Designing Production-Ready REST APIs with Spring Boot

Web Technologies — Lecture 10a (Deep Dive)

Masoud Hamad

State University of Zanzibar (SUZA)

Semester II, 2025/2026

Today's Agenda

What REST Actually Is

REST = *Representational State Transfer*.

An architectural style for **decoupled, stateless** communication between a client and a server.

Five guarantees we will rely on:

1. **Resource Orientation** — every business entity has a stable URL
2. **Statelessness** — the server stores no session memory
3. **Uniform Interface** — standard HTTP verbs + status codes
4. **Client–Server Separation** — UI and storage evolve independently
5. **Cacheability** — responses declare whether they can be reused

Statelessness is what lets us run 10 instances of our API behind a load balancer with no shared session store.

Safe vs. Idempotent — the matrix to memorise

HTTP verb	Safe?	Idempotent?	Effect on storage
`GET`	YES	YES	Read only
`HEAD`	YES	YES	Read only (no body)
`POST`	NO	NO	Create / append
`PUT`	NO	YES	Replace / overwrite
`PATCH`	NO	*sometimes*	Partial update
`DELETE`	NO	YES	Remove

- **Safe** = does not change server state. Browsers may prefetch safe URLs.
- **Idempotent** = same end state no matter how many identical requests arrive.

Idempotence protects you from network retries. If a client retries on timeout, an idempotent endpoint will not corrupt data.

Crafting Clean URIs

Bad (verb-based)	Good (resource-based)
`GET /api/getProducts`	`GET /api/v1/products`
`POST /api/createProduct`	`POST /api/v1/products`
`POST /api/update_product?id=4`	`PUT /api/v1/products/4`
`GET /api/deleteUserOrder?u=3&o=12`	`DELETE /api/v1/users/3/orders/12`

Rules of thumb

- URLs are **nouns**. The HTTP method is the verb.
- Collections are **plural**: `/products`, not `/product`.
- Hierarchy expresses ownership: `/users/{id}/orders`.
- **kebab-case** for multi-word names: `/shipping-addresses`.

Filtering, Sorting, Paging, and Versioning

Use query parameters — not new endpoints — for collection control

```
GET /api/v1/books?genre=scifi&author=asimov&sort=-published&page=2&size=20
```

API versioning — the URL prefix is your contract

```
/api/v1/customers      <-- stable forever, never breaks v1 clients
/api/v2/customers      <-- breaking schema change introduced here
```

Do not create a special `/scifi-books` endpoint. Keep paths generic and let filters do the work.

Status Code Blueprint

Code	Meaning	When you return it
200	OK	Successful GET / PUT / PATCH with body
201	Created	Successful POST, with `Location:` header
204	No Content	Successful DELETE, or PUT with no body
400	Bad Request	Malformed JSON, validation failure
401	Unauthorised	Missing / invalid auth credentials
403	Forbidden	Authenticated but not allowed
404	Not Found	Resource ID does not exist
409	Conflict	Duplicate, version mismatch
500	Server Error	Unhandled exception
503	Unavailable	Maintenance, dependency down

Golden rule: *never return `200 OK` for a failed operation.* Monitors, gateways, and clients all rely on the status code to detect errors.

Request Lifecycle in a Spring Boot API

- **Controller** — HTTP only: route, deserialize, status.
- **Service** — business logic, transactions, validation rules.
- **Repository** — database access (Spring Data JPA).

Each layer talks only to the layer immediately below it.

The Key Annotations

```
@RestController                    // = @Controller + @ResponseBody
@RequestMapping("/api/v1/products")
public class ProductController { ... }

@GetMapping                        // GET    /api/v1/products
@GetMapping("/{id}")               // GET    /api/v1/products/42
@PostMapping                       // POST   /api/v1/products
@PutMapping("/{id}")               // PUT    /api/v1/products/42
@DeleteMapping("/{id}")            // DELETE /api/v1/products/42

@PathVariable Long id              // pulls {id} out of the URL
@RequestParam String sort          // ?sort=...
@RequestBody ProductDto dto        // deserialises JSON body
@Valid                             // triggers bean validation
```

- `@RestController` tells Spring to serialise every return value as JSON (via Jackson).
- `@RequestMapping` on the class is the common prefix; method annotations add the path suffix and HTTP verb.

ProductController — the skeleton (1/2)

```
@RestController
@RequestMapping("/api/v1/products")
public class ProductController {

    private final ProductService service;

    // Constructor injection -- clean, immutable, testable
    public ProductController(ProductService service) {
        this.service = service;
    }

    @GetMapping
    public ResponseEntity<List<ProductDto>> list() {
        return ResponseEntity.ok(service.findAll());
    }

    @GetMapping("/{id}")
    public ResponseEntity<ProductDto> getById(@PathVariable Long id) {
        return service.findById(id)
            .map(ResponseEntity::ok)
            .orElseGet(() -> ResponseEntity.notFound().build());
    }
}
```

ProductController — the skeleton (2/2)

```
@PostMapping
public ResponseEntity<ProductDto> create(
        @Valid @RequestBody ProductCreateDto dto) {
    ProductDto saved = service.create(dto);
    URI location = URI.create("/api/v1/products/" + saved.id());
    return ResponseEntity.created(location).body(saved);   // 201
}

@PutMapping("/{id}")
public ResponseEntity<ProductDto> update(
        @PathVariable Long id,
        @Valid @RequestBody ProductUpdateDto dto) {
    return ResponseEntity.ok(service.update(id, dto));      // 200
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> delete(@PathVariable Long id) {
    service.delete(id);
    return ResponseEntity.noContent().build();              // 204
}
```

Notice every handler returns `ResponseEntity` so we control status code and headers.

Entity vs DTO — why we split them

Risk: return the Entity	**Fix: introduce a DTO**
Leaks DB column names	Public shape is yours to design
Exposes audit fields, password hashes	Only the fields you choose go out
Over-posting: clients can set `isAdmin`	Only declared DTO fields are bound
Tight coupling: schema change = API break	Migrate DB without breaking clients

The flow:

```
HTTP JSON  -->  CreateDto  -->  Entity (DB)
HTTP JSON  <--  ResponseDto <--  Entity (DB)
```

Use a mapper (manual, MapStruct, or ModelMapper) between the layers.

DTO + Validation example

```
public record ProductCreateDto(

    @NotBlank(message = "name is required")
    @Size(min = 2, max = 100)
    String name,

    @NotNull @Min(0)
    BigDecimal price,

    @NotBlank @Pattern(regexp = "[A-Z]{3}",
                       message = "currency must be ISO 4217, e.g. USD")
    String currency
) {}
```

In the controller:

```
@PostMapping
public ResponseEntity<ProductDto> create(@Valid @RequestBody ProductCreateDto dto)
```

If validation fails, Spring throws `MethodArgumentNotValidException` **before** *your method runs — we'll catch it next.*

Never leak a stack trace

The problem. A raw stack trace tells an attacker:

- your exact frameworks and versions
- your package structure and class names
- which database driver you use
- sometimes: SQL fragments and table names

The pattern. `@RestControllerAdvice` is a global interceptor. Every exception thrown anywhere in your controllers passes through it before reaching the client.

Your service layer just throws meaningful exceptions. The advice turns them into clean HTTP responses.

GlobalExceptionHandler — the implementation

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ProductNotFoundException.class)
    public ResponseEntity<ProblemDetail> handleNotFound(
            ProductNotFoundException ex, HttpServletRequest req) {
        ProblemDetail body = ProblemDetail
            .forStatusAndDetail(HttpStatus.NOT_FOUND, ex.getMessage());
        body.setTitle("Resource not found");
        body.setInstance(URI.create(req.getRequestURI()));
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(body);
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ProblemDetail> handleValidation(
            MethodArgumentNotValidException ex) {
        ProblemDetail body = ProblemDetail
            .forStatusAndDetail(HttpStatus.BAD_REQUEST,
                ex.getBindingResult().getAllErrors().get(0).getDefaultMessage());
        body.setTitle("Validation failed");
        return ResponseEntity.badRequest().body(body);
    }
}
```

Standard error payload — RFC 7807 Problem Details

```
{
  "type":     "https://api.suza.ac.tz/errors/product-not-found",
  "title":    "Resource not found",
  "status":   404,
  "detail":   "Product with ID 492 does not exist.",
  "instance": "/api/v1/products/492"
}
```

- **type** — stable URI naming this error category.
- **title** — short, human-readable summary.
- **status** — mirrors the HTTP status code.
- **detail** — specific to this occurrence.
- **instance** — the URL where it happened.

Spring 6 ships a `ProblemDetail` class — use it. Front-end teams can write one shared error handler for the whole platform.

Key Takeaways

1. **Resources, not actions.** URLs are nouns; HTTP verbs are the actions.
2. **Status codes are the API.** 200 only if it really succeeded.
3. **Thin controllers.** HTTP plumbing only — business logic lives in services.
4. **DTOs at the edge.** Never expose JPA entities to the outside world.
5. **Validate early.** `@Valid` + bean validation rejects bad input at the perimeter.
6. **Sanitise errors globally.** `@RestControllerAdvice` + `ProblemDetail`, never raw stack traces.
7. **Version your URL.** `/api/v1/...` from day one.

Practice This Today

Lab pointer. Lab 6 (Spring Boot REST API: Students CRUD with validation & Swagger) puts every concept from this lecture into practice.

Quick start your own project.

1. Open `https://start.spring.io`
2. Pick: Maven, Java 21, Spring Boot 3.x
3. Add dependencies: *Spring Web*, *Validation*, *Spring Data JPA*, *H2*
4. Generate, unzip, `./mvnw spring-boot:run`
5. Build the same `ProductController` we walked through today.

Coming next.

- **L10b** — the same API in Node + Express
- **L10c** — the same API in Django REST Framework
- **L11** — securing it: JWT, roles, OWASP Top 10

Questions?

See the course site for cheatsheets and the full Spring Boot reference list:

`https://massoudhamad.github.io/web-technologies/resources/`