WEEK 7 COURSE CONTENT : RESTFUL WEB SERVICES

TOPICS:

- Request mappings and HTTP methods in Spring MVC
- Handling request and response bodies (JSON)
- Validation and error handling in RESTful APIs
- HATEOAS (Hypermedia as the Engine of Application State)
- Versioning and documentation of RESTful APIs
- Consuming RESTful APIs with RestTemplate or WebClient

LEARNING OUTCOMES:

By the end of this module, students will be able to:

- Implement request mappings and handle different HTTP methods (GET, POST, PUT, DELETE) in a Spring MVC-based RESTful application.
- Handle request and response bodies in RESTful APIs, supporting formats such as JSON
- Validate input data and handle validation errors in RESTful APIs, providing appropriate error responses.
- Implement error handling mechanisms to handle exceptions and return meaningful error messages in RESTful APIs.
- Understand and apply the principles of HATEOAS (Hypermedia as the Engine of Application State) to enhance the discoverability and navigability of the API.
- Learn different approaches to versioning and documentation of RESTful APIs, ensuring backward compatibility and facilitating API evolution.
- Understand the concepts of consuming external RESTful APIs and utilize either RestTemplate or WebClient to make API calls and handle responses.

DAY 1:

REQUEST MAPPINGS AND HTTP METHODS IN SPRING MVC

In Spring MVC, request mappings and HTTP methods play a crucial role in defining how incoming requests from clients are handled by the server-side application. Let's break down these concepts in a beginner-friendly manner:

- Request Mapping: A request mapping in Spring MVC is used to map incoming requests to specific handler methods that will process those requests. It helps determine which method should be invoked based on the URL pattern of the incoming request. By defining request mappings, you can specify the entry points for different functionalities or resources in your application.

For example, consider a Spring MVC application that manages a blog. You might define a request mapping for handling requests related to creating a new blog post, such as "/posts/new". When a client sends a request to this URL, the corresponding handler method will be invoked to handle the request and perform the necessary actions, such as saving the new blog post.

- HTTP Methods: HTTP methods, also known as HTTP verbs, are standardized methods that define the type of operation the client wants to perform on a resource. These methods are sent along with the request to indicate the desired action. The most commonly used HTTP methods are:
- GET: Retrieves information or data from the server. It is often used to read or fetch resources without modifying them. For example, retrieving a list of blog posts or viewing a specific post.
- POST: Sends data to the server to create or submit new resources. It is typically used for actions like submitting a form or creating a new record. For example, creating a new blog post by submitting a form with the post details.
- PUT: Updates an existing resource on the server. It is used to modify or replace the entire resource with the new data provided. For example, updating the content of a blog post.
- DELETE: Deletes a specified resource on the server. It is used to remove or delete a resource. For example, deleting a blog post.
- PATCH: Partially updates an existing resource on the server. It is used to modify a specific part or properties of a resource without replacing the entire resource.

In Spring MVC, you can associate specific HTTP methods with your request mappings. This means that a handler method will only be invoked if the incoming request matches both the URL pattern and the HTTP method specified in the request mapping. This allows you to have different methods for different operations on the same resource.

For example, you can define a request mapping with the HTTP method "GET" to handle requests for retrieving a blog post, and another request mapping with the HTTP method "POST" to handle requests for creating a new blog post. The appropriate method will be invoked based on the HTTP method specified in the request.

By combining request mappings and HTTP methods, you can create a well-structured and organized API in your Spring MVC application, ensuring that each request is handled by the appropriate handler method based on the URL pattern and the intended operation specified by the HTTP method.

HANDLING REQUEST AND RESPONSE BODIES (JSON)

In Spring MVC, handling request and response bodies in JSON format is a common requirement for building RESTful APIs. JSON (JavaScript Object Notation) is a lightweight data interchange format that is widely used for transmitting structured data between a client and a server.

To handle JSON request and response bodies in Spring MVC, you need to configure your application to support JSON data. Here are the key steps involved:

- Jackson Dependency: Add the Jackson library dependency to your project. Jackson is a popular JSON processing library in the Java ecosystem that provides support for converting Java objects to JSON and vice versa.

- Request Body: To handle a JSON request body, you can use the **@RequestBody** annotation on a method parameter in your controller. This annotation tells Spring MVC to automatically convert the incoming JSON data into a Java object.

For example, consider a scenario where you have an API endpoint for creating a new user. You can define a controller method like this:

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@RequestBody User user) {
  // Process the user object (e.g., save it to the database)
  // Return an appropriate response
}
```

In this example, the **@RequestBody** annotation indicates that the **user** parameter should be populated with the JSON data sent in the request body. Spring MVC uses Jackson to convert the JSON data into a **User** object.

- Response Body: To send a JSON response body, you can use the **@ResponseBody** annotation on a method or at the class level. This annotation instructs Spring MVC to convert the returned object to JSON and include it in the response body.

For example, consider a scenario where you have an API endpoint for retrieving user details. You can define a controller method like this:

```
@GetMapping("/users/{id}")
@ResponseBody
public User getUser(@PathVariable Long id) {
   // Retrieve the user from the database based on the ID
   // Return the user object
}
```

In this example, the **@ResponseBody** annotation ensures that the returned **User** object is converted to JSON and sent as the response body.

- Content Negotiation: By default, Spring MVC automatically performs content negotiation to determine the response format based on the client's request. However, you can explicitly specify the response format as JSON using the **produces** attribute in the **@RequestMapping** annotation.

For example:

```
@GetMapping(value = "/users/{id}", produces = "application/json")
@ResponseBody
public User getUser(@PathVariable Long id) {
   // Retrieve the user from the database based on the ID
   // Return the user object
}
```

In this example, the **produces = "application/json"** attribute ensures that the response is in JSON format, regardless of the client's request headers.

By following these steps, you can easily handle JSON request and response bodies in your Spring MVC application. This allows you to build APIs that communicate using JSON data, enabling interoperability with various clients and frameworks.

DAY 2:

VALIDATION AND ERROR HANDLING IN RESTFUL APIS

Validation and error handling are crucial aspects of building RESTful APIs to ensure data integrity and provide meaningful responses to clients. In RESTful APIs, validation is the process of verifying the correctness and integrity of incoming request data, while error handling involves appropriately handling and communicating errors to clients. Let's explore these concepts in more detail:

- Validation: Validation is the process of checking if the incoming request data meets the expected criteria or conforms to the defined rules. It helps prevent invalid or inconsistent data from being processed or stored in the system. In RESTful APIs, validation is typically performed on the request payload, query parameters, path variables, or headers.

Common validation techniques include:

- Data Type Validation: Ensuring that the data types of the incoming values match the expected types (e.g., string, number, boolean).
- Required Field Validation: Checking if mandatory fields are present in the request.
- Length and Range Validation: Verifying the length or range of values (e.g., minimum/maximum length, minimum/maximum numeric value).
- Format Validation: Validating the format of data, such as email addresses, phone numbers, or dates, based on specific patterns or regular expressions.
- Business Rule Validation: Applying custom business-specific rules to ensure data integrity.

In Spring MVC, you can perform validation using annotations provided by the Spring Validation framework, such as **@NotNull**, **@Size**, **@Pattern**, and **@Valid**. These annotations can be applied to request model attributes or method parameters in your controller methods.

- Error Handling: Error handling is the process of gracefully handling and communicating errors that occur during the execution of an API. It involves providing meaningful error messages and appropriate HTTP status codes to clients. Error responses should help clients understand what went wrong and guide them in taking corrective actions.

In RESTful APIs, common practices for error handling include:

- HTTP Status Codes: Using appropriate HTTP status codes to indicate the outcome of the request. For example, **400 Bad Request** for invalid requests, **404 Not Found** for resource not found, and **500 Internal Server Error** for server-side errors.
- Error Response Payload: Returning error details in the response body using a consistent error format, typically in JSON format. The payload may include error codes, error messages, timestamps, and additional information to assist clients in understanding and resolving the issue.
- Exception Handling: Handling exceptions that occur during request processing and mapping them to meaningful error responses. In Spring MVC, you can use exception handling mechanisms such as **@ExceptionHandler** to handle specific exceptions and provide custom error responses.
- Validation Errors: Handling validation errors by capturing validation failures and returning appropriate error responses. Spring MVC's **BindingResult** can be used to collect validation errors and provide detailed information about the validation failures.

By combining validation and error handling techniques, you can ensure that only valid data is processed by your API, and clients receive informative and consistent error responses when issues occur.

It's important to note that error handling and validation should be applied at various layers, including input validation, business logic validation, and database integrity checks, to ensure comprehensive data validation and error handling throughout the application.

HATEOAS (HYPERMEDIA AS THE ENGINE OF APPLICATION STATE)

HATEOAS, which stands for Hypermedia as the Engine of Application State, is a principle and architectural constraint in the design of RESTful APIs. It is one of the key principles of the REST (Representational State Transfer) architectural style.

At its core, HATEOAS is about including hypermedia links in API responses to enable clients to navigate and discover available resources and actions dynamically. In other words, instead of relying on prior knowledge or fixed URLs, clients can discover and interact with resources based on the links provided in the API response.

Here are some key points to understand about HATEOAS:

- Hypermedia Links: Hypermedia links are URLs embedded within API responses that provide navigation and interaction capabilities. These links represent relationships between resources and actions that can be performed on those resources. The links may include information about the allowed HTTP methods, request parameters, and additional metadata.
- Dynamic Resource Discovery: With HATEOAS, clients can dynamically discover and navigate available resources by following the hypermedia links provided in the API response. This allows for a more flexible and evolvable API architecture, as clients can adapt to changes in resource URLs or the introduction of new resources without relying on hard-coded URLs.
- Stateful Interaction: HATEOAS promotes the idea that API responses should include not only the requested resource but also the links to related resources and available actions. This makes the API responses stateful, providing clients with the necessary

information to determine the next actions to take based on the current state of the application.

- Reduced Coupling: By providing hypermedia links, HATEOAS reduces the coupling between the client and the server. Clients do not need to have prior knowledge of the API structure or hard-coded URLs, as they can discover and interact with resources based on the provided links. This allows for more loosely coupled and scalable systems.
- Improved API Discoverability: HATEOAS enhances the discoverability of APIs by making the API behavior self-descriptive. Clients can explore the API by following links and understanding the available actions based on the provided hypermedia. This enables easier adoption of the API and encourages self-guided exploration.

Implementing HATEOAS in an API involves including hypermedia links in the response payloads, typically in a standardized format like JSON or XML. Common formats for expressing hypermedia links include HAL (Hypertext Application Language), JSON-LD (JSON Linked Data), and Siren.

Frameworks and libraries, such as Spring HATEOAS in the Spring ecosystem, provide tools and abstractions to simplify the implementation of HATEOAS principles in RESTful APIs. These libraries offer utilities for creating and managing hypermedia links, allowing developers to focus on the core functionality of the API.

By following the HATEOAS principle, APIs become more discoverable, self-descriptive, and flexible, enabling clients to navigate and interact with resources in a dynamic and scalable manner.

DAY 3:

VERSIONING AND DOCUMENTATION OF RESTFUL APIS

Versioning and documentation are important aspects of designing and maintaining RESTful APIs. They help in managing changes, ensuring compatibility, and providing clear information to API consumers. Let's look at each of these topics in more detail:

- Versioning: Versioning is the practice of assigning a unique identifier to different versions of an API. It allows multiple versions of the API to coexist, enabling clients to choose the version they want to use. There are different approaches to versioning RESTful APIs:
- URL Versioning: In this approach, the version is included in the URL, such as **api/v1/resource**. It provides a clear distinction between different versions but can clutter the URL structure.
- Query Parameter Versioning: Version information is passed as a query parameter, such as **api/resource?version=1**. It keeps the URL cleaner but may have caching implications.
- Header Versioning: The version is specified in a custom header, like **Accept-Version: 1**. It allows cleaner URLs and avoids caching issues, but requires clients to send the appropriate header.
- Content Negotiation Versioning: Different versions are represented by different media types, and clients specify the desired version in the **Accept** header. For example,

Accept: application/vnd.company.resource-v1+json. This approach requires careful management of media types.

Choosing the right versioning approach depends on the specific requirements of your API and the needs of your clients.

- Documentation: API documentation is crucial for enabling developers to understand and use the API effectively. It provides details about available endpoints, request/response formats, authentication, error handling, and other important aspects. Here are some common practices for documenting RESTful APIs:
- API Reference Documentation: This includes detailed documentation of all endpoints, their input/output parameters, and expected responses. It may also cover authentication mechanisms, error codes, and usage examples.
- Interactive Documentation: Interactive documentation tools like Swagger or OpenAPI allow developers to explore the API, make requests, and view responses in a user-friendly interface. They generate documentation based on annotations in the code or a separate specification file.
- Code Examples: Including code examples in multiple programming languages can help developers understand how to interact with the API. Examples may cover authentication, making requests, and handling responses.
- Tutorials and Guides: Additional tutorials or guides can provide step-by-step instructions on common use cases, best practices, and integration with other systems.
- Changelog: Maintaining a changelog or release notes helps API consumers track changes, bug fixes, and new features introduced in different versions. It helps in managing backward compatibility and communicating changes to developers.

Documenting APIs ensures clarity, reduces ambiguity, and facilitates adoption by developers. It serves as a reference for understanding API capabilities, making correct requests, and handling responses effectively.

There are various tools available to assist in API documentation, such as Swagger, RAML, OpenAPI, and tools provided by API management platforms like Postman or Apigee.

Remember to keep the documentation up to date as the API evolves, and provide clear instructions on versioning, deprecation, and migration paths to help API consumers transition smoothly to new versions.

CONSUMING RESTFUL APIS WITH RESTTEMPLATE OR WEBCLIENT

Consuming RESTful APIs is a common task in modern application development. Two popular ways to interact with RESTful APIs in Java Spring applications are by using **RestTemplate** or **WebClient**. Both options allow you to make HTTP requests to remote servers, retrieve data, and handle responses. Let's explore each of them:

- RestTemplate: **RestTemplate** is a synchronous HTTP client provided by the Spring Framework. It simplifies communication with RESTful services by abstracting away the complexities of making HTTP requests and handling responses. Here's how you can use **RestTemplate** to consume a RESTful API:

Step 1: Create a **RestTemplate** instance:

```
RestTemplate restTemplate = new RestTemplate();
```

Step 2: Make HTTP requests: You can use various methods like **getForObject**, **getForEntity**, **postForObject**, **exchange**, etc., to make different types of HTTP requests and get responses in different formats (JSON, XML, etc.).

```
String url = "https://api.example.com/data";
ResponseEntity<MyObject> response = restTemplate.getForEntity(url, MyObject.class);
MyObject myObject = response.getBody();
```

Step 3: Handle Responses: You can then process the data received in the response, handle errors, and perform appropriate actions based on the API's behavior.

RestTemplate is easy to use and suitable for simple use cases. However, it is a synchronous client, which means it may block the execution of your application until the response is received, potentially impacting the performance of your application.

- WebClient: **WebClient** is a part of the Spring WebFlux framework, which is used for building reactive applications. It provides a non-blocking, asynchronous way to consume RESTful APIs. Here's how you can use **WebClient**:

Step 1: Create a **WebClient** instance:

```
WebClient webClient = WebClient.create("https://api.example.com");
```

Step 2: Make HTTP requests: **WebClient** provides several methods for making HTTP requests, such as **get**, **post**, **put**, etc. Each of these methods returns a **Mono** or **Flux**, which represents the result of the asynchronous operation.

```
webClient.get()
  .uri("/data")
  .retrieve()
  .bodyToMono(MyObject.class)
  .subscribe(myObject -> {
    // Handle the received data
  });
```

Step 3: Handle Responses: With **WebClient**, you handle responses asynchronously using the reactive programming model. Instead of blocking the execution thread, you can subscribe to the **Mono** or **Flux** to receive the response data once it is available.

WebClient is more suitable for reactive applications that benefit from non-blocking I/O and handling multiple concurrent requests efficiently.

In summary, **RestTemplate** is a traditional synchronous HTTP client and is straightforward to use. On the other hand, **WebClient** is a more modern approach and is suitable for building reactive applications. The choice between the two depends on your application's requirements and whether you want to embrace reactive programming.

DAY 4: REVIEW AND KEY POINTS TO NOTE IN THIS MODULE

1. Request mappings and HTTP methods in Spring MVC:
   - Spring MVC uses annotations like **@RequestMapping** and **@GetMapping** to define mappings between URLs and methods in the controller.
   - HTTP methods (GET, POST, PUT, DELETE) determine the type of operation to be performed on a resource.
   - The **@PathVariable** annotation can be used to extract variables from the URL path.
   - Query parameters can be accessed using the **@RequestParam** annotation.
2. Handling request and response bodies (JSON):
   - Use the **@RequestBody** annotation to bind the request body to a method parameter.
   - Use the **@ResponseBody** annotation to convert the return value of a method to the response body.
   - Spring provides built-in support for JSON serialization and deserialization using libraries like Jackson.
   - Customize JSON processing using annotations like **@JsonIgnore**, **@JsonView**, and **@JsonFormat**.
3. Validation and error handling in RESTful APIs:
   - Use validation annotations like **@NotNull**, **@Size**, and **@Pattern** to enforce data constraints.
   - Handle validation errors using the **BindingResult** object and return appropriate error responses.
   - Customize error handling using the **@ControllerAdvice** annotation and exception handling techniques.
   - Return consistent error responses with meaningful error messages, status codes, and error object structures.
4. HATEOAS (Hypermedia as the Engine of Application State):
   - HATEOAS enables clients to navigate the API by including hyperlinks in response representations.
   - Use the **Link** class to create hyperlinks and the **EntityModel** class to represent resources with associated links.
   - Spring HATEOAS provides additional abstractions like **LinkBuilder** and **ResourceAssembler** to simplify HATEOAS implementation.
   - Incorporate HATEOAS in RESTful APIs to enhance discoverability, reduce coupling, and improve client-server interaction.
5. Versioning and documentation of RESTful APIs:
   - API versioning allows managing changes while ensuring backward compatibility.
   - Different versioning approaches include URL-based versioning, request header-based versioning, and media type-based versioning.
   - Document APIs using tools like Swagger or Springfox to provide comprehensive API documentation.
   - Documenting APIs helps developers understand endpoints, request/response structures, and available resources.

6. Consuming RESTful APIs with RestTemplate or WebClient:
   - RestTemplate is a synchronous HTTP client provided by Spring for consuming RESTful APIs.
   - WebClient is a non-blocking HTTP client introduced in Spring WebFlux for asynchronous API consumption.
   - Use RestTemplate's methods like **getForObject**, **postForObject**, etc., to send requests and handle responses.
   - WebClient provides a fluent and reactive API for making API calls, handling responses, and processing streams of data.

EXTRA LEARNING MATERIALS

https://docs.spring.io/spring-data/jpa/docs/2.2.5.RELEASE/reference/html/#reference

https://spring.io/guides/gs/rest-service/

https://www.baeldung.com/rest-template

https://www.baeldung.com/spring-webclient-resttemplate

https://www.mulesoft.com/resources/api/what-is-rest-api-design
https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/

https://howtodoinjava.com/spring-boot-tutorials/

https://www.toptal.com/java/spring-boot-rest-api-error-handling
https://mkyong.com/spring-boot/spring-rest-error-handling-example/