Spring Boot MVC Application

Component	**Code**

User.java	package com.m1.m2;
	
	public class User {
	        private String userName;
	
	        public String getUserName() {
	                return userName;
	        }
	
	        public void setUserName(String userName) {
	                this.userName = userName;
	        }
	}

HomeContrller.java	package com.m1.m2;
	
	import java.text.DateFormat;
	import java.util.Date;
	import java.util.Locale;
	
	import org.slf4j.Logger;
	import org.slf4j.LoggerFactory;
	import org.springframework.stereotype.Controller;
	import org.springframework.ui.Model;
	import
	org.springframework.web.bind.annotation.RequestMapping;
	import
	org.springframework.web.bind.annotation.RequestMethod;
	import org.springframework.validation.annotation.Validated;
	
	/**
	 * Handles requests for the application home page.
	 */
	@Controller
	public class HomeController {
	
	        private static final Logger logger =
	LoggerFactory.getLogger(HomeController.class);

```
	/**
	 * Simply selects the home view to render by returning its
name.
	 */
	@RequestMapping(value = "/", method =
RequestMethod.GET)
	public String home(Locale locale, Model model) {
		logger.info("Welcome home! The client locale is {}.",
locale);

		Date date = new Date();
		DateFormat dateFormat =
DateFormat.getDateTimeInstance(DateFormat.LONG,
DateFormat.LONG, locale);

		String formattedDate = dateFormat.format(date);

		model.addAttribute("serverTime", formattedDate );

		return "home";
	}
	@RequestMapping(value = "/user", method =
RequestMethod.POST)
	public String user(@Validated User user, Model model) {
		System.out.println("User Page Requested");
		model.addAttribute("userName",
user.getUserName());
		return "user";
	}

}
```

Home.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
<head>
	<title>Home</title>
</head>
<body>
<h1>
	Hello world!
```

	</h1> <P> The time on the server is ${serverTime}. </P> <form action="user" method="post"> <input type="text" name="userName"> <input type="submit" value="Login"> </form> </body> </html>
User.jsp	<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%> <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "https://www.w3.org/TR/html4/loose.dtd"> <html> <head> <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"> <title>User Home Page</title> </head> <body> <h3>Hi ${userName}</h3> </body> </html>

Spring	Spring Boot
Spring Framework is a widely used Java EE framework for building applications.	**Spring Boot Framework** is widely used to develop **REST APIs**.
It aims to simplify Java EE development that makes developers more productive.	It aims to shorten the code length and provide the easiest way to develop **Web Applications**.

The primary feature of the Spring Framework is **dependency injection**.	The primary feature of Spring Boot is **Autoconfiguration**. It automatically configures the classes based on the requirement.
It helps to make things simpler by allowing us to develop **loosely coupled** applications.	It helps to create a **stand-alone** application with less configuration.
The developer writes a lot of code (**boilerplate code**) to do the minimal task.	It **reduces** boilerplate code.
To test the Spring project, we need to set up the sever explicitly.	Spring Boot offers **embedded server** such as **Jetty** and **Tomcat**, etc.
It does not provide support for an in-memory database.	It offers several plugins for working with an embedded and **in-memory** database such as **H2**.
Developers manually define dependencies for the Spring project in **pom.xml**.	Spring Boot comes with the concept of **starter** in pom.xml file that internally takes care of downloading the dependencies **JARs** based on Spring Boot Requirement.

Spring Boot	**Spring MVC**
Spring Boot is a module of Spring for packaging the Spring-based application with sensible defaults.	**Spring MVC** is a model view controller-based web framework under the Spring framework.

It provides default configurations to build **Spring-powered** framework.	It provides **ready to use** features for building a web application.
There is no need to build configuration manually.	It requires build configuration manually.
There is **no requirement** for a deployment descriptor.	A Deployment descriptor is **required**.
It avoids boilerplate code and wraps dependencies together in a single unit.	It specifies each dependency separately.
It **reduces** development time and increases productivity.	It takes **more** time to achieve the same.

Introduction to RESTful Web Services

Advantages of RESTful web services

- RESTful web services are **platform-independent**.
- It can be written in any programming language and can be executed on any platform.
- It provides different data format like **JSON, text, HTML,** and **XML**.
- It is fast in comparison to SOAP because there is no strict specification like SOAP.
- These are **reusable**.
- They are **language neutral**.

Spring Boot provides a very good support to building RESTful Web Services for enterprise applications

REST stands for **REpresentational State Transfer**. It is developed by **Roy Thomas Fielding**, who also developed HTTP. The main goal of RESTful web services is to make web services **more effective**. RESTful web services try to define services using the different concepts that are already present in HTTP. REST is an **architectural approach**, not a protocol.

- **RESTful web services** are loosely coupled, lightweight web services that are particularly well suited for creating APIs for clients spread out across the internet. Representational State Transfer (REST) is an architectural style of client-server application centered around the **transfer** of **representations** of **resources** through requests and responses. In the REST architectural style, data and functionality are considered resources and are accessed using **Uniform Resource Identifiers (URIs)**, typically links on the Web. The resources are represented by documents and are acted upon by using a set of simple, well-defined operations.
- For example, a REST resource might be the current weather conditions for a city. The representation of that resource might be an XML document, an image file, or an HTML page. A client might retrieve a particular representation, modify the resource by updating its data, or delete the resource entirely.
- The REST architectural style is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

It does not define the standard message exchange format. We can build REST services with both XML and JSON. JSON is more popular format with REST. The **key abstraction** is a resource in REST. A resource can be anything. It can be accessed through a **Uniform Resource Identifier (URI)**. For example:

The resource has representations like XML, HTML, and JSON. The current state capture by representational resource. When we request a resource, we provide the representation of the resource. The important methods of HTTP are:

- **GET:** It reads a resource.
- **PUT:** It updates an existing resource.
- **POST:** It creates a new resource.
- **DELETE:** It deletes the resource.

For example, if we want to perform the following actions in the social media application, we get the corresponding results.

POST /users: It creates a user.

GET /users/{id}: It retrieves the detail of a user.

GET /users: It retrieves the detail of all users.

DELETE /users: It deletes all users.

DELETE /users/{id}: It deletes a user.

GET /users/{id}/posts/post_id: It retrieve the detail of a specific post.

POST / users/{id}/ posts: It creates a post of the user.

Further, we will implement these URI in our project.

HTTP also defines the following standard status code:

- **404:** RESOURCE NOT FOUND
- **200:** SUCCESS
- **201:** CREATED
- **401:** UNAUTHORIZED
- **500:** SERVER ERROR

RESTful Service Constraints

- There must be a service producer and service consumer.
- The service is stateless.
- The service result must be cacheable.
- The interface is uniform and exposing resources.
- The service should assume a layered architecture.