# **Assignment 10**

Q1. What is the Spring MVC framework?

Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control and Dependency Injection.

A Spring MVC provides an elegant solution to use MVC in spring framework with the help of **DispatcherServlet**. Here, **DispatcherServlet** is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.

* **Model** - A model contains the data of the application. Data can be a single object or a collection of objects.
* **Controller** - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.
* **View** - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.
* **Front Controller** - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible for managing the flow of the Spring MVC application.

Q2. What are the benefits of Spring MVC framework over other MVC frameworks?

Benefits of Spring MVC Framework: -

* **Separate roles** - The Spring MVC separates each role, where the model object, controller, command object, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.
* **Light-weight** - It uses a light-weight servlet container to develop and deploy your application.
* **Powerful Configuration** - It provides a robust configuration for both framework and application classes that includes easy referencing across contexts, such as from web controllers to business objects and validators.
* **Rapid development** - The Spring MVC facilitates fast and parallel development.
* **Reusable business code** - Instead of creating new objects, it allows us to use the existing business objects.
* **Easy to test** - In Spring, generally we create JavaBeans classes that enable you to inject test data using the setter methods.
* **Flexible Mapping** - It provides specific annotations that easily redirect the page.

Q3. What is DispatcherServlet in Spring MVC? In other words, can you explain the Spring MVC architecture?

DispatcherServlet acts as the **Front Controller** for Spring-based web applications. Any request is going to come into our website the front controller is going to stand in front and is going to accept all the requests and once the front controller accepts that request then this is the job of the front controller that it will decide that who is the right controller to handle that request. For example, refer to the below image. Suppose we have a website called **student.com** and the client is making a request to save student data by hitting the following URL **student.com/save** and its first come to the front controller and once the front controller accepts that request it is going to assign to the Controller\_1 as this controller handle the request for /save operation. Then it is going to return the response to the Client.

In other words, *DispatcherServlet is*

*DispatcherServlet handles an incoming HttpRequest, delegates the request, and processes that request according to the configured HandlerAdapter interfaces that have been implemented within the Spring application along with accompanying annotations specifying handlers, controller endpoints, and response objects.*

Q4. What is a View Resolver pattern and explain its significance in Spring MVC?

**Spring MVC** is a Web MVC Framework for building web applications. In generic all MVC frameworks provide a way of working with views. Spring does that via the **ViewResolvers**, which enables you to render models in the browser without tying the implementation to specific view technology.

### Configure ViewResolver inside the Spring Configuration File

code as follows:

<bean id = 'viewResolver' class = "org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <property name="prefix" value="/WEB-INF/views/"/>  
 <property name="suffix" value=".jsp"/>  
</bean>

Q5. What are the differences between @RequestParam and @PathVariable annotations?

Both **@PathVariable and @RequestParam** annotations simplify the process of extracting data from incoming requests in Spring Boot. They provide a clean and declarative way to access dynamic values from URLs and query parameters, making it easier to handle and process request data in your REST API endpoints.

The **@PathVariable** annotation is used to retrieve data from the URL path. By defining placeholders in the request mapping URL, you can bind those placeholders to method parameters annotated with @PathVariable. This allows you to access dynamic values from the URL and use them in your code. For example, you can extract a user ID from a URL like /users/123 and pass it to a method that retrieves the corresponding user’s details.

Example

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping("/{userId}")

public ResponseEntity<User> getUserDetails(@PathVariable Long userId) {

// Implementation to fetch user details based on the provided userId

// ...

return ResponseEntity.ok(user);

}

}

The **@RequestParam** annotation enables you to extract data from the query parameters in the request URL. Query parameters are key-value pairs appended to the URL after a question mark (?). With @RequestParam, you can specify the name of the parameter to retrieve and bind it to a method parameter. This is useful when you need to pass additional information or filters to your API endpoints. For instance, you can extract the value of a name parameter from a URL like /users/search?name=John and use it to search for users with the given name.

Example

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping("/search")

public ResponseEntity<List<User>> searchUsers(@RequestParam("name") String name) {

// Implementation to search users based on the provided name

// ...

return ResponseEntity.ok(users);

}

}

Q6. What is the Model in Spring MVC?

In Spring MVC, the model works in a container that contains the data of the application. Here, data can be in any form such as objects, strings, information from the database, etc.

It is required to place the **Model** interface in the controller part of the application. The object of **HttpServletRequest** reads the information provided by the user and passes it to the **Model** interface. Now, a view page easily accesses the data from the model part.

|  |  |
| --- | --- |
| **Method** | **Description** |
| Model addAllAttributes(Collection<?> arg) | It adds all the attributes in the provided Collection into this Map. |
| Model addAllAttributes(Map<String,?> arg) | It adds all the attributes in the provided Map into this Map. |
| Model addAllAttribute(Object arg) | It adds the provided attribute to this Map using a generated name. |
| Model addAllAttribute(String arg0, Object arg1) | It binds the attribute with the provided name. |
| Map<String, Object> asMap() | It returns the current set of model attributes as a Map. |
| Model mergeAttributes(Map< String,?> arg) | It adds all attributes in the provided Map into this Map, with existing objects of the same name taking precedence. |
| boolean containsAttribute(String arg) | It indicates whether this model contains an attribute of the given name |

Q7. What is the role of @ModelAttribute annotation?

In Spring MVC, the @ModelAttribute annotation binds a method parameter or method return value to a named model attribute and then exposes it to a web view. It refers to the property of the Model object.

For example, if we have a form with a form backing object that is called “Student” then we can have Spring MVC supply this object to a Controller method by using the @ModelAttribute annotation:

@RequestMapping("/home")  
public String showHomePage(@ModelAttribute("studentInfo") StudentInfoDTO studentInfoDTO) {  
   
 return "something";  
   
}

Q8. What is the significance of @Repository annotation?

@Repository Annotation is a specialization of **@Component** annotation which is used to indicate that the class provides the mechanism for storage, retrieval, update, delete and search operation on objects. Though it is a specialization of @Component annotation, Spring Repository classes are autodetected by spring framework through classpath scanning. This annotation is a general-purpose stereotype annotation which is very close to the DAO pattern where DAO classes are responsible for providing CRUD operations on database tables.

Q9. What does REST stand for? and what is RESTful web services?

REST stands for REpresentational State Transfer.

### REST is an architectural style not a protocol.

### **RESTful Architecture:**

**Division of State and Functionality:** State and functionality are divided into distributed resources. This is because every resource be accessible via normal HTTP commands. That means a user should be able to issue the GET request to get a file, issue the POST or PUT request to put a file on the server, or issue the DELETE request to delete a file from the server.

**Stateless, Layered, Caching-Support, Client/Server Architecture:** A type of architecture where the web browser acts as the client, and the web server acts as the server hosting the application, is called a client/server architecture. The state of the application should not be maintained by REST. The architecture should also be layered, meaning that there can be intermediate servers between the client and the end server. It should also be able to implement a well-managed caching mechanism.

### **Principles of RESTful applications:**

1. **URI Resource Identification:** A RESTful web service should have a set of resources that can be used to select targets of interactions with clients. These resources can be identified by URI (Uniform Resource Identifiers). The URIs provide a global addressing space and help with service discovery.
2. **Uniform Interface:** Resources should have a uniform or fixed set of operations, such as PUT, GET, POST, and DELETE operations. This is a key principle that differentiates between a REST web service and a non-REST web service.
3. **Self-Descriptive Messages:** As resources are decoupled from their representation, content can be accessed through a large number of formats like HTML, PDF, JPEG, XML, plain text, JSON, etc. The metadata of the resource can be used for various purposes like control caching, detecting transmission errors, finding the appropriate representation format, and performing authentication or access control.
4. **Use of Hyperlinks for State Interactions:** In REST, interactions with a resource are stateless, that is, request messages are self-contained. So explicit state transfer concept is used to provide stateful interactions. URI rewriting, cookies, and form fields can be used to implement the exchange of state. A state can also be embedded in response messages and can be used to point to valid future states of interaction.

Q10.What are the differences between RESTful web services and SOAP web services?

|  |  |  |
| --- | --- | --- |
| **No.** | **SOAP** | **REST** |
| 1) | SOAP is a **protocol**. | REST is an **architectural style**. |
| 2) | SOAP stands for **Simple Object Access Protocol**. | REST stands for **REpresentational State Transfer**. |
| 3) | SOAP **can't use REST** because it is a protocol. | REST **can use SOAP** web services because it is a concept and can use any protocol like HTTP, SOAP. |
| 4) | SOAP **uses services interfaces to expose the business logic**. | REST **uses URI to expose business logic**. |
| 5) | **JAX-WS** is the java API for SOAP web services. | **JAX-RS** is the java API for RESTful web services. |
| 6) | SOAP **defines standards** to be strictly followed. | REST does not define too much standards like SOAP. |
| 7) | SOAP **requires more bandwidth** and resources than REST. | REST **requires less bandwidth** and resource than SOAP. |
| 8) | SOAP **defines its own security**. | RESTful web services **inherits security measures** from the underlying transport. |
| 9) | SOAP **permits XML** data format only. | REST **permits different** data formats such as Plain text, HTML, XML, JSON etc. |
| 10) | SOAP is **less preferred** than REST. | REST **more preferred** than SOAP. |