Contents

[Design Pattern 1](#_Toc44093905)

[MVC Design Pattern 1](#_Toc44093906)

[Implementation 1](#_Toc44093907)

[Spring Boot 2](#_Toc44093908)

[Create Controller 3](#_Toc44093909)

[Create Main Class Responsible to initialization the spring boot 4](#_Toc44093910)

[Create JSP file 5](#_Toc44093911)

[Running the Spring Boot 5](#_Toc44093912)

[To Access Deployed Application 5](#_Toc44093913)

[JSTL(JSP Standard Tag Library) Language for VIEW Control 5](#_Toc44093914)

[Creating service. 6](#_Toc44093915)

[JPA (Java Persistence API) 7](#_Toc44093916)

[JSON 11](#_Toc44093917)

[WEB SERVICES 11](#_Toc44093918)

[Types of WebService 12](#_Toc44093919)

[BIG: 12](#_Toc44093920)

[REST Full (Representational State Transfer) 12](#_Toc44093921)

[Spring REST 12](#_Toc44093922)

# Design Pattern

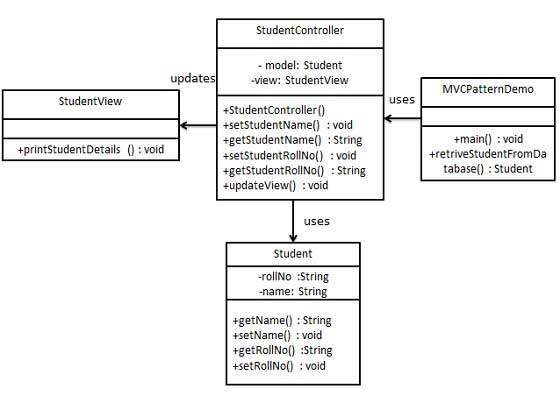
A design pattern is a repeatable solution to a software engineering problem. Unlike most program-specific solutions, design patterns are used in many programs. Design patterns are not considered finished product; rather, they are templates that can be applied to multiple situations and can be improved over time, making a very robust software engineering tool. Because development speed is increased when using a proven prototype, developers using design pattern templates can improve coding efficiency and final product readability.

# MVC Design Pattern

MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.

* **Model** - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.
* **View** - View represents the visualization of the data that model contains.
* **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

## Implementation



# Spring Boot

For all Spring applications, you should start with the [Spring Initializr](https://start.spring.io/). The Initializer offers a fast way to pull in all the dependencies you need for an application and does a lot of the setup for you.

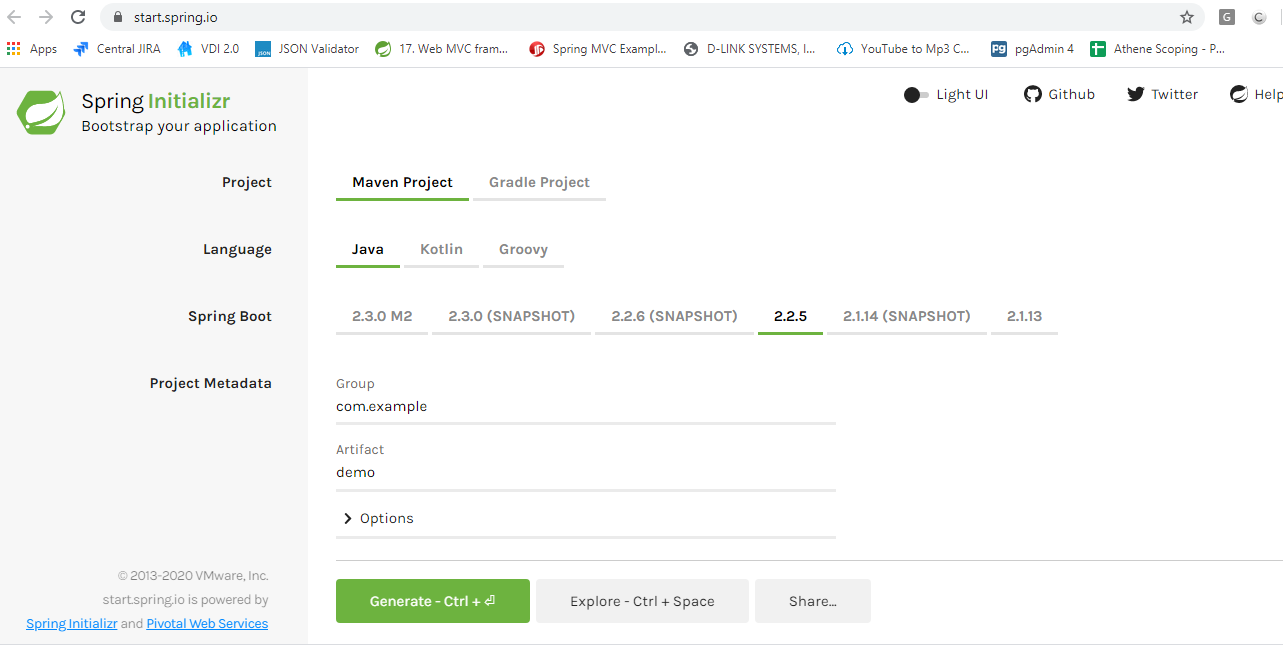


Fig 1. Spring Initilizer Web Application

1. Go to <https://start.spring.io/>

2. Provide required information , In Dependencies ,Select Web

3. Click on Generate

4. This will generate the spring boot project

5. Download it , UnZipp It

6. Open Eclipse File -> Import -> Maven -> Existing Maven Project -> Select the Unzipped folder

7. Open pom.xml , add following entry

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>jsp-api</artifactId>

<version>2.1</version>

<scope>provided</scope>

</dependency>

\**This dependencies is for making jsp as view*

8. Open application.properties, add following entry. This is to make load jsp from /WEB-INF/jsp folder

spring.mvc.view.prefix=/WEB-INF/jsp/

spring.mvc.view.suffix=.jsp

9. Create webapp/WEB-INF/jsp folder inside main folder.

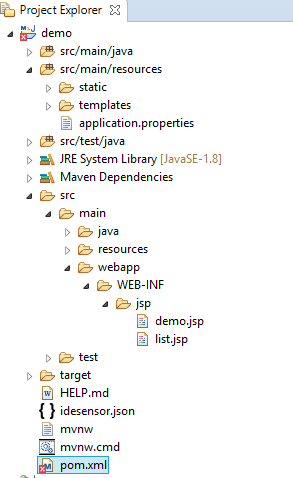


Fig 2: Eclipse Project Explorer

## Create Controller

**Eg,**

**package** com.example.demo.controller;

**import** javax.servlet.http.HttpServletRequest;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.Model;

**import** org.springframework.web.bind.annotation.GetMapping;

@Controller //Annotation

**public** **class** DemoController {

@GetMapping("/demo")

**public** String demo(HttpServletRequest request,Model model) {

model.addAttribute("MESSAGE", "Hello World!");

**return** "demo";

}

}

Here,

* @Controller: Indicates that this class is controller class
* @GetMapping: Indicates that HTTP GET requests to /demo are mapped to demo method of controller class
* We can take help of HttpServletRequest class to get data from request. This class is similar to jsp request implicit object
* We can take help of Model to pass data from controller to view
* the function demo is returning "demo" which means , server will redirect the request to demo.jsp

## Create Main Class Responsible to initialization the spring boot

This class is by default created by Spring Initializer

Eg,

*package com.example.demo;*

*import org.springframework.boot.SpringApplication;*

*import org.springframework.boot.autoconfigure.SpringBootApplication;*

*import org.springframework.context.annotation.ComponentScan;*

*@SpringBootApplication*

*@ComponentScan("com.example.demo.controller")*

*public class DemoApplication {*

*public static void main(String[] args) {*

*SpringApplication.run(DemoApplication.class, args);*

*}*

*}*

Here,

*@ComponentScan("com.example.demo.controller")*

This will tell server to load all the controller classes inside *com.example.demo.controller* Package

## Create JSP file

create jsp file inside webapp/WEB-INF/jsp

Eg,

<h1>Hello,${MESSAGE}</h1>

Here,

As demo function of DemoController is returing "demo". Server will load this page. demo.jsp

## Running the Spring Boot

Right click on DemoApplication.java and select **run as java application**

## To Access Deployed Application

Open browser type following url

http://localhost:8080/demo

Add static files such as css,images,js inside resources/static folder

-> Using request object we can able to get the request parameter values

-> Using model we can able to pass the data to view (Eg, jsp)

-> Using response we can able to work with servlet response

# JSTL(JSP Standard Tag Library) Language for VIEW Control

Required following tag lib

Add following dependency

<dependency>

<groupId>javax.servlet.jsp.jstl</groupId>

<artifactId>jstl-api</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

</dependency>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

if else condition

<c:if test="${condition}">

</c:if>

Loop

<c:forEach items="${products}" var="product">

${product.customerName}

</c:forEach>

here, products is list of object and product is object of current element of loop

product.customerName is property of product class.

C:set

<c:set var="salary" scope="session" value="${2000\*2}"/>

C:Choose

<c:choose>

<c:when test="${salary <= 0}">

Salary is very low to survive.

</c:when>

<c:when test="${salary > 1000}">

Salary is very good.

</c:when>

<c:otherwise>

No comment sir...

</c:otherwise>

</c:choose>

NOTE: For Working with session,cookie follow same way as of Servlets or JSP

# Creating service.

Create an interface

Create a service class that should implement the service interface

Use @Service annotation to create service class

Use @Inject to initialize the service class

For Eg,

*public interface IEmployeeService{*

*public void addEmployee(Employee obj);*

*}*

Now Create Service Class

package com.demo.dao.ora;

*@Service*

*public class EmplyeeServiceOracle implements IEmployeeService{*

*public void addEmployee(Employee obj){*

*//Perform the add operation*

*}*

*}*

Scan the service class package on Application class

Eg,

@ComponentScan("com.demo.dao.ora")

Now Access this service class from controller

@Controller

public class EmployeeController{

@Inject IEmployeeService employeeService; //This will initialize the EmployeeService while loading this controller, service class will be dynamically loaded based on the scanned package

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

public String addEmployee(HttpServletRequest request, Model model, HttpServletResponse response){

Employee e=new Employee();

employeeService.addEmployee(e); //Call the addEmployee method

return "success";

}

}

# JPA (Java Persistence API)

The java ORM(Object Relation Mapping) standard for storing, accessing and managing Java Object in a relational database. Framework to persist the java object into database and vise versa.

**Spring Data JPA** focuses on using JPA to store data in a relational database. Its most compelling feature is the ability to create repository implementations automatically, at runtime, from a repository interface.

Required Dependencies

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.6</version>

</dependency>

Add following parameters in application.porperties

spring.jpa.hibernate.ddl-auto=update

spring.datasource.url=jdbc:mysql://${MYSQL\_HOST:localhost}:3306/jpadb

spring.datasource.username=root

spring.datasource.password=root

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

here, as we are using jpadb database we need to create this database in mysql. No need to create table Spring JPA will create table for the first load

Create Entity Model

Eg,

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

@Entity//Represent the database table

**public** **class** User {

@Id //Primary key of table

@GeneratedValue(strategy=GenerationType.***AUTO***)

**private** Integer id;

**private** String name;

**private** String email;

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

}

Create an interface Repository. This interface is responsible for working with database. The interface should extends from CrudRepository<EntityClass,TYPE ID>

Eg,

**import** org.springframework.data.jpa.repository.Query;

**import** org.springframework.data.repository.CrudRepository;

**import** org.springframework.data.repository.query.Param;

**import** com.example.democomp.dto.User;

**public** **interface** UserRepository **extends** CrudRepository<User, Integer>{

@Query("select u from User u where u.name=:name") //this is to use custom query

**public** User findByName(@Param("name") String name);

}

No need to write code for basic operation CrudRepository will auto provide the implementation for insert,update,delete and select

Now Access This repository from Controller

**import** java.util.List;

**import** java.util.Optional;

**import** javax.servlet.http.HttpServletRequest;

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

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.Model;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.PostMapping;

**import** com.example.democomp.dto.User;

**import** com.example.democomp.mysql.dao.UserRepository;

@Controller

**public** **class** DemoCompController {

@Autowired // This means to get the bean called userRepository

// Which is auto-generated by Spring, we will use it to handle the data

**private** UserRepository userRepository;

@PostMapping("/adduserdb")

**public** String addUserdb(HttpServletRequest req,Model model) {

User u=**new** User();

u.setName(req.getParameter("name"));

u.setEmail(req.getParameter("email"));

userRepository.save(u);

System.***out***.println("Data base updated...");

**return** "adduser";

}

@GetMapping("/getuser")

**public** String getUser(HttpServletRequest req,Model model) {

User user=userRepository.findById(Integer.*parseInt*(req.getParameter("id"))).get();

model.addAttribute("USER",user);

**return** "showuser";

}

@GetMapping("/users")

**public** String getUsers(HttpServletRequest req,Model model) {

Iterable<User> users=userRepository.findAll();

model.addAttribute("USERS",users);

**return** "viewusers";

}

@GetMapping("/getusername")

**public** String getUserName(HttpServletRequest req,Model model) {

User user=userRepository.findByName(req.getParameter("name"));

model.addAttribute("USER",user);

**return** "showusername";

}

}

here, repository.save(user); //will insert/update the data in table

repository.delete(user); //will delete the data with given id in table

repository.findAll(); //returts all available data from table

repository.findById(id); //returns the data from given id

# JSON

JSON (JavaScript Object Notation) is a minimal, readable format for structuring data. It is used primarily to transmit data between a server and web application, as an alternative to XML.

JSON stands for **J**ava**S**cript **O**bject **N**otation

JSON is a lightweight data-interchange format

JSON is language independent

JSON is "self-describing" and easy to understand

Eg,

Simple JSON

{

"Name": "Ram",

"Age": 20,

"class": "BEIT"

}

JSON Array

{

"Name": "Ram",

"Age": 20,

"class": "BEIT",

"course": [

{

"name": "java",

"credits": 3

},

{

"name": "Oracle",

"credits": 2

}

]

}

For More:

<http://developers.squarespace.com/what-is-json/>

<http://www.w3schools.com/json/>

# WEB SERVICES

**Web services** are client and server applications that communicate over the World Wide Web’s (WWW) HyperText Transfer Protocol (HTTP). As described by the World Wide Web Consortium (W3C), web services provide a standard means of interoperating between software applications running on a variety of platforms and frameworks. Web services are characterized by their great interoperability and extensibility, as well as their machine-processable descriptions, thanks to the use of XML. Web services can be combined in a loosely coupled way to achieve complex operations. Programs providing simple services can interact with each other to deliver sophisticated added-value services.

## Types of WebService

Big

Restfull

### BIG:

 Big web services use XML messages that follow the Simple Object Access Protocol (SOAP) standard, an XML language defining a message architecture and message formats. Such systems often contain a machine-readable description of the operations offered by the service, written in the Web Services Description Language (WSDL), an XML language for defining interfaces syntactically.

The SOAP message format and the WSDL interface definition language have gained widespread adoption. Many development tools, such as NetBeans IDE, can reduce the complexity of developing web service applications.

### REST Full (Representational State Transfer)

REST is well suited for basic, ad hoc integrations scenarios, RESTFul web services, often better integrated with HTTP than SOAP-based services are. Do not require XML messages or WSDL service-API definitions.

A RESTful API uses existing HTTP methodologies defined by the RFC 2616 protocol. They use GET to retrieve a resource; PUT to change the state of or update a resource, which can be an [object](https://searchapparchitecture.techtarget.com/definition/object), [file](https://whatis.techtarget.com/definition/file) or [block](https://searchsqlserver.techtarget.com/definition/block); POST to create that resource; and DELETE to remove it.

Ref: <http://docs.oracle.com/javaee/6/tutorial/doc/giqsx.html>

Some frameworks for Restfull webservice using java

i.Jersey : <https://jersey.java.net/>

ii.Dropwizard : <http://www.dropwizard.io/0.9.2/docs/>

iii.Spring: <https://spring.io/guides/gs/rest-service/>

## Spring REST

Spring boot comes with in build REST api support. It is as simple as creating java methods. Just add required annotation in java class and method, your REST API is ready.

Simple Rest API Example

**package** com.example.demoit.controller;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.springframework.data.repository.query.Param;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.PostMapping;

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

**import** org.springframework.web.bind.annotation.RequestParam;

**import** org.springframework.web.bind.annotation.RestController;

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

**import** com.example.demoit.dto.User;

@RestController

@RequestMapping("/user") //Indicates the base url for this API

**public** **class** DemoRestController {

@GetMapping("/get") //Specific URL

**public** Iterable<User> getUser() {

List<User> data = **new** ArrayList<User>();

User u = **new** User();

u.setEmail("abc@gmail.com");

u.setName("Abc");

data.add(u);

**return** data;

}

@PostMapping("/add")

**public** String addUser(@RequestBody User u) {

// here code to add user in db or other places

**return** "success";

}

@GetMapping("/getbyname") //get with request param

**public** User getByName(@RequestParam(value = "name") String name) {

User u = **new** User();

u.setEmail("abc@gmail.com");

u.setName(name.toUpperCase());

**return** u;

}

}

Here,

@RestController-> will create a Rest API Controller

@RequestMapping("/user") -> will create a base url as /user

@GetMapping -> will create a GET method based API

@PostMapping -> will create a POST method based API

Example to access above three API:

**Method**: POST

**URL**:http://localhost:8080/user/add

**PayLoad**:

{

"id": 1,

"name": "Abc",

"email": "abc@gmail.com"

}

This will call **public** String addUser(User u) {

**Method**: get

**URL**: http://localhost:8080/user/getbyname?name=test

This will call **public** User getByName(@RequestParam(value = "name") String name) {

**Method**: get

**URL**: http://localhost:8080/user/get

This will call **public** Iterable<User> getUser() {

We can call these APIs using any rest client such as Advance Rest client , PostMap Rest Client,

using java script APIs Eg, jquery,angular,react etc

using any programming language, Java,Php,.net, python etc

# JQuery

JQuery is java script library, Used to simplify the client side scripting of HTML.Its free and open source. It is the most popular java scripting library used today. jQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications.

**How to integrated jquery in html**

Include the jquery.js file with the help of script src tag

For eg,

*<html>*

*<head>*

*<script src=”jquery-<version>.js”>*

*</script> <!—include the jquery --- >*

*<script>*

***function*** *hide(){*

*$('#abc').hide();*

*}*

***function*** *show(){*

*$("#mytext").val("Give you name");*

*$('#abc').show();*

*}*

*</script>*

*</head>*

*<body>*

*<div id=”abc”>hello</div>*

*<a href=”#” onclick=”hide()”>hide</a>*

*<a href=”#” onclick=”show()”>show</a>*

*<input type=”text” id=”mytext”>*

*</body>*

*</html>*

Explore more on : <https://jquery.com/>

* **AJAX(Asynchronous JavaScript and XML)**

Ajax is a set of web development techniques using many web technologies on the client-side to create asynchronous Web applications. With Ajax, web applications can send data to and retrieve from a server asynchronously (in the background) without interfering with the display and behavior of the existing page. By decoupling the data interchange layer from the presentation layer, Ajax allows for web pages, and by extension web applications, to change content dynamically without the need to reload the entire page. In practice, modern implementations commonly substitute JSON for XML due to the advantages of being native to JavaScript.

JQuery Ajax

Used to implement the Ajax call using jquery $.ajax function. Eg,

<html>

<head>

<link rel=*"stylesheet"*

href=*"https://code.jquery.com/ui/1.12.1/themes/base/jquery-ui.css"*>

<link rel=*"stylesheet"*

href=*"https://jqueryui.com/jquery-wp-content/themes/jqueryui.com/style.css"*>

<script src=*"https://code.jquery.com/jquery-1.12.4.js"*></script>

<script src=*"https://code.jquery.com/ui/1.12.1/jquery-ui.js"*></script>

</script>

<script>

**function** hide(){

$('#abc').hide();

}

**function** show(){

$("#mytext").val("Give you name");

$('#abc').show();

}

**function** webserviceCall() {

tabledata='<table><tr><td>Id</td><td>Name</td><td>Email</td></tr>';

$.ajax({

url : 'http://localhost:8080/room/gets',

contentType: "application/json",

dataType: 'json',

success : **function**(result) {

console.log('Result:'+JSON.stringify(result));

//$('#result').html(data);

$.each(result,**function**(index,item){

console.log(item.id+":"+item.name+":"+item.email);

tabledata+='<tr><td>'+item.id+"</td><td>"+item.name+"</td><td>"+item.email+"</td></tr>";

});

tabledata+="</table>";

document.getElementById("data").innerHTML=tabledata;

},

error: **function**(e){

console.log("error :"+e);

}

});

}

</script>

</head>

<body>

<div id=*"abc"*>hello</div>

<a href=*"#"* onclick="hide()">hide</a><br>

<a href=*"#"* onclick="show()">show</a><br>

<a href=*"#"* onclick="webserviceCall()">WebService Call</a><br>

<input type=*"text"* id=*"mytext"*>

<div id=*"data"*>

</div>

</body>

</html>

Here,

url -> url of the page to call

success -> perform required action on successful call

error: perform required action on error

data-> parameter to pass to page

Following REST API was used by above ajax call

Simple Rest API Example

**package** com.example.demoit.controller;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.springframework.data.repository.query.Param;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.PostMapping;

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

**import** org.springframework.web.bind.annotation.RequestParam;

**import** org.springframework.web.bind.annotation.RestController;

**import** com.example.demoit.dto.User;

@RestController

@CrossOrigin(origins = "\*")

@RequestMapping("/user") //Indicates the base url for this API

**public** **class** DemoRestController {

@GetMapping("/get") //Specific URL

**public** Iterable<User> getUser() {

List<User> data = **new** ArrayList<User>();

User u = **new** User();

u.setEmail("abc@gmail.com");

u.setName("Abc");

data.add(u);

**return** data;

}

}

# Image Upload

-Use  MultipartFile class for File upload

Eg,

@PostMapping("/uploadimage")

**public** String uploadImage(@RequestParam("image") MultipartFile multipartFile) **throws** IOException {

String fileName = StringUtils.*cleanPath*(multipartFile.getOriginalFilename()); //Gets the uploaded file name

String uploadDir = "user-photos";

Path uploadPath = Paths.*get*(uploadDir);

**if** (!Files.*exists*(uploadPath)) {

Files.*createDirectories*(uploadPath); //Create a directory if not available

}

**try** (InputStream inputStream = multipartFile.getInputStream()) {

Path filePath = uploadPath.resolve(fileName);

Files.*copy*(inputStream, filePath, StandardCopyOption.***REPLACE\_EXISTING***);

} **catch** (IOException ioe) {

**throw** **new** IOException("Could not save image file: " + fileName, ioe);

}

**return** "demo";

}

Expose the directory to WWW

-Create a configuration class and add directory

Eg,

**package** com.example.comdemo;

**import** java.nio.file.Path;

**import** java.nio.file.Paths;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;

**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration

**public** **class** MvcConfig **implements** WebMvcConfigurer {

@Override

**public** **void** addResourceHandlers(ResourceHandlerRegistry registry) {

exposeDirectory("user-photos", registry);

}

**private** **void** exposeDirectory(String dirName, ResourceHandlerRegistry registry) {

Path uploadDir = Paths.*get*(dirName);

String uploadPath = uploadDir.toFile().getAbsolutePath();

**if** (dirName.startsWith("../")) dirName = dirName.replace("../", "");

registry.addResourceHandler("/" + dirName + "/\*\*").addResourceLocations("file:/"+ uploadPath + "/");

}

}