

MVC Core Principle

Introduction to MVC:

MVC stands for Model-View-Controller; it is a design pattern that separates the application logic into three interconnected components: the model, the view, and the controller.

- **The Model** represents the data and the business logic of the application. It is responsible for handling the data and providing the necessary manipulation methods. It is also responsible for validating the data and ensuring that it is in the correct format.
- **The View** represents the user interface of the application. It is responsible for displaying the data to the user and providing a way for the user to interact with the application. It receives data from the model and displays it to the user in an appropriate format.
- **The Controller** is the component that receives the user's input and decides how to respond to it. It receives input from the view, processes it, and updates the model and the view accordingly. It acts as a mediator between the model and the view, controlling the data flow between them.

The main benefit of the MVC pattern is that it separates the concerns of the application into distinct components, making it easier to understand, maintain, and extend the application. It also promotes the reusability of the code, allowing different views to be used with the same model or vice versa.

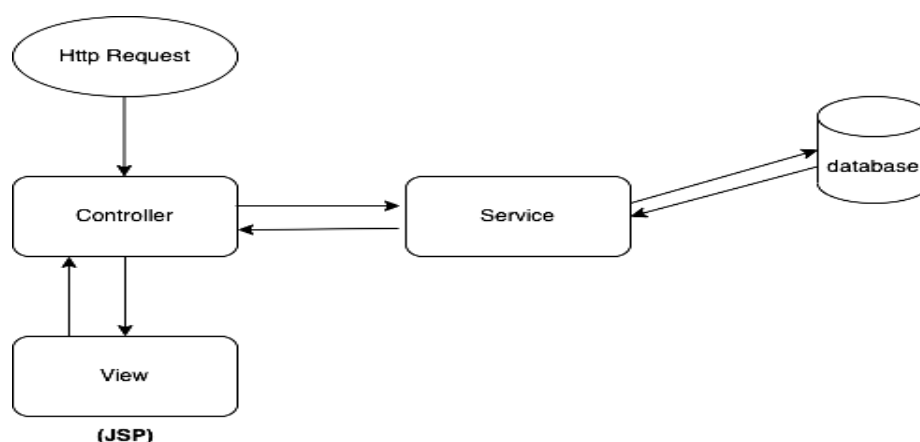


Figure: Diagram of how MVC Architecture works

Example:

A simple example of MVC is a basic calculator application. The Model would handle the calculations, such as addition and subtraction. The View would be the user interface, where the user can enter numbers and see the results. The Controller would receive the user's input and commands and then use the Model to perform the calculations and update the View to display the results.

In this example, the Model handles the data and calculations, the View handles the user interface, and the Controller coordinates and controls the interaction between the Model and the View. This separation of concerns makes the application easier to understand, maintain and extend.

Java Server Pages(JSP):

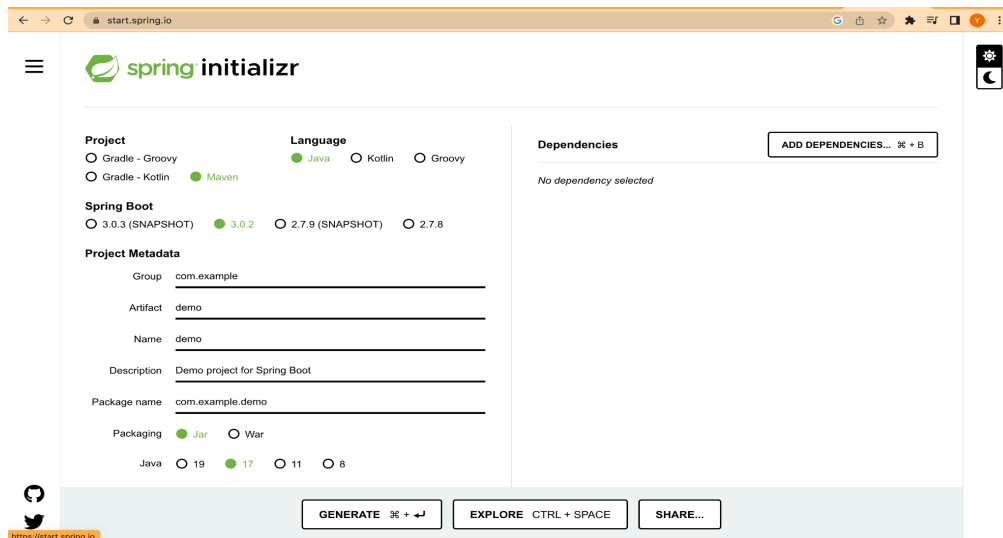
JSP (Java Server Pages) is a technology that allows developers to create dynamic web pages. It uses basic HTML tags to display and format the content. It will enable users to use specific JSP tags to use java code inside the HTML.

In a Spring Boot application, JSPs can be used as the view layer to display the content or take input from the user. For example, it can be used to display data from a database or other sources or to create forms that allow users to submit data to the server for processing.

Project Configuration:

A. Spring Initializer:

- Spring Initializer is a web-based tool provided by the Spring Framework that allows developers to quickly create a new Spring Boot project with minimal configuration. It can be accessed at start.spring.io.
- Using Spring Initializer, developers can create a new project by selecting the desired options and dependencies. The dependencies can be added to the project by selecting them from a list or manually entering their coordinates.
- Once the options and dependencies are selected, Spring Initializer generates a zip file containing the project structure, configuration files, and dependencies. Developers can then download the zip file and import the project into their preferred development environment.
- Spring Initializer is a convenient tool for quickly creating a new Spring Boot project with minimal configuration. It can save developers time by providing a simple way to configure a new project with the desired options and dependencies.



The screenshot shows the Spring Initializer web interface in a browser. The URL is start.spring.io. The interface is divided into several sections:

- Project:** Radio buttons for ☐ Gradle - Groovy, ☐ Gradle - Kotlin, and ☒ Maven.
- Language:** Radio buttons for ☒ Java, ☐ Kotlin, and ☐ Groovy.
- Spring Boot:** Radio buttons for ☐ 3.0.3 (SNAPSHOT), ☒ 3.0.2, ☐ 2.7.9 (SNAPSHOT), and ☐ 2.7.8.
- Project Metadata:**
 - Group:
 - Artifact:
 - Name:
 - Description:
 - Package name:
- Packaging:** Radio buttons for ☒ Jar and ☐ War.
- Java:** Radio buttons for ☐ 19, ☒ 17, ☐ 11, and ☐ 8.
- Dependencies:** A section with the text "No dependency selected" and a button "ADD DEPENDENCIES...".

At the bottom, there are three buttons: "GENERATE" (with a download icon), "EXPLORE" (with "CTRL + SPACE" text), and "SHARE...".

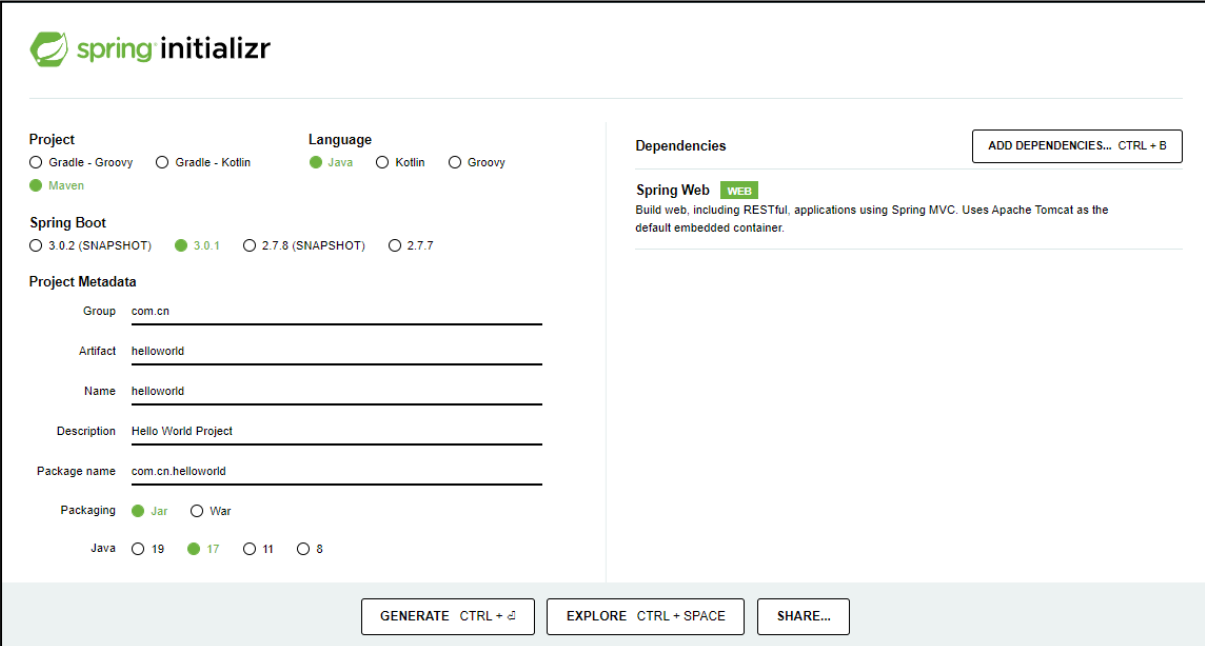
B. Dependencies:

1. **"Web" dependency:** In a Spring Boot project, the "web" dependency is a module that provides support for building web applications using Spring MVC (Model-View-Controller). It includes various features, such as support for handling HTTP requests and responses, form submissions, and exceptions.
2. **"Devtools" dependency:** The "dev tools" dependency in a Spring Boot project provides development-time support for developers, including features such as automatic restart, live reload, remote development, and improved logging. These features make development easier and more efficient. It is typically used in development environments and is not recommended for production environments.

3. **JSTL:** JSTL (JavaServer Pages Standard Tag Library) is a set of tags that can be used in JSP pages to perform common tasks such as iteration, conditional logic, and URL handling. Including the "jstl" dependency in the Spring Boot project gives access to these tags and makes it easy to create dynamic web pages by separating the presentation logic from the business logic.

C. Getting started with the spring boot project:

1. Visit <https://start.spring.io> (This is a web-based Spring Initializer).
Add Spring Web Dependency to your project.



The screenshot shows the Spring Initializr web interface. On the left, under 'Project', 'Maven' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', '3.0.1' is selected. The 'Project Metadata' section has fields for Group (com.cn), Artifact (helloworld), Name (helloworld), Description (Hello World Project), and Package name (com.cn.helloworld). Under 'Packaging', 'Jar' is selected, and under 'Java', '17' is selected. On the right, under 'Dependencies', 'Spring Web' is added with a 'WEB' tag. At the bottom, there are buttons for 'GENERATE', 'EXPLORE', and 'SHARE...'. A 'ADD DEPENDENCIES...' button is also visible in the top right of the dependencies section.

In the end, the selections should look like this, and then click on **GENERATE** button on the bottom. It would download a zip file.

2. Extract the zip in a folder.
3. Import the project in Eclipse by going to -
File -> Open projects from File System -> Directory -> Select the extracted folder -> Finish.
4. Add the following dependencies in **pom.xml**.

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
</dependency>
<dependency>
  <groupId>jstl</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
```

D. Configuration for Application.yml file:

1. For the port number.
2. For the JSP file, add prefixes and suffixes.

```
spring:
  mvc:
    view:
      prefix: "WEB-INF/jsp/"
      suffix: ".jsp"
server:
```

Controller Class:

A Controller class in Spring Boot is a Java class that acts as an intermediary between a client and the server to handle HTTP requests. It handles incoming HTTP requests, processes the data, and returns a response to the client. Controllers are annotated with the `@Controller` annotation and contain methods annotated with `@RequestMapping` to map to specific URLs.

1. HomeController

```
@Controller
public class HomeController {
    @RequestMapping("/home")

    public String getHomePage(){
        return "home";
    }
}
```

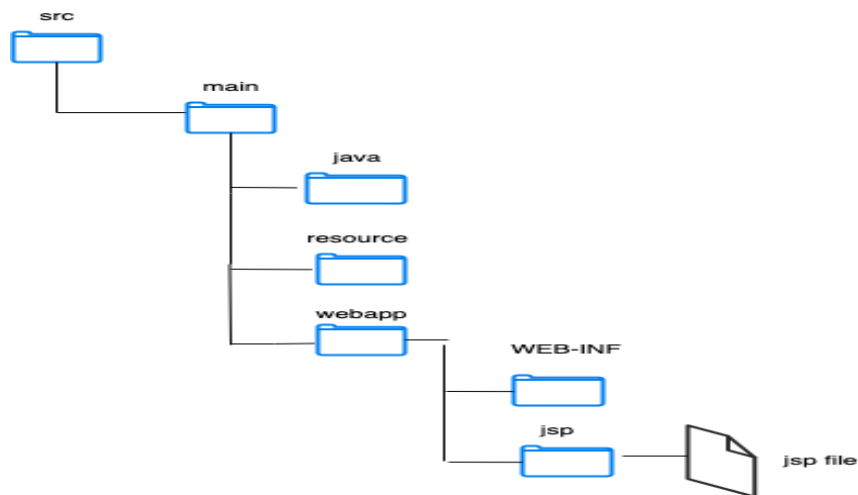
2. SignupController

```
@Controller
public class SignupController {
    @RequestMapping("/signup")
    public String signup(){
        return "signup";
    }
}
```

View Page(JSP pages):

In a Spring Boot application, JSPs can be used as the view layer to display the content or take input from the user. For example, it can be used to display data from a database or other sources or to create forms that allow users to submit data to the server for processing.

A. Directory Structure of the JSP file:



B. Add two pages:

These pages will be triggered using the controllers.

Welcome.jsp

```
<html>
<div>
  <h1>
    <a href="/signup">Sign me up</a>
  </h1>
</div>
</html>
```

Signup.jsp

```
<html>
<div>
  <h1>
    This is signup Page.
  </h1>
</div>
</html>
```

Domain layer

The Domain layer in Spring Boot is a crucial component in the architecture of a Spring Boot application. It contains the domain objects, which are the classes that represent the business entities and their relationships.

1. User interface:

```
public interface User {  
    public boolean createUser(String name,String gender,String location,String  
college);  
    public Integer saveUser();  
  
}
```

2. StudentUser Class:

We have to implement the User interface for better code reusability.

```
package com.example.website.domain;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class StudentUser implements User{  
    String name;  
    String gender;  
    String location;  
    String college;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getGender() {  
        return gender;  
    }  
  
    public void setGender(String gender) {  
        this.gender = gender;  
    }  
  
    public String getLocation() {
```



```
        return location;
    }

    public void setLocation(String location) {
        this.location = location;
    }

    public String getCollege() {
        return college;
    }

    public void setCollege(String college) {
        this.college = college;
    }

    @Override
    public boolean createUser(String name, String gender, String location, String
college) {
        this.name=name;
        this.gender=gender;
        this.location=location;
        this.college=college;
        return true;
    }

    @Override
    public Integer saveUser() {
        System.out.println(this.name);
        return 0;
    }
}
```

Service layer

The Service layer in Spring Boot is a component in the architecture of a Spring Boot application that acts as an intermediary between the Domain layer and the Presentation layer. The Service layer contains the application's business logic and is responsible for processing requests, using the objects in the Domain layer to perform the necessary actions.

1. UserService interface:

```
public interface UserService {  
    public User getUser();  
    public boolean signUp(String name,String gender,String location,String  
college);  
}
```

2. StudentUserService Class:

```
@Service  
public class studentUserService implements UserService{  
  
    @Autowired  
    User studentUser;  
    @Override  
    public User getUser() {  
        return studentUser;  
    }  
  
    @Override  
    public boolean signUp(String name, String gender, String location, String  
college) {  
        boolean isStudentCreated=studentUser.createUser(name, gender, location,  
college);  
        studentUser.saveUser();  
        return isStudentCreated;  
    }  
}
```

MODEL AND MODEL ATTRIBUTE

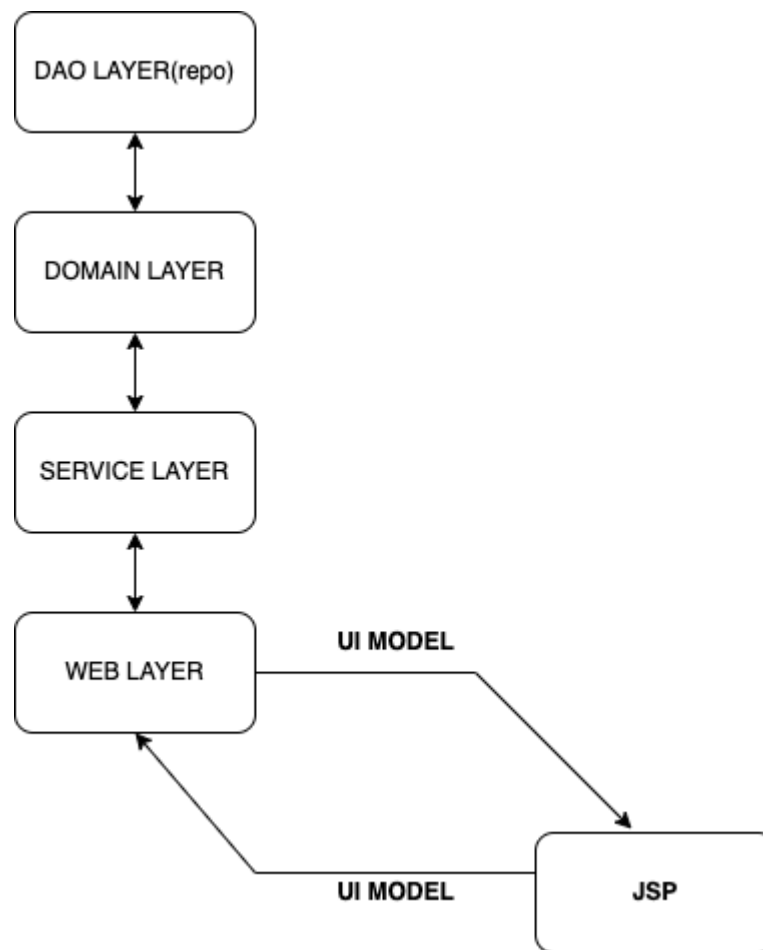


Figure: Representation of WEB Layer and JSP connection using MODEL

Model and model attribute are important concepts in Spring UI that are used to bind data to a view in a web application.

A model refers to an object that holds the data that will be displayed on a view. This data can be any type you want to pass from the controller to the view, such as lists, maps, or simple strings.

The model is created in the controller and can be accessed in the view using the JSTL tags.

A model attribute refers to an annotated parameter in a controller's method. It binds the form data to an object that can be passed to the view. For example, if you have a form that captures user information, you can use a model attribute to bind the form data to an object that represents the user, such as a User object. The model attribute is then passed to the view and can be accessed using JSTL tags.

To use the model and model attribute in a Spring UI application, you need to create a model in the controller, add the model attribute to a method in the controller, and use JSTL tags to access the data in the view.

This code needs to be added in the **User controller** class

```
@Autowired
UserService userService;

@RequestMapping(value = "/signUp")
public String getSignupPage(Model ui) {
    User user = userService.getUser();
    ui.addAttribute("user", user);
    return "signup";
}
```

MVC FORM TAG LIBRARY

The MVC (Model-View-Controller) form tag library is a set of custom JSP tags that provide a convenient way to create dynamic HTML forms in a JSP-based web application that follows the MVC design pattern.

The tag library provides tags for generating various types of form controls, such as text fields, radio buttons, checkboxes, select lists, and more. In an MVC-based web application, the form tag library is used in the View component, which is responsible for presenting the data to the user.

The tags provided by the form tag library are used to render the form controls. The form data is automatically populated from the Model component, which is responsible for managing the application data.

The form tag library provides several benefits over traditional HTML forms, including

Abstraction: It provides a higher level of abstraction, making it easier for developers to create forms without having to write complex HTML code.

Data binding: The form tag library provides data binding between the Model component and the form controls, automatically populating the form with data from the Model and updating the Model with data from the form.

Validation: The form tag library provides built-in support for form validation, making it easy to validate form data and display error messages to the user. Overall, the MVC form tag library is an essential component of MVC-based web applications, providing a powerful and flexible solution for creating dynamic HTML forms.

Configuration:

To utilize the form tag library in a JSP page, it's necessary to add a reference to the required configuration. This is done by including the following directive at the start of the JSP file:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
```

This directive maps the form prefix to the URI of the Spring Framework form tag library, allowing you to use the form tags in your JSP code.

In JSP, several form-related tags can be used to create dynamic HTML forms, including

form:form	This tag creates an HTML form and binds it to a model object. It can be used to specify the action URL, the HTTP method, and other form attributes.
form:input	This tag creates an HTML text input field and binds it to a model property. It can specify the input type (such as text, password, or hidden), the size, the max length, and other attributes.
form:password	This tag creates an HTML password input field and binds it to a model property. It can specify the size, the max length, and other attributes.
form:textarea	This tag creates an HTML textarea field and binds it to a model property. It can specify the rows, columns, and other attributes.
form:textarea	This tag creates an HTML textarea field and binds it to a model property. It can specify the rows, columns, and other attributes.
form:radiobutton	This tag creates an HTML radio button and binds it to a model property. It can be used to specify the selected state and other attributes.
form:select:	This tag creates an HTML select list and binds it to a model property. It can specify the options, the selected value, and other attributes.

Now after Understanding the Form tags let's see the sample code for the "Signup.jsp."

Signup.jsp

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<h1>sign up Page</h1>
<form:form action="registerUser" modelAttribute="user">
```

```
name<form:input path="name" />
<br />
<br />
Gender
<br />
Male<form:radiobutton path="gender" value="male" />
Female<form:radiobutton path="gender" value="female" />
<br/>
<br/>
    Location:
<form:select path="location">
    <form:option value="India"></form:option>
    <form:option value="NRI"></form:option>
</form:select>
<br />
<br />
College
<form:select path="college">
<form:option value="test123"></form:option>
<form:option value="testCollege"></form:option>
</form:select>
<br />
<br />

    <input type="submit" />
</form:form>
</html>
```

To get the details from the form, the following code needs to be added in the **User controller** to register the user:

```
@RequestMapping(value="/register")
public String getResponsePage(@ModelAttribute("user") StudentUser studentUser) {

    if(userService.signup(studentUser.getName(),studentUser.getGender(),studentUser.
        getLocation(),studentUser.getCollege())) {
        return "success";
    }
    return "signup";
}
```

The domain layer, controller layer, and service layer are all important components in a software application architecture.

The domain layer is responsible for defining and managing the core business logic and rules, while the controller layer acts as an intermediary between the domain layer and the presentation layer, handling user requests and input validation.

The service layer is a higher level of abstraction than the domain layer, it defines and implements the application's core services that can be used by multiple parts of the

application, including the domain and controller layers. This layer helps to encapsulate complex logic, promote reusability, and decouple different parts of the application.

In conclusion, these three layers work together to provide a well-structured and modular architecture for a software application, allowing for a clear separation of concerns, better maintainability, and easier testing.

Now we are going to understand how where to store the data

DAO layer

The Data Access Object (DAO) layer is a design pattern that is often used in conjunction with the Model-View-Controller (MVC) architecture. It provides a way to interact with the data storage in a separated and decoupled manner. The DAO layer acts as an intermediary between the Model layer and the data storage.

1. DAO Interface

```
package com.example.registration.repository;

import java.util.Optional;

public interface DAO<T> {
    public Optional<T> get(Integer id);
    int save(T t);
}
```

2. StudentUserDao Class

```
package com.example.registration.repository;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;

import com.example.registration.domain.StudentUser;

@Repository
public class StudentUserDAO implements DAO<StudentUser>{
    private List<StudentUser> studentUserList = new ArrayList<>();

    @Override
    public Optional<StudentUser> get(Integer id) {
```

```
        return Optional.ofNullable(studentUserList.get(id));
    }

    @Override
    public int save(StudentUser studentUser) {
        int studentId = studentUserList.size();
        studentUser.setUserId(studentId);
        studentUserList.add(studentUser);
        System.out.println("user saved");
        return studentId;
    }
}
```

Conclusion

The Model-View-Controller (MVC) architecture in Spring Boot consists of three main layers: Model, View, and Controller. The Model layer represents the data and business logic of the application. The View layer is responsible for presenting the data to the user.

The Controller layer acts as an intermediary between the Model and View layers, handling user inputs and directing data flow between the two.

The MVC architecture provides a clear separation of concerns, making it easier to maintain and modify the code in the future. Additionally, Spring Boot provides a robust framework for building and deploying applications using the MVC pattern.

Instructor Codes

- [Website Application](#)

References

1. [MVC Architecture](#)
2. [MVC Architecture II](#)
3. [JSP](#)
4. [JSP Tag Library](#)