Desarrollo web en entorno servidor

Desarrollo de API

Spring Boot REST API – ResponseEntity – Control de errores

IES Clara del Rey – Madrid

Códigos de respuesta en REST API

Cuando se desarrolla una API REST, se deben intentar seguir una serie de convenciones, relacionadas con la URL, con los métodos HTTP, con los códigos de error que se devuelven, etc., que ya se han mencionado.

En lo que respecta a la respuesta al cliente, hay que usar siempre códigos para informar del resultado de la petición:

- Códigos 2xx para peticiones que se ha resuelto con éxito
- Códigos 4xx para peticiones que no se realizan por error del cliente
- Códigos 5xx para peticiones que no se realizan por error en servidor

¿Cómo especificar estos códigos de la forma más simple posible, sin "engordar" el controlador con código de verificación o validación?

Códigos de respuesta en Spring REST API

En Spring se dispone de la clase "ResponseEntity" que permite personalizar la respuesta HTTP en los controladores de una API REST.

Permite:

- Definir el cuerpo de la respuesta, los datos que se devuelven al cliente.
- Especificar el código de estado HTTP de la respuesta 200 OK, 201 Created, 404 Not Found, etc.).
- Agregar cabeceras HTTP personalizadas.

Dispone de métodos para realizar fácilmente la devolución de códigos.

ResponseEntity

Para usar ResponseEntity hay que:

- Hacer que los métodos del controlador devuelvan ResponseEntity<T>
- T es el tipo de datos que se quiere devolver. Por ejemplo:
  - ResponseEntity<Persona> para devolver una persona.
  - ResponseEntity<List<Persona>> para una lista de personas.
  - ResponseEntity<String> para devolver una cadena de texto.
- Usar alguna de los métodos de ResponseEntity para "envolver" los datos devueltos.
- El método más flexible para devolver datos es ResponseEntity.status(...).body(...)

ResponseEntity

En su uso más "simple" se utiliza el método "status", que recibe un número entero, el código de respuesta HTTP, y se encadena con el método "body", que establece el cuerpo de la respuesta:

```
return ResponseEntity.status(200).body("Esto es el cuerpo");
```

Se recomienda usar el enumerado HttpStatus para no escribir los números directamente, y hacer el código más legible.

```
return ResponseEntity.status(HttpStatus.OK).body("Esto es el cuerpo");
```

Si se quiere devolver una respuesta sin cuerpo se puede llamar al método "build" en lugar de llamar a "body()":

```
return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
```

ResponseEntity – BodyBuilder

El método estático status() devuelve un objeto del tipo "BodyBuilder". Este objeto se puede usar para construir y personalizar la respuesta.

BodyBuilder (como StringBuilder) sigue el patrón "builder", en el que cada método devuelve el mismo objeto, y esto permite encadenar llamadas a métodos para construir la salida. Ejemplo:

```
return ResponseEntity.status(HttpStatus.OK)
        .header("Custom-Header", "Valor de cabecera") // Cabecera personalizada
        .header("Cache-Control", "no-store") // Evita el almacenamiento en caché
        .contentType(MediaType.APPLICATION_JSON) // Define tipo de contenido
        .body(producto); // Establece el cuerpo de la respuesta.
```

ResponseEntity tiene otros métodos de utilidad, "atajos" que permiten realizar operaciones habituales sin encadenar llamadas al builder.

ResponseEntity – BodyBuilder

Algunos métodos de utilidad de BodyBuilder:

Método	Descripción
body(T body)	Establece el cuerpo de la respuesta con el tipo T. Un mensaje, un objeto, una colección, etc.
header(String name, String... values)	Agrega una o más cabeceras a la respuesta.
headers(HttpHeaders headers)	Agrega una o más cabeceras a la respuesta.
contentLength(long contentLength)	Especifica longitud del contenido. Útil en archivos.
contentType(MediaType contentType)	Define el tipo de contenido en la respuesta. application/json, text/plain, image/png, etc.
cacheControl(CacheControl cControl)	Configura políticas de caché. no-cache", etc.
location(URI location)	Agrega un header "Location" para redirecciones. Se puede usar con "201 Created" para indicar dónde se creó el recurso.

ResponseEntity – Atajos

ResponseEntity tiene métodos de utilidad, "atajos" que permiten realizar operaciones habituales de forma más simple.

Algunos ejemplos:

Método atajo	Equivale a
ok(T body)	status(HttpStatus.OK).body(body)
ok()	status(HttpStatus.OK).build()
created(URI location)	status(HttpStatus.CREATED).header("Location", location).build()
badRequest()	status(HttpStatus.BAD_REQUEST).build()
badRequest().body(T body)	status(HttpStatus.BAD_REQUEST).body(body)
notFound()	status(HttpStatus.NOT_FOUND).build()
notFound().body(T body)	status(HttpStatus.NOT_FOUND).body(body)
noContent()	status(HttpStatus.NO_CONTENT).build()

ResponseEntity – Atajos "of" y "ofNullable"

Hay dos atajos, "of" y "ofNullable" que pueden ser especialmente útiles para evitar comprobación de nulos o de Optional<T> vacíos:

- of(Optional<T> body)
  - Para devolver opcionales sin verificar manualmente si existen.
  - Equivale a *body.isPresent() ? ok(body.get()) : notFound().build()*.
  - Devuelve "200 OK" si el Optional tiene valor, si no, "404 Not Found".
- ofNullable(T body)
  - Para devolver objetos sin verificar manualmente si son null.
  - Equivale a *body != null ? ok(body) : notFound().build()*
  - Devuelve "200 OK" si el objeto no es null, si lo es, "404 Not Found".

Control de errores – Recomendaciones

A la hora de desarrollar una API REST con arquitectura n-capas, hay que intentar cumplir, entre otras, un par de "directrices":

- Centralizar la lógica de negocio en la capa de servicios
- Como consecuencia de lo anterior, intentar mantener los controladores lo más "ligeros" posibles, evitando en antipatrón "fat controller"

Al trasladar todo el negocio a los servicios, los errores se producirán fundamentalmente en ellos, porque son la parte de la aplicación que acumula más código.

En concreto, si un proceso debe cumplir precondiciones, habrá que lanzar errores adecuados desde la capa de servicios, y conseguir que se devuelvan los códigos adecuados.

Control de errores – Recomendaciones

Cuando se programa una API REST con Spring se recomienda seguir cierto patrón para el lanzamiento de los errores personalizados:

1. Lanzar las excepciones adecuadas desde el servicio. Si no se cumple la precondición, lanzar la excepción asociada al error.

   Si es necesario, crear excepciones para los errores que no se pueden asociar a una excepción concreta que ya exista en la API de Java o en Spring.
2. Crear una clase para gestionar las excepciones. Se debe anotar con @RestControllerAdvice, y tendrá métodos para gestionar cada tipo de excepción. Los métodos se deben anotar con @ExceptionHandler.

Errores – Ejemplo

Supongamos que en un servicio para la gestión de tareas tenemos:

- Controlador "TaskAssignmentController" que tiene un método para completar una asignación de tarea, que recibe en @PathVariable. Para hacerlo, usa el servicio “TaskAssignmentService”.
- Servicio “TaskAssignmentService”, con un método para completar la asignación de tarea, que recibe en parámetro. El método comprueba:
  - Que existe asignación con el id indicado. Si no, lanzará una excepción EntityNotFoundException con el mensaje adecuado.
  - Que la asignación no se ha completado previamente. Si ya está completada, se lanzará una excepción personalizada, TaskAssignmentAlreadyCompletedException.

Errores – Ejemplo

La API REST debe lanzar los errores adecuados. Se ha decidido que:

- Si no se encuentra la asignación de tarea, lanzará un 404 con mensaje “No existe la asignación de tarea con id <id de la asignación>”.
- Si la tarea ya está completada, lanzará un 409 (Conflict) con el mensaje “La asignación de tarea <id de asignación> ya se había completado”.

Por defecto, si el servicio lanza excepciones, el código devuelto es el 500. No hay asociación previa entre excepciones y códigos de error HTTP.

En este punto hay dos opciones:

- Control “estándar”, con try/catch, para devolver códigos HTTP.
- Usar @RestControllerAdvice y @ExceptionHandler

Errores – Ejemplo – Control “estándar”

El método de controlador podría quedar así:

```
@PutMapping("/complete/{id}")
public ResponseEntity<String>completeTaskAssignment(@PathVariable(“id") Integer id) {
    try {
        taskAssignmentService.CompleteTaskAssignment(id);
        return ResponseEntity.ok("Tarea completada");
    } catch (EntityNotFoundException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                             .body(String.format("No existe la asignación de tarea con id %d", id));
    } catch (TaskAssignmentAlreadyCompletedException e) {
         return ResponseEntity.status(HttpStatus.BAD_REQUEST)
                             .body(String.format("La tarea con id %d ya ha sido completada", id));
    }
}
```

Sin ser una solución terrible, puede aligerarse MUCHO el controlador.

Errores – Ejemplo – @RestControllerAdvice

Basta con crear una clase para la gestión de errores, y anotarla con @RestControllerAdvice. Esta clase tendrá un método para gestionar cada tipo de excepción. En nuestro caso una para EntityNotFoundException, y otro para TaskAssignmentAlreadyCompletedException.

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(EntityNotFoundException.class)
    public ResponseEntity<String> handleEntityNotFound(EntityNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());  }

    @ExceptionHandler(TaskAssignmentAlreadyCompletedException.class)
        public ResponseEntity<String>
                    handleTaskAlreadyCompleted(TaskAssignmentAlreadyCompletedException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());  }
}
```

Errores – Ejemplo – @RestControllerAdvice

De esta forma el controlador quedaría:

```
@PutMapping("/complete/{id}")
public ResponseEntity<String>completeTaskAssignment(@PathVariable(“id") Integer id) {

        taskAssignmentService.CompleteTaskAssignment(id);
        return ResponseEntity.ok("Tarea completada");

}
```

Bastante más simple que la versión anterior. Ventajas de este enfoque:

- Centraliza y facilita la gestión de las excepciones
- Mantiene los controladores ligeros, y mejora la mantenibilidad.
- Separa la lógica de negocio del control de errores

Errores – Ejemplo – @RestControllerAdvice

Por defecto, @RestControllerAdvice aplica a todos los controladores (anotados con @RestController) de la aplicación.

Si no se quiere hacer global, sino que se quiere hacer con ciertos tipos, hay varias opciones para restringir el ámbito:

- Indicar el paquete al que se quiere aplicar.
- Indicar los controladores específicos a los que se quiere aplicar, utilizando la clase de los controladores.
- Indicar las anotaciones a las que se quiere aplicar. Se aplicará a todos los controladores que tengan esas anotaciones.

Errores – Ejemplo – @RestControllerAdvice

Restricción por paquete:

```
@RestControllerAdvice(basePackages = "com.shop.products")
public class ProductExceptionHandler { ... }
```

Restricción por clases específicas:

```
@RestControllerAdvice(assignableTypes =
                                {ProductController.class, OrderController.class})
public class ProductExceptionHandler { ... }
```

Restricción por anotaciones específicas:

```
@RestControllerAdvice(annotations = {RestApi.class})
public class ProductExceptionHandler { ... }
```

Errores – Ejemplo – @RestControllerAdvice

En el caso de anotaciones específicas, puede que haya que crear una anotación. Ejemplo de anotación:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@RestController
public @interface RestApi {}
```

Esta anotación "incluye" @RestController, por lo que el controlador quedaría así:

```
@RestApi
public class WhateverController { ... }
```

@RestControllerAdvice vs @ControllerAdvice

@RestController es igual que @Controller, y añade @ResponseBody.

Del mismo modo, @RestControllerAdvice es igual que @ControllerAdvice, y también añade @ResponseBody, lo que por defecto hace que lo que devuelven sus métodos se devuelva en formato JSON.

En esta tabla se puede ver a qué elementos aplica cada anotación:

Controlador	@ControllerAdvice	@RestControllerAdvice	Formato de respuesta
@Controller	Aplica	No Aplica	HTML / VISTA
@RestController	Aplica	Aplica	JSON