STUDENT REGISTARATION

StudentR.jsp

```
<link rel="stylesheet" href="${pageContext.request.contextPath}/pages/student.css">
```

write in .jsp which is html file to link with css file which is seperatly save as .css extension

${pageContext.request.contextPath} → This ensures the correct path is used dynamically, even if the application is deployed in a subdirectory.
/pages/student.css → Since your CSS file is inside src/main/webapp/pages/, this correctly points to it.

pom.xml

1) in pom.xml I used java version 17 and stable spring boot version 2.7.1

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.7.1</version>
        <relativePath/>
    </parent>
```

2) we have to add tomcat jasper dependency version 9.0.64
   - go on chrom and search maven repository go to website and search tomcat jasper and click on 9.0.64 copy depenedency and paste it in pom.xml

```
<dependency>
  <groupId>org.apache.tomcat</groupId>
  <artifactId>tomcat-jasper</artifactId>
  <version>9.0.64</version>
</dependency>
```

What is tomcat-jasper?

- Jasper is **Apache Tomcat's JSP Engine**, responsible for **compiling JSP files into servlets**.

- **By default**, Spring Boot does **not** support JSP out of the box because it uses **embedded Tomcat**, which doesn't include the Jasper compiler.
- Adding this dependency allows Spring Boot's embedded Tomcat to **process JSP files** correctly.

3) Need to add jpa dependency

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

if you **don't** add the JPA dependency, Spring Boot **won't recognize** the @Entity annotation, and your application will throw an error.

Spring Boot needs the **JPA dependency** to process **Entities, Tables, and Repositories**.

The **JPA (Java Persistence API)** dependency in Spring Boot allows you to interact with a database using **Java objects instead of SQL queries**.

4) Need to add my sql dependency for database connection

```
<dependency>
      <groupId>mysql</groupId>
       <artifactId>mysql-connector-java</artifactId>
</dependency>
```

```
package com.example.demo;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.RequestMapping;

@Controller

public class studentController {

       @Autowired

       StudentService ss; //it is service class and its object injected

       @RequestMapping("/")

       public String Default() {

      return "studentR";

       }

       @RequestMapping("/savedata")

       public String Savedata(@ModelAttribute studentEntity student, Model  model) {

              ss.save(student);

               model.addAttribute("message", "Student registered successfully!");

              return "Welcome";

}

}
```

StudentController.java

The**StudentController.java** file is responsible for handling HTTP requests and interacting with the service layer to process student registration.

@Autowired: Enables automatic dependency injection.

What Happens Internally?

- Spring Boot **automatically finds an instance** of StudentService and assigns it to ss.
- No need to manually create an object (new StudentService()).

@Controller: Defines this class as a Spring MVC controller.

What Happens Internally?

- **Spring Boot registers this class** as a controller.
- It tells Spring, **"This class will handle user requests (like form submission, login, etc.)."**
- Without @Controller, this class would be treated as a normal Java class, not a web handler.

Model: Used to pass data from the controller to the view (JSP file).

@ModelAttribute: Binds form data to a Java object.

What Happens Internally?

- When a form is submitted, Spring Boot **automatically fills the studentEntity object** with the form data.
- If a form field has:

<input type="text" name="firstName">

  - The value entered is **automatically mapped to** student.firstName.

@RequestMapping: Maps HTTP requests to handler methods.

What Happens Internally?

- When the user submits a form to /savedata, Spring **calls this method automatically**.

- It processes the request and returns a view (Login.jsp).

```
@RequestMapping("/")

public String Default() {

    return "studentR";

}
```

@RequestMapping("/"): Maps the root URL (localhost:8080/) to this method.

Returns "studentR", which refers to studentR.jsp (registration page).

This displays the student registration form when the user visits the home page.

```
@RequestMapping("/savedata")

public String Savedata(@ModelAttribute studentEntity student, Model model) {

    ss.save(student);

    model.addAttribute("message", "Student registered successfully!");

    return "Login";

}
```

@RequestMapping("/savedata") → This method runs when a form is submitted to /savedata.

@ModelAttribute studentEntity student:

Automatically binds form data (from studentR.jsp) to the studentEntity object.

<input type="text" name="firstName">

The entered first name is mapped to student.firstName.

Calling the Service Layer (ss.save(student))

ss.save(student);

Calls the save() method from StudentService, which saves student data into MySQL.

Adding a Success Message

model.addAttribute("message", "Student registered successfully!");

The message is passed to the Login.jsp page

Interface StudentRepository.java

```
package com.example.demo;

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

import org.springframework.stereotype.Repository;

@Repository

public interface StudentRepository extendsJpaRepository<studentEntity,Long>{

}
```

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

- Imports **JpaRepository**, which is a **Spring Data JPA interface**.
- It gives you ready-made methods like:
  - save(), findById(), findAll(), deleteById() — so you don't have to write SQL code manually!

import org.springframework.stereotype.Repository;

- Imports the @Repository annotation.
- Tells Spring Boot:

  👉 "Hey, this is a Repository layer – responsible for interacting with the database."

@Repository

- This annotation **registers the interface as a Spring bean**.

Spring handles database exceptions automatically when this is used.

public interface StudentRepository extendsJpaRepository<studentEntity, Long>

Here's what it does:

- **StudentRepository** is a custom interface for interacting with your database.
- It **extends JpaRepository** — this is where the magic happens!

extends JpaRepository<studentEntity, Long>

<studentEntity, Long>

- studentEntity: The name of your Entity class — this tells JPA what type of object it will work with (like a Student table).
- Long: The data type of the primary key (ID) field in your entity.

Interface StudentService.java

```
package com.example.demo;

public interface StudentService {

   void save(studentEntity model);

}
```

This is a **Service Layer Interface** in a Spring Boot application.

In a Spring project, we usually follow **3-layer architecture**:

Controller → Service → Repository → Database

This interface belongs to the **Service Layer**. It's like a **middle layer** between your **controller (UI logic)** and the **repository (DB logic)**.

public interface StudentService {

- This declares an **interface** named StudentService.
- An interface is like a **contract** — it defines **what** the class should do, not **how** it does it.
- Later, another class (like StudentServiceImpl) will **implement** this interface.

void save(studentEntity model);

- A method declaration named save()
- void: It returns **nothing**.
- studentEntity model: This method accepts a studentEntity object as input.

💡 So, any class that implements StudentService must **write the logic for saving a studentEntity object.**

StudServiceImplement.java

```
package com.example.demo;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;



@Service

public class StudServiceImplement implements StudentService {

	@Autowired

	StudentRepository studentrepository;



	public void save(studentEntity model) {

		studentrepository.save(model);

		} }
```

@Service

- This marks the class as a **Service Component**.
- It is **automatically detected** and managed by Spring's container (@ComponentScan).

public class StudServiceImplement implements StudentService {

- A **concrete class** that implements your earlier StudentService interface.
- This is where the **actual logic** is written for whatever methods were declared in the interface.

@Autowired

- This injects the StudentRepository bean into this class **automatically**.
- So you can use it without writing new StudentRepository().

StudentRepository studentrepository;

- This is the **interface** that handles **database operations** like save(), findById(), etc.

---

public void save(studentEntity model) {

- This method is the **implementation** of the method declared in your interface:

void save(studentEntity model);

---

studentrepository.save(model);

- This line calls the save() method from JpaRepository.
- It saves the studentEntity object to the **database**.

Spring Data JPA takes care of all the SQL behind the scenes.

Why We Create Object of Repository Class (studentrepository)

@Autowired

StudentRepository studentrepository;

Because we want to use database methods like:

- save()
- findById()
- findAll()
- deleteById()

❓ But Why Not Write Logic Directly in Service?

Because we follow **Layered Architecture** in Spring Boot:

Controller → Service → Repository → Database

The **Repository Layer** contains **all the database logic**.

- It interacts directly with the **MySQL database**.

- So if the **Service Layer** wants to save, update, or get data from the DB, 👉 it **needs to talk to the Repository Layer**

Application.properties

```
spring.application.name=StudentRegistration

spring.mvc.view.prefix=/pages/

spring.mvc.view.suffix=.jsp

# Database Configuration

spring.datasource.url=jdbc:mysql://localhost:3306/StudentRegistration

spring.datasource.username=root

spring.datasource.password=root@79

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA (Hibernate) Settings

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true
```

spring.mvc.view.prefix=/pages/

spring.mvc.view.suffix=.jsp

These tell Spring Boot **where to find your JSP files** (your view pages).

Database Configuration

spring.datasource.url=jdbc:mysql://localhost:3306/StudentRegistration

- This is your **database connection URL**.
- Connects to a **MySQL database** named StudentRegistration running on your local machine.

spring.datasource.username=root

- Username for MySQL (commonly root by default).

spring.datasource.password=root@79

- Password to connect to your MySQL database.
- ⚠ Be cautious: never share passwords publicly.

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

- The **JDBC driver class** for MySQL.
- This tells Spring which driver to use for the connection.

JPA (Hibernate) Settings

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

- This sets the **dialect**, i.e., how Hibernate will talk to the MySQL 8 database.
- It optimizes SQL queries for MySQL syntax.

▨ **spring.jpa.hibernate.ddl-auto=update**

Controls what Hibernate does to your DB schema:

studentEntity.java

```
package com.example.demo;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
@Entity
public class studentEntity {
        @Id
          @GeneratedValue(strategy = GenerationType.IDENTITY)
          private Long id;
          private String firstName;
          private String middleName;
          private String lastName;
          private String dob;
          private String studentId;
          private String street;
          private String city;
          private String state;
          private String country;
    private String zip;
          private String email;
          private String course;
          private String level;
          private String language;
          private String startDate
 private String comments;
```

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

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

public String getFirstName()return firstName;}

public void setFirstName(String firstName) {this.firstName = firstName;}

public String getMiddleName(){return middleName; }

public void setMiddleName(String middleName) {this.middleName = middleName;}

public String getLastName(){return lastName;}

public void setLastName(String lastName) {this.lastName = lastName;}

public String getDob(){return dob;}

public void setDob(String dob) {this.dob = dob;}

public String getStudentId() { return studentId;}

public void setStudentId(String studentId) {this.studentId = studentId;}

public String getStreet(){return street;}

public void setStreet(String street){this.street = street;}

public String getCity() {return city;}

public void setCity(String city) {this.city = city;}

public String getState() {return state;}

public void setState(String state) {this.state = state;}

public String getCountry() {return country;}

public void setCountry(String country) {this.country = country;}

public String getZip() {return zip;}

public void setZip(String zip) {this.zip = zip;}

public String getEmail() {return email;}

public void setEmail(String email) {this.email = email;}

public String getCourse() {return course;}

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

public String getLevel() {return level;}}

public void setLevel(String level) {this.level = level;
```

An **Entity class** is just a **normal Java class** that:

- Represents a **table in the database**.
- Each **field/variable** in the class becomes a **column** in the table.
- It's used with **JPA (Java Persistence API)** to automatically handle database operations.

Why Do We Create a Service Class Object in the Controller?

- In short:
  We use a **Service class object in the Controller** to **separate business logic** from the request handling logic.

HTML Form → 🎮 Controller → 💼 Service → 💾 Repository → 🗄️ Database

1. @GetMapping

```
@GetMapping("/login")

public String showLoginPage() {

   return "login";  // returns login.jsp page

}
```

✅ **When to use:**

- User opens a web page (form/page)
- You want to display data
- URL typed in browser

✅ **2. @PostMapping**

Handling POST requests — like submitting a form.

```
@PostMapping("/register")

public String registerStudent(@ModelAttribute studentEntity student) {

   studentService.save(student); // saves form data

   return "success"}
```

✅ **When to use:**

- Form submission
- Login form, registration form
- Sending data to the backend

✅ 3. @RequestMapping

Handling **any type of request** (GET, POST, PUT, DELETE) — but not specific unless you tell it.

```
@RequestMapping(value = "/home", method = RequestMethod.GET)

public String showHomePage() {

    return "home";

}
```

You can use it like:

- @RequestMapping(method = RequestMethod.GET)
- @RequestMapping(method = RequestMethod.POST)

But usually we prefer @GetMapping or @PostMapping now (they are shorter and cleaner).

Feature	@PostMapping	@RequestMapping
🔧 Purpose	Handles only **POST** requests	Handles **any** HTTP method (GET, POST, PUT, DELETE, etc.)
✍️ Syntax	Short and easy to read	Longer, needs method specified
📦 Introduced In	Spring 4.3+	Older, available since Spring 2.x
🧼 Clean Code	Cleaner for POST-only methods	More flexible, but more verbose

Flow

1. Spring Boot starts and creates tables from @Entity.
2. User opens /register → StudentController shows register.jsp.
3. User fills form and submits → @PostMapping("/register") called.
4. Controller sends data to StudentService.save().
5. Service sends it to StudentRepository.save().
6. JPA automatically inserts data into MySQL.
7. User is shown welcome.jsp.

Writing Flow (UI-Driven)

1. **Design form in JSP (register.jsp)**
2. **Create Entity class to match form fields**
3. **Create Repository**
4. **Create Service + Impl**
5. **Create Controller to handle form**
6. **Configure application.properties**
7. **Test the app**