**Annotations and Class in Spring Boot**

**1. \*\*`@GetMapping`\*\*:** Use this annotation when you want to handle HTTP GET requests. GET requests are typically used to retrieve data or resources from the server. Here are common scenarios for using `@GetMapping`:

- Retrieving data: Use it to fetch data from the server. For example, getting a list of products, fetching user profiles, or reading blog posts.

- Idempotent operations: Use it for operations that don't modify server state, meaning you can perform the same GET request multiple times without causing side effects.

- Bookmarkable URLs: GET requests are suitable for generating bookmarkable URLs because they are safe and can be easily cached.

**2. \*\*`@PostMapping`\*\*:** Use this annotation when you want to handle HTTP POST requests. POST requests are used to submit data to the server for processing. Here are common scenarios for using `@PostMapping`:

- Creating resources: Use it to create new resources on the server. For example, registering a new user account, adding a new product to a catalog, or submitting a comment.

- Non-idempotent operations: Use it for operations that modify server state and aren't safe to repeat multiple times without side effects.

- Data submission: POST requests are appropriate when you need to send data to the server that isn't suitable for including in the URL (e.g., form data or JSON payloads).

In summary, use `@GetMapping` for read-only operations that retrieve data, and use `@PostMapping` for operations that create or modify resources on the server.

**3. `@PutMapping` : -**

The `@PutMapping` annotation in Spring MVC is used to map HTTP PUT requests to specific controller methods. It is part of the HTTP methods used in the RESTful architectural style. Here's when you should use `@PutMapping`:

1. \*\*Updating Resources\*\*:

- Use `@PutMapping` when you want to update an existing resource on the server.

2. \*\*Idempotent Operations\*\*:

- The HTTP PUT method is considered idempotent, which means that making multiple identical PUT requests should have the same effect as a single request. It's suitable for operations where repetition does not lead to different outcomes.

3. \*\*Updating Specific Resource\*\*:

- When you have a specific resource identified by a unique identifier (e.g., an ID in the URL), you can use `@PutMapping` to update that particular resource. The unique identifier helps the server identify which resource to update.

4. \*\*Complete Resource Replacement\*\*:

- `@PutMapping` is often used when you need to send a complete representation of the resource, including all its fields. It's expected that the entire resource will be replaced with the new representation sent in the request.

**4. ResponseEntity : -** ResponseEntity is a class in the Spring Framework that represents the entire HTTP response, including the status code, headers, and body. It's often used in Spring applications, particularly in Spring MVC and Spring Boot, to provide more control over the HTTP response sent back to clients.

HTTP Response: An HTTP response is what a server sends back to a client (e.g., a web browser or another application) in response to an HTTP request. It consists of a status code, headers, and an optional body (content). The status code indicates the result of the request (e.g., success, failure, redirection), headers provide additional metadata, and the body contains the data or content being sent.

In Spring Boot, `@RequestBody` and `@Valid` are annotations used in controller methods to handle and validate incoming HTTP request data, particularly in the context of RESTful APIs.

**5. \*\*`@RequestBody` Annotation\*\*:** Typically, you use `@RequestBody` in a controller method to extract data sent by the client in the request body. This data is often in JSON or XML format.

Certainly! In simple language, the `@RequestBody` annotation in Spring Boot is like opening a gift box and taking out the present inside.

Here's a breakdown:

1. \*\*Gift Box\*\*: Imagine an HTTP request as a gift box. This box can contain something special, like a toy.

2. \*\*`@RequestBody`\*\*: When you use `@RequestBody`, you're telling Spring Boot to open the gift box (the HTTP request) and take out the special thing inside (the data in the request body). This data is often in a format like JSON or XML and represents information sent by a client to the server.

3. \*\*Controller Method\*\*: You use `@RequestBody` in a controller method to specify that you want to receive and work with the special thing inside the gift box (the request data).

Here's a simple example:

## @PostMapping("/create")

## public ResponseEntity<User> createUser(@RequestBody User newUser) {

## // Here, newUser contains the special data from the request body

## // You can now use this data to create a new user in your application

## // ...

## }

## ```

In this example, when a client sends an HTTP POST request with a JSON object representing a user, the `@RequestBody` annotation tells Spring Boot to extract that user data from the request and provide it as the `newUser` parameter in the `createUser` method. This allows you to work with the user data inside the method and perform actions like creating a new user account.

**6. \*\*`@Valid` Annotation\*\*:** The `@Valid` annotation is used in conjunction with bean validation to indicate that the annotated object should be validated according to validation constraints defined in the Java Bean Validation API (JSR 303).

You use `@Valid` when you want to apply validation rules to the properties of an object, such as checking if fields are required, validating data formats, or enforcing custom validation logic.

To use `@Valid`, you also need to configure a validation provider in your project, such as Hibernate Validator, and define validation constraints on your model objects using annotations like `@NotNull`, `@Size`, `@Email`, etc.

By combining `@RequestBody` and `@Valid`, you can ensure that the data sent by clients in the request body is not only extracted but also validated against defined rules, helping you maintain data integrity and consistency in your Spring Boot application.

**7. Lombok: -**

Lombok is a popular Java library that helps reduce boilerplate code by automatically generating common code constructs, such as getters, setters, constructors, equals(), hashCode(), and toString(). It does this by using annotations in your Java code. Lombok simplifies code writing and maintenance, making it a valuable tool for Java developers.

Here's an explanation of the code and its annotations:

1. \*\*`@Data` Annotation\*\*:

- `@Data` is a Lombok annotation that automatically generates standard boilerplate code for the class, including getter and setter methods, `equals()`, `hashCode()`, and `toString()`. So, you don't need to write these methods manually.

2. \*\*`@NoArgsConstructor` Annotation\*\*:

- `@NoArgsConstructor` is another Lombok annotation that generates a no-argument constructor for the class. This constructor can be helpful when you need to create instances of the `UserDetails` class without providing constructor arguments.

3. \*\*Class Fields\*\*:

- The class contains various fields (instance variables) that represent user-related data. These fields include `id`, `firstName`, `lastName`, `email`, `countryCode`, `contactNo`, `gender`, `dob`, `password`, `kycStatus`, `redDate`, and `modifiedDate`. Each field corresponds to a specific piece of user information.

4. \*\*Validation Annotations\*\*:

- The class uses validation annotations to specify constraints on the fields, ensuring that the data meets certain criteria when it's received. For example:

- `@NotBlank` is used to ensure that the fields are not blank (empty or containing only whitespace).

- `@Length(min=3,max=15)` specifies the minimum and maximum lengths for the `firstName` and `lastName` fields.

- `@Pattern` is used with regular expressions to validate the format of the `email`, `countryCode`, `contactNo`, and `password` fields.

These validation annotations are typically used in conjunction with validation frameworks like Hibernate Validator to enforce data validation rules.

In summary, the `UserDetails` class is a data object that represents user-related information. Lombok annotations (`@Data` and `@NoArgsConstructor`) automatically generate getter and setter methods, as well as other common methods. The validation annotations are used to ensure that the data adheres to specific constraints when it's received or processed in your application.

**8. @PathVariable: -**

@PathVariable is an annotation used in Spring Framework-based applications, specifically in Spring MVC, to extract values from the URI (Uniform Resource Identifier) of a web request. It allows you to capture dynamic parts of the URL and use them as method parameters in your controller methods.

**9. \*\*@Controller\*\* and\*\*@RestController\*\*: -**

In Spring Framework, both `@Controller` and `@RestController` are used to define classes that handle HTTP requests and map them to specific methods within the class. However, there is a key difference between these two annotations:

1. \*\*@Controller\*\*:

- The `@Controller` annotation is used to define a class as a controller in a Spring MVC (Model-View-Controller) application.

- Controllers annotated with `@Controller` are typically used for handling traditional web pages and returning views (HTML pages) to the client.

- Controller methods usually return a `String` representing a view name or a `ModelAndView` object that includes both data and the view name.

- When using `@Controller`, you often work with view resolvers to render HTML templates, and the response typically contains HTML content.

2. \*\*@RestController\*\*:

- The `@RestController` annotation is a specialized version of `@Controller` designed for building RESTful web services.

- Controllers annotated with `@RestController` are primarily used for handling HTTP requests and returning data (usually in JSON or XML format) rather than rendering views.

- Controller methods in a `@RestController` return the actual data that should be serialized into the HTTP response body.

- Spring automatically serializes the returned data to JSON (by default) or other formats based on the request's `Accept` header.

In summary, the main difference is that `@Controller` is used for building web pages and returning views, while `@RestController` is used for creating RESTful APIs that return data in a format like JSON. The choice between them depends on whether you are building web pages (use `@Controller`) or exposing data services (use `@RestController`).

**10. ModelMapper: -**

`ModelMapper` is a Java library that provides a convenient way to map data from one Java object (or DTO - Data Transfer Object) to another. It simplifies the process of copying data from one object's fields to another object's fields, even when the objects have different structures. Key features and use cases include:

- \*\*Object Mapping\*\*: ModelMapper allows you to map objects of different types by matching their properties based on their names and types.

- \*\*Mapping Configuration\*\*: You can customize how specific fields are mapped or ignored using configuration settings.

- \*\*Nested Mapping\*\*: It can handle nested objects, mapping complex object graphs.

- \*\*Mapping Collections\*\*: ModelMapper can map collections of objects.

- \*\*Type Conversion\*\*: It supports type conversion, allowing you to map between different data types.

- \*\*Fluent API\*\*: It provides a fluent API for mapping configuration.

**11. ObjectMapper: -**

`ObjectMapper` is part of the Jackson library, which is a popular library for working with JSON data in Java. `ObjectMapper` is specifically used for serialization (converting Java objects to JSON) and deserialization (converting JSON to Java objects). Key features and use cases include:

- \*\*JSON Serialization\*\*: It can convert Java objects into JSON format.

- \*\*JSON Deserialization\*\*: It can parse JSON data and create Java objects.

- \*\*Custom Serialization/Deserialization\*\*: You can customize how objects are serialized or deserialized using annotations or custom serializers/deserializers.

- \*\*Handling Polymorphism\*\*: It supports polymorphic types and can serialize/deserialize objects with type information.

In summary, `ModelMapper` is primarily used for mapping data between Java objects with different structures, while `ObjectMapper` is used for serializing Java objects to JSON and deserializing JSON into Java objects. They serve different purposes and are commonly used in different scenarios, but both are valuable tools for data manipulation in Java applications.

**12. Differences between `ModelMapper` and `BeanUtils.copyProperties: -**

1. Configuration: -

- `ModelMapper`: It's like having a tool that can understand and follow detailed instructions. You can tell it exactly how to copy data from one thing to another.

- `BeanUtils.copyProperties()`: It's like a basic copying machine. It copies things from one place to another without much control. You just give it the things to copy, and it does its best to copy them.

2. Type Conversion: -

- `ModelMapper`: It's good at figuring out how to change something into a different kind of thing if needed.

- `BeanUtils.copyProperties()`: It might get confused if you ask it to copy something that's not exactly the same type as what it expects.

3. Fluent API: -

- `ModelMapper`: It speaks a clear and easy-to-understand language for setting up how things should be copied.

- `BeanUtils.copyProperties()`: It's more like straightforward copying without fancy explanations.

4. Nesting and Complex Mapping: -

- `ModelMapper`: It can copy things inside other things (like a box inside another box) without much trouble.

- `BeanUtils.copyProperties()`: It's better for simple copying and might need extra help for copying things inside other things.

5. Community and Maintenance: -

- `ModelMapper`: It's like a tool that many people are using and keeping up-to-date, so it stays in good shape.

- `BeanUtils.copyProperties()`: It's an older tool that still works but doesn't get as much attention these days.

In short, if you have simple copying to do, `BeanUtils.copyProperties()` is like a basic copying machine. But if you need more control and want to handle different types and complex copying, `ModelMapper` is like a smart assistant that can follow detailed instructions and convert things if needed. Choose the one that fits your copying needs best!

13.