UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Backend de Aplicaciones
Apunte de Clases

Apunte de Clases 17 - Proyectos Spring - Elementos transversales

Este documento continúa el trabajo realizado sobre el ejemplo práctico de una API REST de gestión de personas, implementada con Spring Boot, Spring Web y Spring Data JPA. A partir de esta base, se desarrollan los elementos transversales clave para el diseño profesional de aplicaciones backend.

1. División en capas y responsabilidades

Una buena arquitectura en aplicaciones Spring Boot se basa en la **separación clara de responsabilidades**. Esto permite mantener el código modular, testable y fácil de mantener.

Capas principales y sus responsabilidades

📍 Controller

- Se compone de componentes anotados con `@RestController` y `@RequestMapping`.
- Expone endpoints REST utilizando anotaciones como `@GetMapping`, `@PostMapping`, etc.
- Mapea parámetros de entrada desde path, query params o body hacia objetos Java.
- Retorna las respuestas HTTP correctamente estructuradas (status codes, headers, body).
- No debe contener lógica de negocio, sino delegar esa responsabilidad al servicio.
- Puede aplicar transformación entre DTOs y entidades.

Ejemplo de responsabilidad:

> Decidir si una respuesta HTTP será 200 OK, 404 Not Found o 201 Created según el resultado del servicio.

⚙️ Service

- Centraliza la lógica de negocio: validaciones, cálculos, transformaciones y reglas del dominio.
- Orquesta acciones sobre múltiples repositorios si es necesario.
- Incluso puede orquestar conexiones a otros microservicios
- Puede aplicar patrones de diseño como Strategy, State, Chain of Responsibility u otros para modelar reglas complejas.
- Detecta y lanza excepciones semánticas del negocio (por ejemplo: `PersonaDuplicadaException`).

Ejemplo de responsabilidad:

> Antes de crear una persona, verificar si ya existe otra con el mismo DNI. Aplicar una regla de negocio sobre un estado para permitir o rechazar una operación.

🗃️ Repository

- Encapsula la relación con la base de datos o esquema de persistencia de cada entidad o modelo.
- Define la interfaz de acceso a datos usando `JpaRepository`, `CrudRepository` u otras variantes.
- Se encarga de leer y persistir entidades en la base de datos.
- No contiene lógica de negocio, aunque puede contener consultas complejas personalizadas.

Ejemplo de responsabilidad:

> Buscar personas por barrio, por nombre, o entre dos fechas de nacimiento.

📦 Model

- Representa entidades del dominio que se mapean a tablas de la base de datos.
- Incluye anotaciones JPA como `@Entity`, `@Id`, `@ManyToOne`, etc.
- Puede incluir validaciones estructurales (ej: `@NotNull`, `@Size`).

Ejemplo de responsabilidad:

> Definir que el atributo `nombre` de una persona es obligatorio y no puede superar los 60 caracteres.

🔄 DTO (opcional)

- Separa los objetos de entrada/salida del API de las entidades internas del sistema.
- Permite ocultar o transformar campos, enriquecer con datos adicionales, o validar formatos específicos.

Ejemplo de responsabilidad:

> Enviar solo nombre completo, edad y ciudad en la respuesta del endpoint público sin exponer el ID o fecha de nacimiento exacta.

Estructura de paquetes sugerida

```
utnfc.isi.back.spring.personas
├── controllers  --> PersonaController.java
├── services     --> PersonaService.java
├── repositories --> PersonaRepository.java
├── models       --> Persona.java
├── dtos         --> PersonaDTO.java (si aplica)
└── exceptions --> GlobalExceptionHandler.java, ServiceException.java
```

Flujo típico de una petición HTTP

1. El cliente realiza una petición HTTP al controlador.
2. El `@RestController` delega en el servicio la ejecución de la lógica.
3. El servicio consulta o modifica entidades a través del/os repositorio/s.
4. El resultado se transforma en una respuesta (`ResponseEntity`) y se devuelve.

🔁 2. Manejo de respuestas HTTP con `ResponseEntity`

Spring facilita el retorno de respuestas HTTP bien formadas usando `ResponseEntity`, que encapsula:

- Código de estado (`200`, `201`, `204`, `400`, `404`, etc.)
  - Ver HHTP Cats
- Headers personalizados (si es necesario)
- Cuerpo de respuesta (normalmente un objeto serializado como JSON)

Un ejemplo concreto típico al obtener una persona por ID desde el controlador sería:

```
@GetMapping("/{id}")
public ResponseEntity<Persona> obtenerPorId(@PathVariable Long id) {
    Optional<Persona> persona = personaService.buscarPorId(id);
    // Equivalente a: persona.map(p -> ResponseEntity.ok(p))
    return persona
        .map(ResponseEntity::ok)
        .orElseGet(() -> ResponseEntity.notFound().build());
}
```

> Este enfoque aprovecha el uso de `Optional` y devuelve de forma clara:
>
> - `200 OK` con el objeto si existe.
> - `404 Not Found` si no se encontró el recurso.

Ejemplos comunes

```
return ResponseEntity.ok(persona); // 200 OK
return ResponseEntity.status(HttpStatus.CREATED).body(persona); // 201
Created
return ResponseEntity.noContent().build(); // 204 No Content
```

> Se recomienda evitar retornar directamente objetos sin `ResponseEntity`, ya que esto dificulta controlar los status codes y headers de forma explícita.

3. Validaciones y manejo de errores globales

Las validaciones se implementan en el modelo utilizando anotaciones como:

```
@NotBlank(message = "El nombre es obligatorio")
private String nombre;

@Past(message = "La fecha debe ser anterior a hoy")
private LocalDate fechaNacimiento;
```

El controlador puede recibir estas validaciones automáticamente con `@Valid`:

```
@PostMapping
public ResponseEntity<Persona> crear(@Valid @RequestBody Persona persona)
{ ... }
```

Para capturar y estructurar los errores de forma global, se utiliza un `@ControllerAdvice`:

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>>
handleValidationErrors(MethodArgumentNotValidException ex) {
        Map<String, String> errores = new HashMap<>();
        ex.getBindingResult().getFieldErrors().forEach(error ->
            errores.put(error.getField(), error.getDefaultMessage()));
        return ResponseEntity.badRequest().body(errores);
    }

    @ExceptionHandler(ServiceException.class)
    public ResponseEntity<String> handleServiceException(ServiceException
ex) {
        return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body(ex.getMessage());
    }
}
```

Esto permite respuestas JSON coherentes incluso ante errores, mejorando la experiencia de uso de la API y facilitando el debugging.

📁 4. Documentación del API con Swagger / OpenAPI

Para facilitar la exploración y el entendimiento de una API REST, es altamente recomendable documentarla usando **OpenAPI**. En el ecosistema Spring, la forma más simple y moderna de hacerlo es utilizando la biblioteca **springdoc-openapi**.

🧩 ¿Qué es springdoc-openapi?

Es una biblioteca que analiza automáticamente las anotaciones de tus controladores (`@RestController`, `@RequestMapping`, etc.) y genera una especificación OpenAPI 3.0, accesible desde una interfaz web (Swagger UI).

🧠 Dependencia Maven

Agregar al `pom.xml`:

```
<dependency>
  <groupId>org.springdoc</groupId>
```

```
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.3.0</version>
</dependency>
```

🚀 Acceso a Swagger UI

Una vez iniciada la aplicación, se puede acceder a la documentación navegando a:

```
http://localhost:8080/swagger-ui.html
```

Esto abre una interfaz gráfica para:

- Visualizar todos los endpoints expuestos
- Probar peticiones directamente desde el navegador
- Ver descripciones, parámetros, request body y responses

📌 Personalización básica

Podés agregar descripciones, ejemplos y tags usando anotaciones como:

```
@Operation(summary = "Buscar persona por ID", description = "Devuelve una
persona si existe")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200", description = "Persona
encontrada"),
    @ApiResponse(responseCode = "404", description = "No se encontró la
persona")
})
@GetMapping("/{id}")
public ResponseEntity<Persona> obtenerPorId(@PathVariable Long id) { ... }
```

Para usar estas anotaciones, agregá al `pom.xml`:

```
<dependency>
  <groupId>io.swagger.core.v3</groupId>
  <artifactId>swagger-annotations</artifactId>
  <version>2.2.20</version>
</dependency>
```

🧠 Ventajas

- Cero configuración: Swagger se genera automáticamente.
- Mejora la comunicación entre desarrolladores backend y frontend.
- Permite probar la API sin necesidad de Postman o Curl.
- Puede exportarse a JSON/YAML para documentación formal.

🧪 5. Testing unitario e integrado

Se muestra cómo testear cada capa:

- Repositorios con `@DataJpaTest`
- Servicios con Mockito y pruebas de lógica
- Controladores con `@WebMvcTest` y MockMvc

No avanzamos aquí con Test unitarios a niveles de servidor porque no lo vamos requerir en el TPI pero se deja constancia de su existencia para el propio avance y estudio del estudiante.

> En este documento no se incluirán detalles sobre seguridad o logging, ya que serán abordados en unidades posteriores con materiales específicos.