REST API Development Notes

REST Basics – HTTP Methods & Status Codes

1. What is REST?

- **REST (Representational State Transfer)** is an architectural style for building scalable and lightweight web services.
- It uses **HTTP protocol** for communication between **Client** and **Server**.
- A **RESTful API** follows principles like **statelessness, resource identification via URI, and uniform interface**.

---

2. HTTP Methods in REST API

Each HTTP method corresponds to a **CRUD** operation:

HTTP Method	Operation (CRUD)	Description	Example
GET	Read	Retrieve resource(s) from the server.	GET /users → Fetch all users
POST	Create	Create a new resource on the server.	POST /users → Add a new user
PUT	Update (Full)	Replace an existing resource completely.	PUT /users/1 → Update user with ID 1
PATCH	Update (Partial)	Update only specific fields of a resource.	PATCH /users/1 → Update email only
DELETE	Delete	Remove a resource from the server.	DELETE /users/1 → Delete user with ID 1
OPTIONS	Meta	Get supported operations for a resource.	OPTIONS /users
HEAD	Header only	Same as GET but retrieves headers only (no body).	HEAD /users

---

3. HTTP Status Codes

REST APIs use **standard HTTP status codes** to indicate the result of an operation.

□ Success Codes

Code	Meaning	When Used
200 OK	Request succeeded	GET /users returns user list
201 Created	Resource created successfully	POST /users created a new user
202 Accepted	Request accepted for processing (async)	Long-running process like file upload
204 No Content	Request successful but no response body	DELETE /users/1

□ Client Error Codes

Code	Meaning	When Used
400 Bad Request	Invalid request syntax	Missing required fields
401 Unauthorized	Authentication required	User not logged in
403 Forbidden	Authenticated but not allowed	User without admin role trying to delete
404 Not Found	Resource not found	GET /users/99 but user 99 doesn’t exist
409 Conflict	Conflict in request	Creating a user with duplicate email

□ Server Error Codes

Code	Meaning	When Used
500 Internal Server Error	Generic server-side error	Unhandled exception
502 Bad Gateway	Invalid response from upstream server	Proxy/gateway issues
503 Service Unavailable	Server is down/overloaded	Maintenance mode

Code	Meaning	When Used
504 Gateway Timeout	Upstream server took too long	Slow backend service

---

 Key Notes

- Use **correct HTTP methods** to make your API predictable and RESTful.
- Always return **proper status codes** to help clients handle responses better.
- Follow **statelessness**: each request must contain all necessary data (no session state stored on server).

Creating REST Controllers (@RestController, @RequestMapping)

What is a REST Controller?

- In Spring Boot, REST APIs are created using **controllers**.
- A controller is a class that handles **HTTP requests** and returns **HTTP responses**.
- REST Controllers are annotated with @RestController (a shortcut for @Controller + @ResponseBody).

@RestController

- Used to mark a class as a RESTful **web service controller**.
- Ensures that **methods return JSON/XML** (instead of rendering a view).
- Example:

```
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.GetMapping;

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, REST API!";
    }
}
```

Here:

- @RestController → Marks the class as a REST controller.
- @GetMapping("/hello") → Maps GET requests on /hello to sayHello() method.

@RequestMapping

- Used at **class level** or **method level** to map URLs to controller methods.
- Can handle multiple HTTP methods using method attribute.

Example with Class + Method Level

```
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/users")  // Base URL for all methods
public class UserController {

    // GET /api/users
    @GetMapping
    public String getAllUsers() {
        return "List of users";
    }

    // GET /api/users/1
    @GetMapping("/{id}")
    public String getUserById(@PathVariable int id) {
        return "User with ID: " + id;
    }

    // POST /api/users
    @PostMapping
    public String createUser() {
        return "User created!";
    }
}
```

□ Key Points:

- @RequestMapping("/api/users") → Defines base path for all APIs inside the class.
- @GetMapping, @PostMapping, @PutMapping, @DeleteMapping → Specializations of @RequestMapping.
- @PathVariable → Extracts value from URI (like id in /users/1).

□ Key Notes

- Always use @RestController for REST APIs (instead of @Controller).
- Use **class-level @RequestMapping** for grouping APIs under one path (like /api/users).
- Use **method-level mappings** (@GetMapping, @PostMapping, etc.) for clarity.
- Return **JSON objects** (using DTOs/POJOs) instead of plain Strings in real-world APIs.

Handling Requests in Spring REST API

Introduction

- In REST APIs, each HTTP method is mapped to a **Java method** inside a @RestController.
- Spring provides **specialized annotations** for each request type:
  - @GetMapping → Read data
  - @PostMapping → Create data
  - @PutMapping → Update entire data
  - @DeleteMapping → Remove data

---

1. @GetMapping (READ)

- Used to **fetch data** from the server.
- Equivalent to @RequestMapping(method = RequestMethod.GET).

 Example:

```
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/products")
public class ProductController {

    // GET /api/products
    @GetMapping
    public String getAllProducts() {
        return "Returning all products";
    }

    // GET /api/products/101
    @GetMapping("/{id}")
    public String getProductById(@PathVariable int id) {
        return "Product with ID: " + id;
    }
}
```

 Notes:

- Use @PathVariable to extract **URI path parameters**.
- Can also use @RequestParam for **query parameters**.

---

2. @PostMapping (CREATE)

- Used to **create a new resource**.
- Typically consumes **JSON request body**.

 Example:

```
@PostMapping
public String addProduct(@RequestBody Product product) {
    return "Product added: " + product.getName();
}
```

 Notes:

- @RequestBody → Maps incoming JSON to a **Java object (POJO)**.
- Return **201 Created** status using ResponseEntity.

---

3. @PutMapping (UPDATE – Full Update)

- Used to **update an existing resource completely**.

Example:

```
@PutMapping("/{id}")
public String updateProduct(@PathVariable int id, @RequestBody Product
product) {
   return "Product " + id + " updated with name: " + product.getName();
}
```

 Notes:

- Replaces the **entire object** with the provided one.
- Good practice: first **check if resource exists** before updating.

---

4. @DeleteMapping (DELETE)

- Used to **remove a resource**.

 Example:

```
@DeleteMapping("/{id}")
public String deleteProduct(@PathVariable int id) {
   return "Product " + id + " deleted!";
}
```

 Notes:

- Should return 204 No Content if successful.
- If resource doesn’t exist → return 404 Not Found.

□ Key Notes

- Always use **proper HTTP methods** for predictable APIs.
- Prefer **ResponseEntity<>** to send custom **status codes + body**.
- @GetMapping → Retrieve, @PostMapping → Create, @PutMapping → Replace, @DeleteMapping → Remove.
- Combine with @PathVariable and @RequestBody for dynamic requests.

Path Variables & Request Parameters in Spring REST

Introduction

When handling client requests in REST APIs, **data can be passed** to the server in two common ways:

1. **Path Variables** → Part of the **URL path**.
2. **Request Parameters** → Passed as **query parameters** (?key=value).

Spring provides two annotations:

- @PathVariable → Extracts values from the **URL path**.
- @RequestParam → Extracts values from the **query string**.

---

1. @PathVariable

- Used when a value is embedded in the **URI path**.
- Typical for identifying a **resource by ID**.

□ Example:

```
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/users")
public class UserController {

    // GET /api/users/10
    @GetMapping("/{id}")
    public String getUserById(@PathVariable("id") int userId) {
        return "User with ID: " + userId;
    }

    // Multiple path variables
    // GET /api/users/10/orders/5
    @GetMapping("/{userId}/orders/{orderId}")
    public String getUserOrder(@PathVariable int userId,
                               @PathVariable int orderId) {
        return "Order " + orderId + " of User " + userId;
    }
}
```

 Key Notes:

- If parameter name in method = variable name in URL → @PathVariable int id (no need for "id").
- Can use multiple path variables in a single mapping.

---

2. @RequestParam

- Used when values are passed as **query parameters** (after ?).
- Useful for filtering, searching, pagination, etc.

 Example:

```
// GET /api/users?role=admin&active=true
@GetMapping
public String getUsersByRole(@RequestParam String role,
                             @RequestParam(defaultValue = "false") boolean active) {
    return "Users with role: " + role + " | Active: " + active;
}

// Optional query param
// GET /api/users?page=2&size=10
@GetMapping("/paged")
public String getUsersPaged(@RequestParam(required = false, defaultValue = "1")
int page,
                    @RequestParam(required = false, defaultValue = "5") int size) {
    return "Page: " + page + ", Size: " + size;
}
```

 Key Notes:

- @RequestParam("paramName") → Maps query parameter to method argument.
- required = false → Makes the parameter optional.
- defaultValue → Provides a fallback if parameter is missing.

 PathVariable vs RequestParam

Feature	@PathVariable	@RequestParam
Location	Inside **URL path**	Inside **query string**
Example URL	/api/users/10	/api/users?id=10
Usage	Identify a specific resource	Filter, search, pagination
Common Use Case	Get resource by ID	Sorting, filtering

Request Body Handling (@RequestBody)

Introduction

- In REST APIs, the **client often sends data in the body** of an HTTP request (usually in **JSON or XML format**).
- Spring provides @RequestBody to automatically **deserialize JSON/XML → Java object (POJO)**.
- This is commonly used in **POST** and **PUT** requests.

@RequestBody

- Annotation placed on a method parameter.
- Tells Spring to **bind the HTTP request body** to that parameter.
- Uses **HttpMessageConverter** (like Jackson for JSON) under the hood.

Example:

```
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/products")
public class ProductController {

    // POST /api/products
    @PostMapping
    public String addProduct(@RequestBody Product product) {
        return "Product added: " + product.getName() + " - Price: " +
product.getPrice();
    }
}
```

Sample JSON Request:

```
{
  "name": "Laptop",
  "price": 75000
}
```

Flow:

1. Client sends JSON in request body.
2. Spring (via Jackson) converts JSON → Product object.
3. Method uses the Product object directly.

Using with PUT (Update Example)

```
@PutMapping("/{id}")
public String updateProduct(@PathVariable int id, @RequestBody Product
product) {
    return "Updated product with ID: " + id + " to " + product.getName();
}
```

□ Validation with @RequestBody

We can validate request body using **Java Bean Validation** (@Valid).

```
import jakarta.validation.Valid;

@PostMapping
public String addValidatedProduct(@Valid @RequestBody Product product) {
    return "Valid Product: " + product.getName();
}
```

Example POJO with Validation

```
import jakarta.validation.constraints.*;

public class Product {
    @NotBlank(message = "Name is required")
    private String name;

    @Min(value = 1, message = "Price must be greater than 0")
    private double price;

    // getters & setters
}
```

□ If validation fails → Spring throws MethodArgumentNotValidException (can be handled via @ExceptionHandler).

@RequestBody vs @RequestParam vs @PathVariable

Annotation	Data Source	Example Use
@RequestBody	JSON/XML in **body**	Create/Update resource (POST/PUT)
@RequestParam	**Query string** (?key=value)	Filters, pagination
@PathVariable	**Path segment** in URL	Identify resource (ID, name)

□ Key Notes

- @RequestBody is mainly used in **POST & PUT** for request data.
- Requires **Jackson (or other JSON library)** for conversion.
- Can be combined with **@Valid** for automatic validation.
- Always define a **POJO (DTO)** for request body → better maintainability.

Sending JSON/XML Responses (@ResponseBody)

Introduction

- In REST APIs, the server usually responds with **data** in formats like **JSON** or **XML**.
- Spring provides @ResponseBody to **directly return objects as JSON/XML** instead of rendering a view.
- It works with **HttpMessageConverter** (Jackson for JSON, JAXB for XML) to serialize Java objects.

@ResponseBody

- Annotation that tells Spring:

  > "The return value of this method should be **written directly to the HTTP response body** (not a view)."

- Often **not needed** in modern Spring Boot since @RestController already includes it implicitly.

Example – Returning JSON

```
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/products")
public class ProductController {

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

```
    public Product getProduct(@PathVariable int id) {
        return new Product(id, "Laptop", 75000);
    }
}
```

Sample POJO

```
public class Product {
    private int id;
    private String name;
    private double price;

    // constructor, getters, setters
}
```

Response (JSON)

```
{
  "id": 1,
  "name": "Laptop",
  "price": 75000
}
```

 Since we used @RestController, Spring auto-applies @ResponseBody.

Example – Returning XML

1. **Add Jackson XML dependency in pom.xml:**

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

2. **Annotate POJO with @XmlRootElement (for XML binding):**

```
import jakarta.xml.bind.annotation.XmlRootElement;

@XmlRootElement
```

```
public class Product {
    private int id;
    private String name;
    private double price;
    // getters & setters
}
```

3. **Controller:**

```
@GetMapping(value = "/{id}", produces = {"application/json",
"application/xml"})
public Product getProduct(@PathVariable int id) {
    return new Product(id, "Phone", 45000);
}
```

Response (XML)

```
<Product>
    <id>1</id>
    <name>Phone</name>
    <price>45000.0</price>
</Product>
```

 Content negotiation decides response type:

- Request with Accept: application/json → JSON.
- Request with Accept: application/xml → XML.

Using @ResponseBody Explicitly

```
@Controller
public class HelloController {

    @ResponseBody
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, REST API!";
    }
}
```

 Here we use @Controller instead of @RestController, so @ResponseBody is needed.

 Key Notes

- @RestController = @Controller + @ResponseBody (no need to use @ResponseBody separately).
- Response type (JSON/XML) depends on **HttpMessageConverters** + client's Accept header.
- Use produces = "application/json" or "application/xml" in mappings to specify response format.
- Always return **objects (POJOs, DTOs)** instead of raw Strings in production APIs.

Exception Handling in Spring REST APIs

Why Exception Handling?

- REST APIs must provide **clear error responses** instead of exposing internal errors (like stack traces).
- Example: instead of 500 Internal Server Error with raw Java exception → return structured JSON:

```
{
  "timestamp": "2025-08-16T22:10:00",
  "status": 404,
  "error": "Not Found",
  "message": "User not found",
  "path": "/api/users/10"
}
```

Spring provides:

- @ExceptionHandler → Handles specific exceptions at the controller level.
- @ControllerAdvice → Global exception handler across all controllers.

@ExceptionHandler (Controller-level)

- Handles exceptions **inside a specific controller**.

 Example:

```
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/users")
public class UserController {

    @GetMapping("/{id}")
    public String getUser(@PathVariable int id) {
        if (id <= 0) {
            throw new IllegalArgumentException("Invalid User ID");
        }
        return "User with ID: " + id;
    }

    // Handle IllegalArgumentException for this controller
    @ExceptionHandler(IllegalArgumentException.class)
    public String handleInvalidArgument(IllegalArgumentException ex) {
        return "Error: " + ex.getMessage();
    }
}
```

 Notes:

- The method annotated with @ExceptionHandler runs when the exception is thrown.
- Can return **custom messages, objects, or ResponseEntity**.

@ControllerAdvice (Global Exception Handling)

- Used to handle exceptions **globally across all controllers**.
- Keeps error-handling code separate from business logic.

 Example:

```
import org.springframework.web.bind.annotation.*;
import org.springframework.http.*;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String>
handleIllegalArgument(IllegalArgumentException ex) {
        return new ResponseEntity<>("Global Error: " + ex.getMessage(),
HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<String> handleRuntime(RuntimeException ex) {
        return new ResponseEntity<>("Something went wrong!",
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

□ **Notes:**

- Any IllegalArgumentException thrown anywhere will be caught here.
- Can define multiple handlers for different exceptions.

□ Structured Error Response (Best Practice)

Instead of returning plain Strings, return a **custom error response object**.

```
public class ErrorResponse {
    private String timestamp;
    private int status;
    private String error;
    private String message;
    private String path;

    // constructor, getters, setters
}
```

□ Global Handler Example:

```
@ExceptionHandler(Exception.class)
public ResponseEntity<ErrorResponse> handleAll(Exception ex, WebRequest
request) {
    ErrorResponse error = new ErrorResponse(
        LocalDateTime.now().toString(),
        HttpStatus.INTERNAL_SERVER_ERROR.value(),
        "Server Error",
        ex.getMessage(),
        request.getDescription(false)
    );
    return new ResponseEntity<>(error,
HttpStatus.INTERNAL_SERVER_ERROR);
}
```

Sample JSON Response:

```
{
  "timestamp": "2025-08-16T22:20:00",
  "status": 500,
  "error": "Server Error",
  "message": "NullPointerException occurred",
  "path": "/api/users"
}
```

□ Key Notes

- Use **@ExceptionHandler** for controller-specific exceptions.
- Use **@ControllerAdvice** for global exception handling.
- Always return **meaningful error responses** (JSON/XML).
- Prefer ResponseEntity<ErrorResponse> to send status codes + messages.
- Helps maintain **clean controllers** and improves **API usability**.

ResponseEntity & Custom Responses in Spring REST API

What is ResponseEntity?

- ResponseEntity<T> represents the **entire HTTP response**.
- It allows full control over:
  - **Body** → Response data (JSON/XML).
  - **Headers** → Metadata (e.g., Location, Authorization).
  - **Status Code** → 200, 201, 400, 404, etc.
- Unlike plain return values, ResponseEntity gives flexibility in **customizing API responses**.

Basic Example – Returning ResponseEntity

```
import org.springframework.http.*;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/products")
public class ProductController {

    @GetMapping("/{id}")
    public ResponseEntity<String> getProduct(@PathVariable int id) {
        if (id == 1) {
            return ResponseEntity.ok("Laptop");  // 200 OK
        } else {
            return ResponseEntity.status(HttpStatus.NOT_FOUND)
                                 .body("Product not found");  // 404
        }
    }
}
```

Notes:

- ResponseEntity.ok(body) → Shortcut for **200 OK**.
- ResponseEntity.status(HttpStatus.XYZ).body(body) → Custom status code.

Returning POJOs as JSON

```
@GetMapping("/{id}")
public ResponseEntity<Product> getProduct(@PathVariable int id) {
    Product product = new Product(id, "Phone", 45000);
    return ResponseEntity.ok(product);  // JSON response with 200 OK
}
```

Response (JSON):

```
{
  "id": 1,
  "name": "Phone",
  "price": 45000
}
```

Custom Headers with ResponseEntity

```
@PostMapping
public ResponseEntity<String> createProduct(@RequestBody Product product) {
    HttpHeaders headers = new HttpHeaders();
    headers.add("Custom-Header", "CreatedProduct");

    return ResponseEntity.status(HttpStatus.CREATED)
                         .headers(headers)
                         .body("Product created: " + product.getName());
}
```

□ **Notes:**

- Useful for adding **Location header** after creating a resource.
- Example: Location: /api/products/1 after creating a product.

Custom Error Responses (Best Practice)

Instead of plain text, return a **structured error object**.

Error DTO:

```
public class ErrorResponse {
    private int status;
    private String message;
    private LocalDateTime timestamp;

    public ErrorResponse(int status, String message) {
        this.status = status;
        this.message = message;
        this.timestamp = LocalDateTime.now();
    }

    // getters & setters
}
```

Example Usage:

```
@GetMapping("/{id}")
public ResponseEntity<?> getProduct(@PathVariable int id) {
    if (id != 1) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                             .body(new ErrorResponse(404, "Product not found"));
    }
    return ResponseEntity.ok(new Product(1, "Laptop", 75000));
}
```

Error JSON Response:

```
{
  "status": 404,
  "message": "Product not found",
  "timestamp": "2025-08-16T22:30:00"
}
```

□ Key Notes

- Use ResponseEntity<T> to **control status codes, headers, and response body**.
- Always return **meaningful structured responses** (DTOs, not Strings).

- For success → return ResponseEntity.ok(obj) or ResponseEntity.status(HttpStatus.CREATED).
- For failure → return structured ErrorResponse objects with proper status codes.
- Works best with **@ControllerAdvice + @ExceptionHandler** for global error handling.