Spring REST Service Exception Handling

by Amit Phaltankar · Jan. 14, 19 · Java Zone · Tutorial

This tutorial talks about Spring Rest Service Exception Handling. In our previous article, we created our very first Spring Boot REST Web Service. In this tutorial, let's concentrate on how to handle an exception in Spring applications. While there always is an option to handle them manually and set a particular `ResponseStatus`, however, Spring provides an abstraction over the entire exception handling and just asks you to put a few annotations — it takes care of everything else. In this article, we will demonstrate these concepts with code examples.

How to Manually Handle Exceptions

In the Spring Boot Rest Service tutorials, we had created a Dogs Service to understand the concepts. In this post, let's extend the same Dogs Service to handle exceptions.

The `DogsController` returns a `ResponseEntity` instance, which has a response body along with `HttpStatus`.

- If no exception is thrown, the following endpoint returns `List<Dog>` as response body and 200 as status.
- For `DogsNotFoundException`, it returns empty body and status 404.
- For `DogsServiceException`, it returns 500 and empty body.

```
package com.amitph.spring.dogs.web;

import com.amitph.spring.dogs.model.DogDto;
import com.amitph.spring.dogs.repo.Dog;
import com.amitph.spring.dogs.service.DogsService;
import lombok.RequiredArgsConstructor;
import lombok.Setter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
@RequestMapping("/dogs")
@RequiredArgsConstructor
@Setter
```

```
public class DogsController {
    @Autowired private final DogsService service;

    @GetMapping
    public ResponseEntity<List<Dog>> getDogs() {
        List<Dog> dogs;

        try {
            dogs = service.getDogs();
        } catch (DogsServiceException ex) {
            return new ResponseEntity<>(null, null, HttpStatus.INTERNAL_SERVER_ERROR);
        } catch (DogsNotFoundException ex) {
            return new ResponseEntity<>(null, null, HttpStatus.NOT_FOUND);
        }
        return new ResponseEntity<>(dogs, HttpStatus.OK);
    }
}
```

The problem with this approach is `Duplication`. The catch blocks are generic and will be needed in other endpoints as well (e.g. DELETE, POST, etc).

Controller Advice (`@ControllerAdvice`)

Spring provides a better way of handling exceptions, which is Controller Advice. This is a centralized place to handle all the application level exceptions.

Our Dogs Controller now looks clean and is free for any sort of handling exceptions.

Handle and Set Response Status

Below is our `@ControllerAdvice` class where we are handling all the exceptions.

```
package com.amitph.spring.dogs.web;

import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

import static org.springframework.http.HttpStatus.INTERNAL_SERVER_ERROR;
import static org.springframework.http.HttpStatus.NOT_FOUND;

@ControllerAdvice
@Slf4j
public class DogsServiceErrorAdvice {

    @ExceptionHandler({RuntimeException.class})
    public ResponseEntity<String> handleRunTimeException(RuntimeException e) {
        return error(INTERNAL_SERVER_ERROR, e);
    }

    @ExceptionHandler({DogsNotFoundException.class})
```

```
    public ResponseEntity<String> handleNotFoundException(DogsNotFoundException e) {
        return error(NOT_FOUND, e);
    }

    @ExceptionHandler({DogsServiceException.class})
    public ResponseEntity<String> handleDogsServiceException(DogsServiceException e){
        return error(INTERNAL_SERVER_ERROR, e);
    }

    private ResponseEntity<String> error(HttpStatus status, Exception e) {
        log.error("Exception : ", e);
        return ResponseEntity.status(status).body(e.getMessage());
    }
}
```

See what is happening here:

- **handleRunTimeException**: This method handles all the `RuntimeException` and returns the status of *INTERNAL_SERVER_ERROR.*
- **handleNotFoundException:** This method handles `DogsNotFoundException` and returns *NOT_FOUND*.
- **handleDogsServiceException:** This method handles `DogsServiceException` and returns *INTERNAL_SERVER_ERROR.*

The key is to catch the checked exceptions in the application and throw `RuntimeExceptions` . Let these exceptions be thrown out of the `Controller` class, and then, Spring applies the `ControllerAdvice` to it.

```
try {
    //
    // Lines of code
    //
} catch (SQLException sqle) {
    throw new DogsServiceException(sqle.getMessage());
}
```

Use `@ResponseStatus` to Map the Exception to the ResponseStatus

Another short way to do achieve this is to use `@ResponseStatus` . It looks simpler and more readable.

```
package com.amitph.spring.dogs.web;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice
public class DogsServiceErrorAdvice {

    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ExceptionHandler({DogsNotFoundException.class})
    public void handle(DogsNotFoundException e) {}
```

```
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    @ExceptionHandler({DogsServiceException.class, SQLException.class, NullPointerException.class})
    public void handle() {}

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler({DogsServiceValidationException.class})
    public void handle(DogsServiceValidationException e) {}
}
```

Have a look at the second handler. We can group multiple similar exceptions and map a common Response Code for them.

Use @ResponseStatus With a Custom Exception

Spring also does an abstraction for the `@ControllerAdvice`, and we can even skip writing one.
The trick is to define your own `RunTimeException` and annotate it with a specific `@ResponseStatus`. When the particular exception is thrown out of a `Controller`, the Spring abstraction returns the specific Response Status.

Here is a custom `RunTimeException` class.

```
package com.amitph.spring.dogs.service;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class DogsNotFoundException extends RuntimeException {
    public DogsNotFoundException(String message) {
        super(message);
    }
}
```

Let's throw it from anywhere in the code. For example, I am throwing it from a Service method here:

```
public List<Dog> getDogs() {
    throw new DogsNotFoundException("No Dog Found Here..");
}
```

I made a call to the respective Controller endpoint, and I received 404 with the following body.

```
{
    "timestamp": "2018-11-28T05:06:28.460+0000",
    "status": 404,
    "error": "Not Found",
    "message": "No Dog Found Here..",
    "path": "/dogs"
}
```

What is interesting here is my exception message, which is correctly propagated in the response body.

Summary

So, in this Spring Rest Service Exception Handling tutorial, we have seen how to handle an exception with a Spring web application. Spring's exception abstraction frees you from writing those bulky catch blocks and improve the readability of your code with the help of annotations.

Like This Article? Read More From DZone

REST API Error Handling With Spring Boot

Handling Exceptions Using Spring's AOP

Exception Handling in Spring Boot WebFlux Reactive REST Web Services

Free DZone Refcard
Quarkus

Topics: JAVA , SPRING , SPRING BOOT , TUTORIAL , EXCEPTION HANDLING , EXCEPTIONS , RUNTIMEEXCEPTION