Class-22: Building CRUD REST Endpoints in Spring Boot

by Pial Kanti Samadder

An Overview of HTTP Methods

- **GET:** Fetches data. It is safe and idempotent, which means repeated requests produce the same result. It only retrieves data and has no other side effects.
- **POST:** Submits data to be processed. It’s neither safe nor idempotent. Typically used for creating resources.
- **PUT:** Updates data. Idempotent but not safe. Typically used for updating existing resources or creating them if they don’t exist.
- **DELETE:** Removes data. Idempotent but not safe. Used for deleting resources.

Annotation for handling Request Mapping

Spring 4.3. introduced some very cool method-level composed annotations to smooth out the handling *@RequestMapping* in typical Spring MVC projects.

Traditional Approach

Typically, if we want to implement the URL handler using traditional *@RequestMapping* annotation, it would have been something like this:

```
@RequestMapping(value = "/get/{id}", method = RequestMethod.GET) // Or RequestMethod.POST
```

New Annotations

Spring currently supports five types of inbuilt annotations for handling different types of incoming HTTP request methods which are

1. *GET* - `@GetMapping`
2. *POST* - `@PostMapping`
3. *PUT* - `@PutMapping`
4. *DELETE* - `@DeleteMapping`
5. *PATCH* - `@PatchMapping`

@GetMapping

This annotation is a shortcut for `@RequestMapping(method = RequestMethod.GET)`.

When you want to handle a GET request, you use `@GetMapping`.

Example:

```
@RestController
public class UserController {

    @GetMapping("/users/{id}")
    public User getUser(@PathVariable Long id) {
        // Logic to fetch a user by id
        return userService.getUserById(id);
    }
}
```

Path Variable vs Query Parameter

Purpose

Concept	Used For
Path Variable	To identify a **specific resource**
Query Parameter	To **filter**, **sort**, or **search** resources

URL Structure Example

Type	Example URL	Typical Use
Path Variable	`/users/42`	Get user by ID
Query Parameter	`/users?role=admin&active=true`	Filter users by role and status

Spring Boot Usage

```
// Path Variable Example
@GetMapping("/users/{id}")
public User getUser(@PathVariable Long id) { ... }

// Query Parameter Example
@GetMapping("/users")
public List<User> getUsers(@RequestParam String role,
                           @RequestParam boolean active) { ... }
```

Key Differences

Aspect	Path Variable	Query Parameter
Purpose	Identify a specific resource	Filter or modify a collection
Required	Usually **required**	Often **optional**
Position	Part of the **URI path**	After a `?` in the URL
Used with	`@PathVariable`	`@RequestParam`

Example in Action

Operation	URL	Meaning
Get one user	`/users/42`	Retrieve user with ID 42
Get filtered users	`/users?country=US&active=true`	Retrieve active users in the US

@PostMapping

This is an alternative to `@RequestMapping(method = RequestMethod.POST)`.

Used mainly for creating new resources.

Example:

```
@RestController
public class UserController {

    @PostMapping("/users")
    public User createUser(@RequestBody User user) {
        // Logic to save a user
        return userService.saveUser(user);
    }
}
```

@PutMapping

This is an alternative to `@RequestMapping(method = RequestMethod.PUT)`.

Used mainly for updating an existing resources.

Example:

```
@RestController
public class UserController {

    @PutMapping("/users/{id}")
    public User updateUser(@PathVariable id, @RequestBody User user) {
        // Logic to update a user
        return userService.updateUser(user);
    }
}
```

@DeleteMapping

shorthand for `@RequestMapping(method = RequestMethod.DELETE)`.

This annotation is utilized when you want to delete a specific resource.

Example:

```
@RestController
public class UserController {

    @DeleteMapping("/users/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        // Logic to delete a user by id
        userService.deleteUser(id);
        return ResponseEntity.noContent().build();
    }
}
```

Combining Annotations

Remember, these annotations are merely shortcuts. They can be combined with other annotations to achieve more specific behaviors. For example, you can combine `@PostMapping` with `@ResponseStatus` to indicate a specific HTTP status code should be returned:

```
@PostMapping("/users")
@ResponseStatus(HttpStatus.CREATED)
public User createUser(@RequestBody User user) {
    return userService.saveUser(user);
}
```

Returning `ResponseEntity` in Spring Boot

`ResponseEntity` represents the entire HTTP response, including:

- Response body
- HTTP status code
- Optional headers

Used for **full control** over what your API returns.

Why Use ResponseEntity?

Reason	Description
✅ Control	Set custom HTTP status codes
🧱 Headers	Add response headers (e.g., `Location`)
💬 Consistency	Standardize all API responses
🔍 Clarity	Explicitly show intent (success, not found, created, etc.)

Example: Returning ResponseEntity

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@RequestBody User user) {
    User savedUser = userService.save(user);

    // Return 201 Created + new resource
    return ResponseEntity
            .status(HttpStatus.CREATED)
            .body(savedUser);
}
```

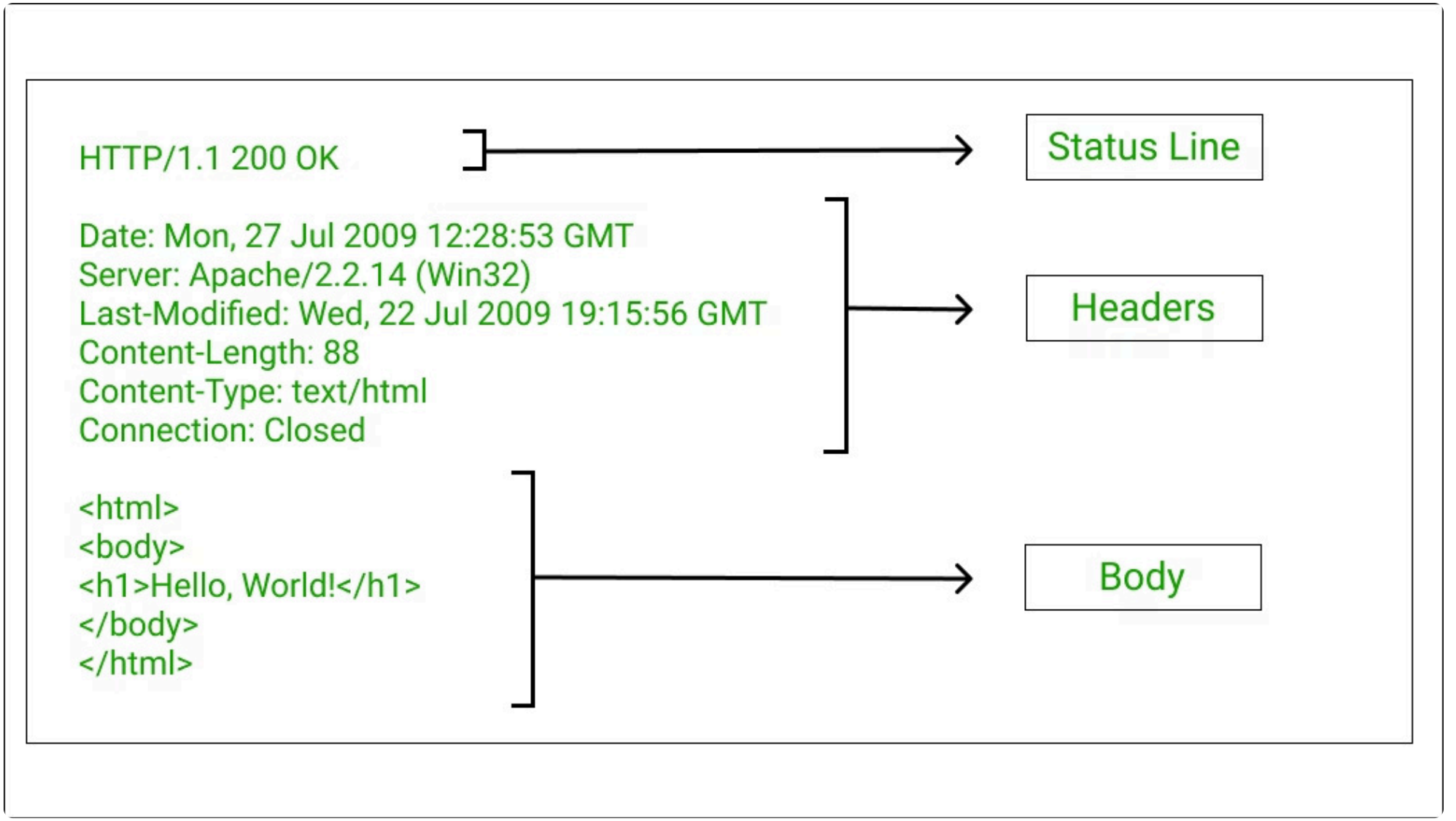
➡️ Returns:

- `HTTP 201 Created`
- JSON body with the created user

Best Practices

- Always return `ResponseEntity` from controller methods (not raw objects)
- Use proper **HTTP status codes** for clarity
- Return DTOs instead of entities
- Avoid returning `null` → use `ResponseEntity.notFound()` or `noContent()`
- Centralize response formats if possible (e.g., a standard API response wrapper)

Structure of HTTP Response



The Status Line contains three important components - HTTP Version, HTTP Response Code, and a Reason-Phrase.

- **HTTP Version:** The HTTP version number shows the HTTP specification to which the server has tried to make the response message comply. In the above example, **1.1** is the HTTP Version.
- **HTTP Response Code:** It is a 3 digit number that shows the conclusion of the Request. In the above example, the response code **200** denotes that the content requested was OK. A very popular Status Code that we frequently encounter is 404 which represents the requested resource was not found.
- **Reason-Phrase:** Also known as Status Text as it summarizes the Status Code in human-readable form.

Why Validate Request Bodies?

Purpose of Validation:

- Ensure API receives **valid and complete data**
- Prevent **invalid or malicious inputs** from reaching business logic
- Provide **clear error messages** to clients

Example:

- User creation API requires `name`, `email`, and `age > 18`

Basic Bean Validation

Jakarta Bean Validation Annotations:

Annotation	Purpose	Example
`@NotNull`	Field must not be null	`@NotNull private String name;`
`@Size`	Restrict string length	`@Size(min=3, max=50) private String username;`
`@Email`	Validate email format	`@Email private String email;`
`@Min/@Max`	Numeric range	`@Min(18) @Max(100) private int age;`

Usage:

```
public class UserDTO {
   @NotNull(message = "Name is required")
   private String name;

   @Email(message = "Invalid email")
   private String email;

   @Min(18) private int age;
}
```

Using @Valid in Controller

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@Valid @RequestBody UserDTO userDTO) {
    User savedUser = userService.save(userDTO);
    return ResponseEntity.status(HttpStatus.CREATED).body(savedUser);
}
```

Explanation:

- `@Valid` triggers **bean validation** before entering method body
- Spring automatically returns **400 Bad Request** if validation fails (with default error messages)

Custom Validation Messages

Example:

```
@NotNull(message = "Username cannot be empty")
@Size(min=3, max=20, message = "Username must be 3-20 characters")
private String username;
```

Tips:

- Customize messages for better clarity to clients
- Can also use **properties file** for internationalization

Nested Object Validation

DTO Example:

```
public class Address {
    @NotNull private String street;
    @NotNull private String city;
}

public class User {
    @NotNull private String name;
    @Valid private Address address; // Validate nested object
}
```

Explanation:

- Use `@Valid` on nested objects
- Ensures full validation of complex request bodies

Handling Validation Results

Optional BindingResult Usage:

```
@PostMapping("/users")
public ResponseEntity<?> createUser(@Valid @RequestBody UserDTO userDTO,
                                    BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return ResponseEntity.badRequest().body(bindingResult.getAllErrors());
    }
    return ResponseEntity.ok(userService.save(userDTO));
}
```

Note:

- For this module, just **show the concept**
- Detailed exception handling is covered in a separate module

Spring handles errors automatically, but you can inspect with `BindingResult`.

Code Reference

GitHub URL: https://github.com/PialKanti/Ostad-SpringBoot-Course

Instructions to clone and run project using IntelliJ Idea:

- Clone the git repo via command (git needed to be installed in your machine)
  `git clone https://github.com/PialKanti/Ostad-SpringBoot-Course.git`
- Switch to today's class branch:
  `git fetch`
  `git switch class-22-crud`
- Open `crud-sample` using *IntelliJ Idea*