Spring Boot - Exception Handling

Gustavo García, 18 de abril de 2022.

# Fuentes

<https://www.tutorialspoint.com/spring_boot/spring_boot_exception_handling.htm>

<https://www.baeldung.com/exception-handling-for-rest-with-spring>

<https://www.baeldung.com/spring-boot-bean-validation>

<https://devwithus.com/exception-handling-spring-boot-rest-api/>

## Atención

Estas son solo fuentes. He tenido que actualizar unas cuantas cosas que estaban obsoletas. Lo que hay en este documento funciona bien hoy.

# Introducción

Aprenderemos cómo manejar excepciones y errores en Spring Boot.

Antes de Spring 3.2, los dos enfoques principales para manejar excepciones en una aplicación Spring MVC eran HandlerExceptionResolver o la anotación @ExceptionHandler. Ambos tienen algunas desventajas claras. Desde la versión 3.2, contamos con la anotación @ControllerAdvice para abordar las limitaciones de las dos soluciones anteriores y promover un manejo de excepciones unificado en toda la aplicación. Ahora, Spring 5 presenta la clase ResponseStatusException, una forma rápida de manejar errores básicos en nuestras API REST. Todos estos tienen una cosa en común: manejan muy bien la separación de responsabilidades. La aplicación puede lanzar excepciones normalmente para indicar una falla de algún tipo, las cualeas luego se manejarán por separado.

Si bien Spring Boot admite una integración perfecta con validadores personalizados, el estándar de facto para realizar la validación es Hibernate Validator, la implementación de referencia del marco Bean Validation.

# Mejores prácticas para manejar excepciones

Antes de profundizar en cómo manejar las excepciones de la API de Spring Boot, hablaremos un poco sobre las mejores prácticas que se pueden adaptar para manejar los errores.

## Utilice códigos de estado significativos

El primer paso para gestionar los errores correctamente es proporcionar a los clientes información legible por humanos y códigos de estado HTTP adecuados. Esto puede ayudar a los consumidores a encaminar sus respuestas en consecuencia.

**Debemos evitar devolver el mismo código de estado HTTP cada vez. Es un gran error.**

Elija cuidadosamente un código de estado relevante que refleje significativamente la respuesta devuelta.

Use siempre 2xx para respuestas exitosas, 3xx para redirecciones, 4xx y 5xx para respuestas de error.

|  |  |
| --- | --- |
| 2xx (Éxito) | Indica que la solicitud fue aceptada con éxito |
| 3xx (Redireccionamiento) | Informa que el cliente debe realizar más acciones para completar la solicitud |
| 4xx (Error del cliente) | Indica un error en la solicitud del lado del cliente |
| 5xx (Error del servidor) | Indica que el servidor no cumplió con la solicitud |

**En pocas palabras, no debemos exponer los errores 500 a los clientes porque son errores del servidor, no errores del cliente. Lo que sucede en nuestros servidores no es asunto del cliente.**

## Usar excepciones detalladas personalizadas

En general, las excepciones estándar, también llamadas Alien, devuelven rastros de pila que suelen ser difíciles de entender. Entonces, aquí es donde las excepciones definidas por el usuario vienen al rescate.

**Una regla general: siempre es una buena práctica lanzar una excepción que signifique lo que queremos que signifique.**

Envolver una excepción en una clase más específica puede darnos más control sobre cómo queremos tratarla. Proporcionar nuestro mensaje personalizado puede hacer que nuestro mecanismo de manejo de excepciones sea más significativo y fácil de usar.

# Build Rest API con Spring Boot

Spring Boot es una plataforma de elección cuando se trata de crear API RESTful. Entonces, en esta sección, usaremos Spring Boot para desarrollar una aplicación web simple que exponga los servicios web RESTful para la administración de usuarios.

## End point

Un end point (un terminal) es una URL. Esa URL la procesa la clase controller, y decide qué hacer. Observar bien el esquema de construcción que tienen estas URLs, o end points. Esta es la práctica usual para API REST.

Los siguientes, serán los puntos finales de nuestra API:

|  |  |  |
| --- | --- | --- |
| GET | /api/users | get all the users |
| GET | /api/users/{id} | retrieve a single user by ID |
| POST | /api/users | create a new user |
| PUT | /api/users/{id} | update a user's details |
| DELETE | /api/users/{id} | delete a user by ID |
| GET | /api/user?username={username} | get a user by username - this endpoint has no real value for our REST API, it is created just for testing purpose |

## Dependencias

Spring Web

Spring Data JPA

Lombok

MySQL Driver

## pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-

        </plugins>

    </build>

</project>

## application.properties

## MicroService name

spring.application.name=msusers

## Port

server.port=8080

## MySQL DataSource

spring.datasource.url=jdbc:mysql://localhost:3306/usersdb

spring.datasource.username=root

spring.datasource.password=

## JPA

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

# Controller Advice

@ControllerAdvice es una anotación para manejar las excepciones globalmente.

Spring 3.2 brinda soporte para un @ExceptionHandler global con la anotación @ControllerAdvice. Esto habilita un mecanismo que rompe con el modelo MVC anterior y hace uso de ResponseEntity junto con la seguridad y flexibilidad de tipos de @ExceptionHandler:

@ControllerAdvice

**public** **class** **RestResponseEntityExceptionHandler** **extends** **ResponseEntityExceptionHandler** {

@ExceptionHandler(value =

{ IllegalArgumentException.class, IllegalStateException.class })

**protected** ResponseEntity<Object> **handleConflict**(

RuntimeException ex, WebRequest request) {

**String** bodyOfResponse = "This should be application specific";

**return** handleExceptionInternal(ex, bodyOfResponse,

**new** **HttpHeaders**(), HttpStatus.CONFLICT, request);

}

}

La anotación @ControllerAdvice nos permite consolidar nuestros @ExceptionHandlers múltiples y dispersos de antes en un único componente global de manejo de errores.

El mecanismo en sí es extremadamente simple pero también muy flexible:

* Nos da control total sobre el cuerpo de la respuesta, así como sobre el código de estado.
* Proporciona mapeo de varias excepciones al mismo método, para ser manejadas juntas.
* Hace un buen uso de la respuesta RESTful más nueva ResposeEntity.

Una cosa a tener en cuenta aquí es hacer coincidir las excepciones declaradas con @ExceptionHandler con la excepción utilizada como argumento del método. Si estos no coinciden, el compilador no se quejará (no hay motivo para hacerlo) y Spring tampoco se quejará. Sin embargo, cuando la excepción se lanza realmente en tiempo de ejecución, el mecanismo de resolución de excepciones fallará con:

java.lang.IllegalStateException: No suitable resolver **for** argument [0] [type=...] HandlerMethod details: ...

import org.springframework.web.bind.annotation.ControllerAdvice;

# Exception Handler

@ExceptionHandler es una anotación que se usa para manejar excepciones específicas y enviar respuestas personalizadas al cliente.

Puede usar el siguiente código para crear la clase @ControllerAdvice para manejar las excepciones globalmente

# ResponseStatusException (Spring 5 and Above)

Spring 5 introdujo la clase ResponseStatusException. Podemos crear una instancia proporcionando un HttpStatus y, opcionalmente, un motivo y una causa:

@GetMapping(value = "/{id}")

**public** Foo **findById**(@PathVariable("id") Long id, HttpServletResponse response) {

**try** {

**Foo** resourceById = RestPreconditions.checkFound(service.findOne(id)); eventPublisher.publishEvent(**new** **SingleResourceRetrievedEvent**(this, response));

**return** resourceById; }

**catch** (MyResourceNotFoundException exc) {

**throw** **new** **ResponseStatusException**( HttpStatus.NOT\_FOUND, "Foo Not Found", exc);

}

}

¿Cuáles son los beneficios de usar ResponseStatusException?

* Excelente para la creación de prototipos: podemos implementar una solución básica bastante rápido.
* Un tipo, varios códigos de estado: un tipo de excepción puede dar lugar a varias respuestas diferentes. Esto reduce el acoplamiento estrecho en comparación con @ExceptionHandler.
* No tendremos que crear tantas clases de excepción personalizadas.
* Tenemos más control sobre el manejo de excepciones, ya que las excepciones se pueden crear mediante programación.

¿Y qué hay de las contras?

* No existe una forma unificada de manejo de excepciones: es más difícil hacer cumplir algunas convenciones en toda la aplicación en comparación con @ControllerAdvice, que proporciona un enfoque global.
* Duplicación de código: es posible que nos encontremos replicando código en múltiples controladores.

También debemos tener en cuenta que es posible combinar diferentes enfoques dentro de una aplicación.