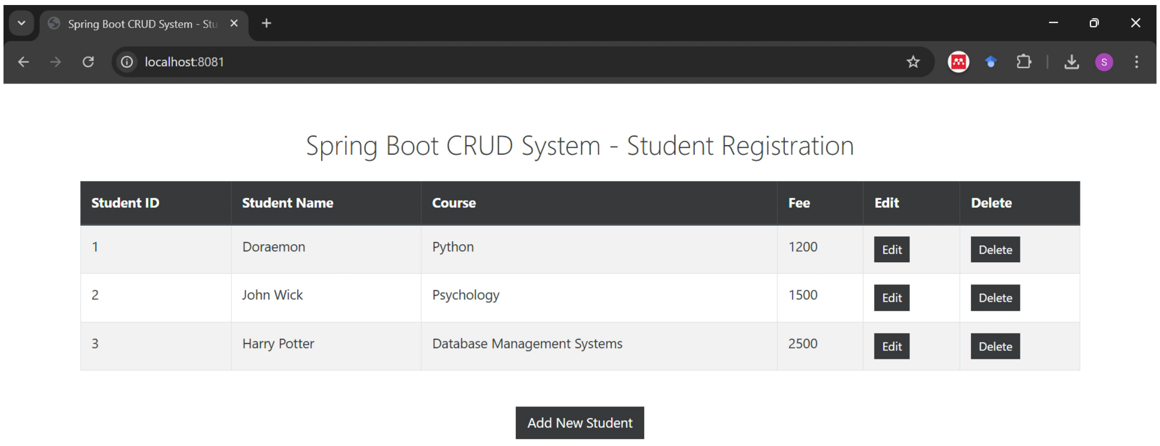
- After the `saveStudent` method completes, a redirect occurs to `/`, reloading the updated list of students.
- The `viewHomePage` method in the `StudentController` retrieves all students again from the database, reflecting the changes.



Create New Student

Please fill out the form to create a new student record.

Student Name	Harry Potter
Course	Database Management Systems
Fee	2500



Spring Boot CRUD System - Student Registration

Student ID	Student Name	Course	Fee	Edit	Delete
1	Doraemon	Python	1200	Edit	Delete
2	John Wick	Psychology	1500	Edit	Delete
3	Harry Potter	Database Management Systems	2500	Edit	Delete

Delete Existing Student

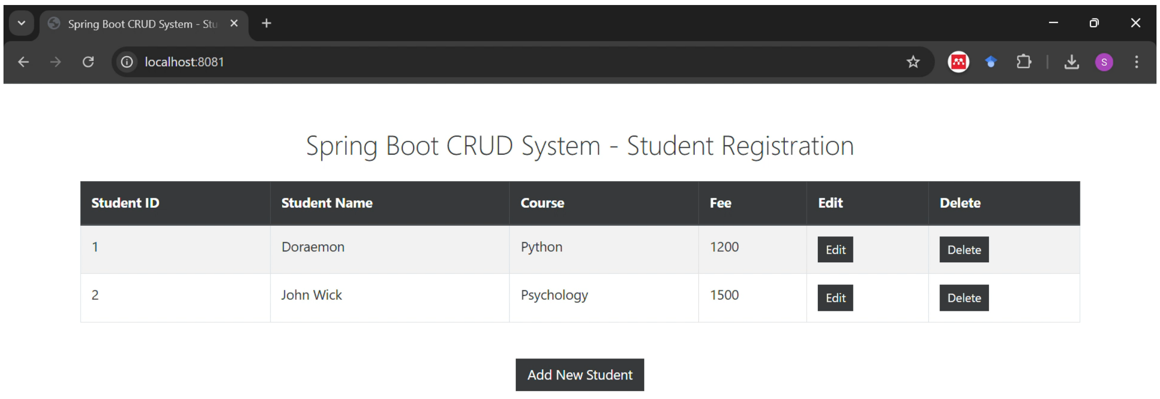
- **FrontEnd Interaction**
  - On the `index.html` page, when the "Delete" is clicked, Thymeleaf syntax (`th:href`) to dynamically generate a URL like `/delete/id`, where `id` is the `id` of the student.
  - The button uses Thymeleaf syntax (`th:href`) to dynamically generate a URL like `/delete/{id}`, where {id} is the student's ID.
- **Controller Layer:**
  - The `@RequestMapping("/delete/id")` method in the `StudentController` handles the request.

- The id in the URL is captured with `@PathVariable(name = "id")` and passed to the `deleteStudent` method.
- The `service.delete(id)` method is called to delete the record.
- After deletion, the method redirects to `/`, refreshing the list of students.

- **Service Layer:**
  - The `delete` method in the `StudentService` interacts with the repository.
  - The service layer calls `repo.deleteById(id)`, which generates a DELETE query for the database.

- **Repository Layer:**
  - The `StudentRepository` interface, which extends `JpaRepository`, provides the `deleteById` method.
  - JPA executes a SQL DELETE query to remove the record with the given ID from the database.`DELETE FROM student WHERE id = ?;`

- **Post-Deletion Flow:**
  - After deletion, the application redirects to `/`, triggering the `viewHomePage` method in the `StudentController`.
  - The updated list of students is fetched from the database, excluding the deleted record.
  - The `index.html` page renders the updated list of students, no longer including the deleted record.



Spring Boot CRUD System - Student Registration

Student ID	Student Name	Course	Fee	Edit	Delete
1	Doraemon	Python	1200	Edit	Delete
2	John Wick	Psychology	1500	Edit	Delete

Add New Student

CONCLUSIONS

The Student Registration System has immense potential for future improvements to make it more robust, user-friendly, and feature-rich. One of the key enhancements could be the implementation of Role-Based Access Control (RBAC). This would allow the system to define roles such as Admin, Teacher, and Student, where each role has specific permissions. For instance, only admins might have the ability to delete student records, while teachers could manage grades and students could view their details. This feature would enhance both functionality and security, ensuring that sensitive operations are performed only by authorized users.

Another significant improvement would be the addition of advanced search and filtering options. Users could search for students based on various parameters such as name, age, course, or registration date, with the ability to use multiple criteria simultaneously. This would save time and improve efficiency, especially when handling a large dataset. Autocomplete suggestions and intelligent search results could further enhance the user experience.

Integrating the system with third-party tools and exposing RESTful APIs would allow seamless communication with external applications, such as Learning Management Systems (LMS) or Attendance Management Systems. These APIs would enable the export and import of data in formats like CSV or JSON, making it easier to collaborate with other software. Moreover, enabling bulk operations such as registering multiple students using file uploads or performing batch updates and deletions would streamline the management of large amounts of data.

The system could also benefit from a notification feature that sends alerts via email or SMS for critical updates, such as successful registrations, upcoming deadlines, or changes in the student database. Utilizing services like Twilio or SendGrid for this functionality would ensure reliable delivery of notifications. Additionally, an analytics dashboard could provide insightful data visualization, such as the total number of registrations, course-wise student distribution, and monthly trends. Such analytics would be valuable for administrators in making data-driven decisions.

Integrating third-party authentication providers like Google, Facebook, or LinkedIn would simplify the login process for users while maintaining a high level of security. Additionally, implementing audit trails and logging mechanisms would help administrators monitor changes made to the database and troubleshoot issues efficiently.

In conclusion, these enhancements would transform the Student Registration System into a scalable, user-centric, and highly efficient solution, making it better equipped to meet the needs of educational institutions and students.

REFERENCES

[1] Guides, “Student management system spring boot project — spring boot thymeleaf web application full course,” May 2021.

[2] Medium, “Student registration application with spring boot, postgresql and docker,” Sep 2023

[3] T. Funny, “Springboot simple project step by step,” Sep 2020.

[4] T. Funny, “Spring boot thymeleaf crud application - tutusfunny,” Jun 2023.

[5] W. b. baeldung, “The registration process with spring security,” Aug 2024.

APPENDICES

APPENDIX 1

Source Code

StudentCrudApplication.java

```
package com.example.StudentCrud;
import org.springframework.boot.SpringApplication;
import
   org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class StudentCrudApplication {
    public static void main(String[] args) {
                SpringApplication.run(StudentCrudApplication.
        class, args);
    }
}
```

Student.java

```
package com.example.StudentCrud.domain;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Student {
    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    private Long id;
    private String studentname;
    private String course;
    private int fee;

    public Student() {}

    public Student(String studentname, String course, int
       fee) {
        this.studentname = studentname;
        this.course = course;
        this.fee = fee;
    }
```

Student.java

```java
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getStudentname() {
        return studentname;
    }

    public void setStudentname(String studentname) {
        this.studentname = studentname;
    }

    public String getCourse() {
        return course;
    }

    public void setCourse(String course) {
        this.course = course;
    }

    public int getFee() {
        return fee;
    }

    public void setFee(int fee) {
        this.fee = fee;
    }
}
```

StudentRepository.java

```java
package com.example.StudentCrud.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.example.StudentCrud.domain.Student;

@Repository
public interface StudentRepository extends
   JpaRepository<Student, Long> {}
```

StudentService.java

```
package com.example.StudentCrud.service;
import java.util.List;
import
   org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.example.StudentCrud.domain.Student;
import com.example.StudentCrud.repository.StudentRepository;

@Service
public class StudentService {

    @Autowired
    private StudentRepository repo;

    public List<Student> listAll() {
        return repo.findAll();
    }

    public void save(Student std) {
        repo.save(std);
    }

    public Student get(long id) {
        return repo.findById(id).get();
    }

    public void delete(long id) {
        repo.deleteById(id);
    }
}
```

StudentController.java

```
package com.example.StudentCrud.controller;
import org.springframework.stereotype.Controller;
import java.util.List;
import
   org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
```

StudentController.java

```java
import
   org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import
   org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import com.example.StudentCrud.domain.Student;
import com.example.StudentCrud.service.StudentService;

@Controller
public class StudentController {

    @Autowired
    private StudentService service;

    @GetMapping("/")
    public String viewHomePage(Model model) {
        List<Student> liststudent = service.listAll();
        model.addAttribute("liststudent", liststudent);
        System.out.print("Get / ");
        return "index";
    }

    @GetMapping("/new")
    public String add(Model model) {
        model.addAttribute("student", new Student());
        return "new";
    }

    @RequestMapping(value = "/save", method =
       RequestMethod.POST)
    public String saveStudent(@ModelAttribute("student")
       Student std) {
        service.save(std);
        return "redirect:/";
    }

    @RequestMapping("/edit/{id}")
    public ModelAndView
       showEditStudentPage(@PathVariable(name = "id") int
       id) {
        ModelAndView mav = new ModelAndView("new");
        Student std = service.get(id);
```

StudentController.java

```
            mav.addObject("student", std);
            return mav;
        }

        @RequestMapping("/delete/{id}")
        public String deletestudent(@PathVariable(name = "id")
           int id) {
            service.delete(id);
            return "redirect:/";
        }
}
```

application.properties

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/adpproject
spring.datasource.username=root
spring.datasource.password=ishitajena
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.thymeleaf.prefix=classpath:/templates/
server.port=8081
```

Note: For the complete source code and implementation details, please visit the following link: https://github.com/Ishita-Jena/ADP_Project.git

APPENDIX 2

Student Table

Student ID	**Student Name**	**Course**	**Fee**
1	Doraemon	Java	1200
2	John Wick	Psychology	1500

REFLECTION OF THE TEAM MEMBERS ON THE PROJECT

What I Learned As A Team

As a team, I learned the importance of collaboration and effective communication in delivering a project like the “Spring Boot Student Registration System With CRUD Operations. Dividing tasks based on individual strengths allowed me to work efficiently while ensuring quality at every stage. I gained insights into problem-solving, debugging, and integrating various technologies to create a cohesive application. The experience also reinforced the value of mutual support and adaptability when encountering challenges during development.

What I Learned As A Member

As an individual team member, I deepened my understanding of full-stack development, including database management with MySQL, front-end design with Thymeleaf, and backend logic with Spring Boot. Working on this project sharpened my skills in troubleshooting, implementing CRUD operations, and ensuring user-friendly functionality. Additionally, I learned the significance of documenting processes and maintaining clean, modular code for better collaboration.

Strengths And Weaknesses Of My Design Process

In reflecting on the design process, a key strength was the systematic approach to planning and implementing each layer of the application architecture. Our emphasis on understanding user requirements and translating them into a functional design proved effective. However, one weakness was underestimating the time needed for testing and refining the user interface, which led to last-minute adjustments. Moving forward, we aim to allocate more time for iterative testing and gather early feedback to ensure smoother project delivery.