**R and D on Mapping in SpringBoot**

Mapping is not just a technical feature in Spring Boot but a fundamental aspect of how web applications interact with clients. It encapsulates the principles of REST (Representational State Transfer), a widely adopted architectural style for building scalable and stateless web services.

**The Role of Mapping in RESTful Architecture**

In RESTful architecture, resources (such as users, orders, products) are identified by URIs (Uniform Resource Identifiers). Each resource is accessed and manipulated through a standard set of HTTP methods (GET, POST, PUT, DELETE). Mapping in Spring Boot aligns with this paradigm by connecting these HTTP methods to Java methods in a controller class.

1. **Resource Identification**:

**Concept**: Resources in REST are the core entities that an application manages, such as a User or Product.

**Mapping Implementation**: In Spring Boot, URIs define the resource location (e.g., /api/users/{id}), while mapping annotations specify which HTTP method to use for interacting with these resources.

1. **Stateless Communication**:

**Concept**: RESTful services are stateless, meaning each request from a client to a server must contain all the information needed to understand and process the request.

**Mapping Implementation**: Spring Boot mappings ensure that each endpoint can handle requests independently, relying on the request's content and URI rather than server state.

1. **Uniform Interface**:

**Concept**: REST emphasizes a uniform interface that simplifies and decouples the architecture, enabling each part to evolve independently.

**Mapping Implementation**: By using standard annotations like

@GetMapping, @PostMapping, @PutMapping, and @DeleteMapping, Spring Boot ensures a consistent and predictable interface for clients interacting with the API.

Mapping is a core concept in Spring Boot that defines how HTTP requests are translated into actions within your application. By using mappings, we can control which controller methods are invoked for specific endpoints, which HTTP methods are supported, and how data is passed between the client and the server.

**There are also HTTP method specific shortcut variants of** @RequestMapping:

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping
* @PatchMapping

# RequestMapping

Flexible and Versatile: @RequestMapping is the most flexible annotation, allowing you to define mappings at both the class and method levels. It can handle multiple HTTP methods and can be customized with various attributes such as path, params, headers, and more.

* **Usage**: General-purpose mapping annotation used at the class or method level.
* **Attributes**: path, method, params, headers, consumes, produces.

**Example**:

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

public List<User> getAllUsers() {

}

**2. GetMapping**

* **Usage**: Shortcut for @RequestMapping(method = RequestMethod.GET).
* **Example**:

GetMapping("/users/{id}")

public User getUserById(@PathVariable Long id) {

}

**3. PostMapping**

• **Usage**: Shortcut for @RequestMapping(method = RequestMethod.POST).

**Example**:

PostMapping("/users")

public User createUser(@RequestBody User user) {

}

1. **PutMapping**

o **Usage**: Shortcut for @RequestMapping(method = RequestMethod.PUT).

**Example**:

PutMapping("/users/{id}")

public User updateUser(@PathVariable Long id, @RequestBody User user) {

}

1. **DeleteMapping**

o **Usage**: Shortcut for @RequestMapping(method = RequestMethod.DELETE).

o **Example**:

@DeleteMapping("/users/{id}")

public void deleteUser(@PathVariable Long id) {

}

…………………………………………………………………………

**# # Path Variables and Query Parameters**

* **@PathVariable**
* **Usage**: Extracts values from the URI.
* **Example**: @GetMapping("/users/{id}")

public User getUser(@PathVariable Long id) {

}

**@RequestParam**

**Usage**: Extracts query parameters from the URI.

**Example**:

@GetMapping("/users")

public List<User> getUsersByAge(@RequestParam int age) {

}

………………………………………………………………

**# # Request and Response Bodies**

**@RequestBody**

**Usage**: Maps the body of the request to a method parameter.

**Example**: @PostMapping("/users")

public User createUser(@RequestBody User user) {

}

**@ResponseBody**

**Usage**: Maps the method return value to the HTTP response body.

**Example**:

@GetMapping("/users/{id}")

@ResponseBody

public User getUser(@PathVariable Long id) {

}

………………………………………………………………………

**# # Error Handling**

• **@ExceptionHandler**

o **Usage**: Handles exceptions thrown by request handlers.

o **Example**:

@ExceptionHandler(UserNotFoundException.class)

public ResponseEntity<String> handleUserNotFound(UserNotFoundException ex)

{

return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT\_FOUND);

}

………………………………………………………………………..

**6. Cross-Origin Resource Sharing (CORS)**

* **@CrossOrigin**

**Usage**: Enables CORS on specific methods or entire classes.

**Example**:

@CrossOrigin(origins = "http://example.com")

@GetMapping("/users") public List<User> getUsers() {

}

……………………………………………………………

# # #Advanced Mapping

* Matrix Variables: A less common form of URI variables, typically used in complex path structures.
* Content Negotiation: Mapping based on the Accept header to serve different formats like JSON, XML.

Mapping in Spring Boot is a powerful and flexible feature that allows developers to build robust and maintainable web applications. By understanding the various mapping annotations and best practices, we can create APIs that are easy to use, secure, and scalable.