Theory: Exception handling

20 minutes 0 / 5 problems solved

§1. Exceptions in web application

Returning errors to a user is one of the regular situations in a web application. If the user sends an incorrect request that cannot be processed or they want to get information about a non-existing object, your web application should give an answer about the situation. There are specified ranges of HTTP status codes for some general errors, like 400 for a bad request or 404 for not found error.

Here you will see different ways to return the error message to a user in a Spring Boot application.

§2. Prepare controller

Imagine the web app returns information about a flight by its number. In a JSON, the result will look like this:

```
{
  "from": "Berlin Teggel",
  "to": "Stuttgart",
  "gate": "D80"
}
```

We won't provide flight date and time to avoid spending time on its representation setting. Let's create a class for this information.

```
public class FlightInfo {

    private String from;
    private String to;
    private String gate;

    //constructor

    //getters and setters



}
```

Now we can create a simple controller with a single handler. Note that keeping the information in `ArrayList` is not a thread-safe way, and we use it here only to save configuration time.

Current topic:

Exception handling

Topic depends on:

Stage 7
Exception handling

Rest controller

Table of contents:

↑ Exception handling

§1. Exceptions in web application

§2. Prepare controller

§3. Add exception handler

§4. Create a global exception handler

§5. Throw an exception at any place

§6. Ordering handlers

§7. Creating custom Exceptions

§8. Conclusion

Feedback & Comments

```
@RestController
public class FlightController {

    private List<FlightInfo> flightInfoList = new ArrayList<>();

    public FlightController() {

flightInfoList.add(new FlightInfo("Delhi Indira Gandhi", "Stuttgart", "D80
"));

flightInfoList.add(new FlightInfo("Tokio Haneda", "Frankfurt", "110"));

flightInfoList.add(new FlightInfo("Kilimanjaro Arusha", "Boston", "15"));


flightInfoList.add(new FlightInfo("Berlin Schönefeld", "Tenerife", "15"));

    }



    @GetMapping("flights/{id}")

    public FlightInfo getFlightInfo (@PathVariable int id) {

        return flightInfoList.get(id);

    }



}
```

Now we are ready to focus on handling exceptions.

§3. Add exception handler

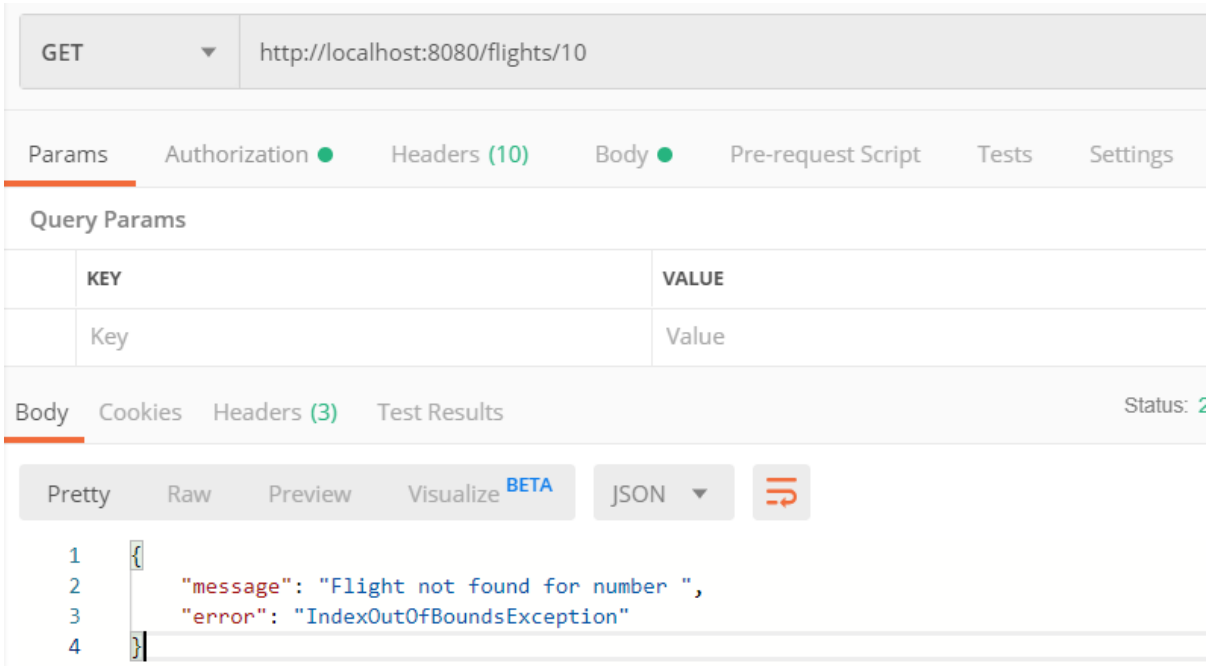
The most obvious exception is when there's no flight for the provided number. For example, let's suppose a user tries to get a flight for the number 1234. If we decide to get the 1234th element of a `flightInfoList`, we will get `IndexOutOfBoundsException`.

Let's create a particular method inside a `FlightController` to handle this exception and return error info to our user:

```

private static final String NOT_FOUND_MESSAGE = "Flight not found for number";

  @ExceptionHandler(IndexOutOfBoundsException.class)
  @ResponseStatus(HttpStatus.NOT_FOUND, reason = NOT_FOUND_MESSAGE)

public HashMap<String, String> handleIndexOutOfBoundsException(Exception e) {
      HashMap<String, String> response = new HashMap<>();
      response.put("message", NOT_FOUND_MESSAGE);
      response.put("error", e.getClass().getSimpleName());
      return response;

  }
```

We added a `handleIndexOutOfBoundsException` method annotated with `@ExceptionHandler`. As you can see, we specify this method for `IndexOutOfBoundsException` handling only, but you can add other exception types using commas. You can even create a handler for all the `RuntimeException` exceptions. The `@ResponseStatus` here sets the code and status to the response.

Let's now test the handler:



@ExceptionHandler test

There are the error name and a specified message in a returned JSON.

§4. Create a global exception handler

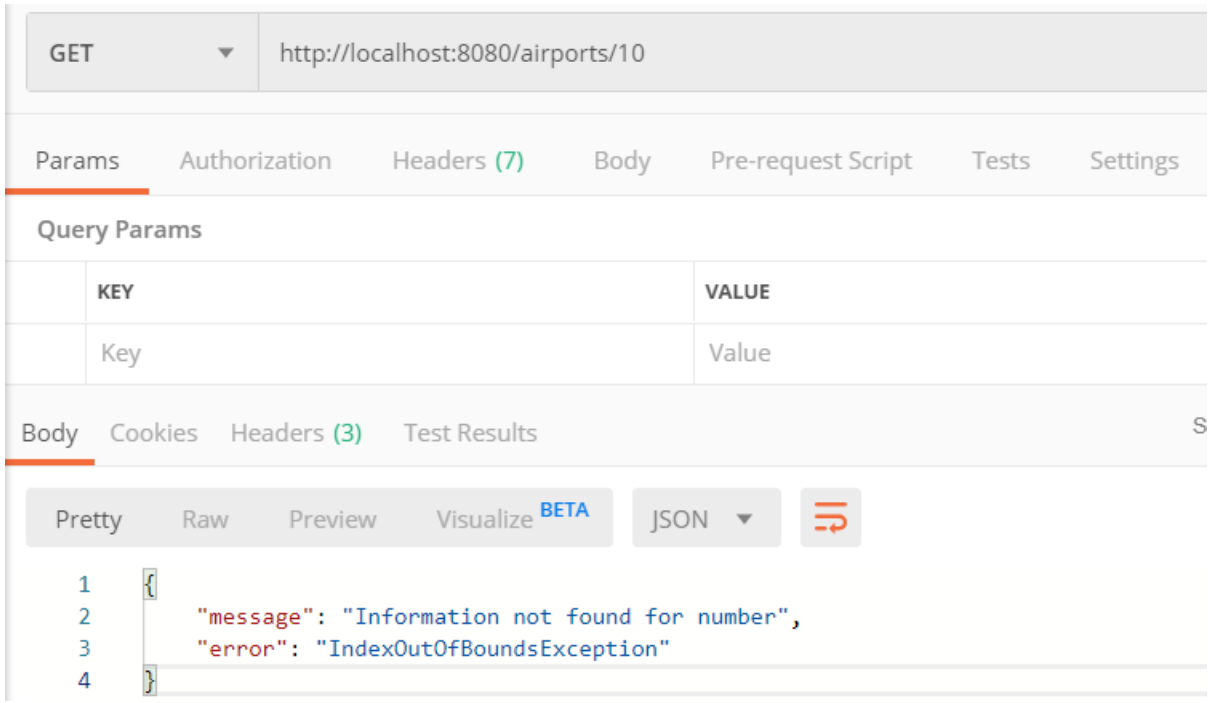
Imagine the web app can also return information about the airports. We will create another controller for this case:

```
@RestController
public class AirportController {

    private List<AirportInfo> airportInfoList = new ArrayList<>();

    public AirportController() {

airportInfoList.add(new AirportInfo("Berlin Teggel", "TXL", "EDDT"));

airportInfoList.add(new AirportInfo("Berlin Schönefeld", "SXF", "EDDB"));
    }



    @GetMapping("airports/{id}")

    public AirportInfo getAirportInfo(@PathVariable int id) {

        return airportInfoList.get(id);

    }



}
```

We can add an `IndexOutOfBoundsException` handler here too. However, to avoid duplicating, it is better to create a single global class for handlers. We need to annotate this class with `@RestControllerAdvice` like this:

```
@RestControllerAdvice
public class ControllerExceptionHandler {


 private static final String NOT_FOUND_MESSAGE = "Information not found for num
ber";

    @ExceptionHandler(IndexOutOfBoundsException.class)

 public HashMap<String, String> handleIndexOutOfBoundsException(Exception e) {
          HashMap<String, String> response = new HashMap<>();
          response.put("message", NOT_FOUND_MESSAGE);

          response.put("error", e.getClass().getSimpleName());

          return response;

    }

}
```

There's no need to keep handlers in controllers because the exception will be handled in a `@RestControllerAdvice` annotated class. We can test it with airports now and see the same returned JSON as it was with flights:



@RestControllerAdvice test

We can also enrich this @ExceptionHandler with the status code by putting `@ResponseStatus` on it:

```
private static final String NOT_FOUND_MESSAGE =
    "Information not found for number";

@ExceptionHandler(IndexOutOfBoundsException.class)
@ResponseStatus(HttpStatus.NOT_FOUND, reason = NOT_FOUND_MESSAGE)

public HashMap<String, String> handleIndexOutOfBoundsException(Exception e)
```

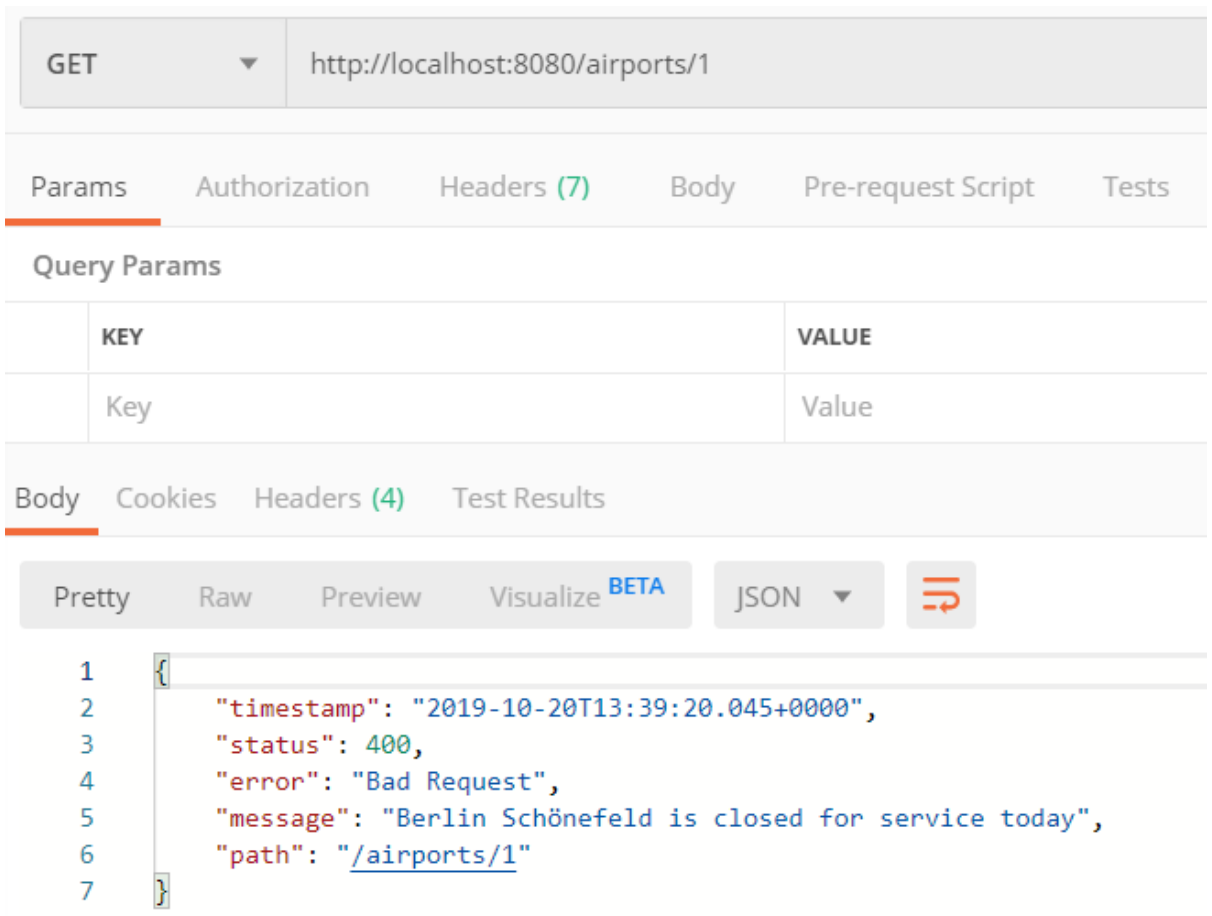
§5. Throw an exception at any place

Sometimes you need to handle the same exception differently in various places. Global handling won't be beneficial here as well as handling in each controller separately. Spring Boot allows handling exception exactly in a place of its appearance. Let's consider another one in the `AirportController`. Imagine a user is looking for some information about the Berlin Schönefeld airport, working hours, for example. If it is closed for service, we will manually return him a warning message.

The modified `AirportController` will look like this:

```
@RestController
public class AirportController {

    private List<AirportInfo> airportInfoList = new ArrayList<>();

 private static final String SERVICE_WARNING_MESSAGE = "Berlin Schönefeld is cl
osed for service today";

    public AirportController() {

    airportInfoList.add(new AirportInfo("Kilimanjaro Arusha", "ARK", "HTAR"));

    airportInfoList.add(new AirportInfo("Berlin Schönefeld", "TXL", "EDDT"));


    airportInfoList.add(new AirportInfo("Delhi Indira Gandhi", "DEL", "VIDP"))
;


    airportInfoList.add(new AirportInfo("Tokio Haneda", "HND", "RJTT"));

    }



    @GetMapping("airports/{id}")

    public AirportInfo getAirportInfo(@PathVariable int id) {

        AirportInfo airportInfo = airportInfoList.get(id);

        if (Objects.equals(airportInfo.getName(), "Berlin Schönefeld")) {


        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, SERVICE_WARN
ING_MESSAGE);

        }

        return airportInfo;

    }



}
```

We use `ResponseStatusException` here. You can specify the HTTP status to return and the message. If you try to test it you will see the standard error info format as a response:



ResponseStatusException test

This JSON provides more information about the situation than just a specified message: timestamp, error name, status code, and the rest path of the request.

§6. Ordering handlers

What can happen if an exception conforms to different handlers? Which one will deal with this exception?

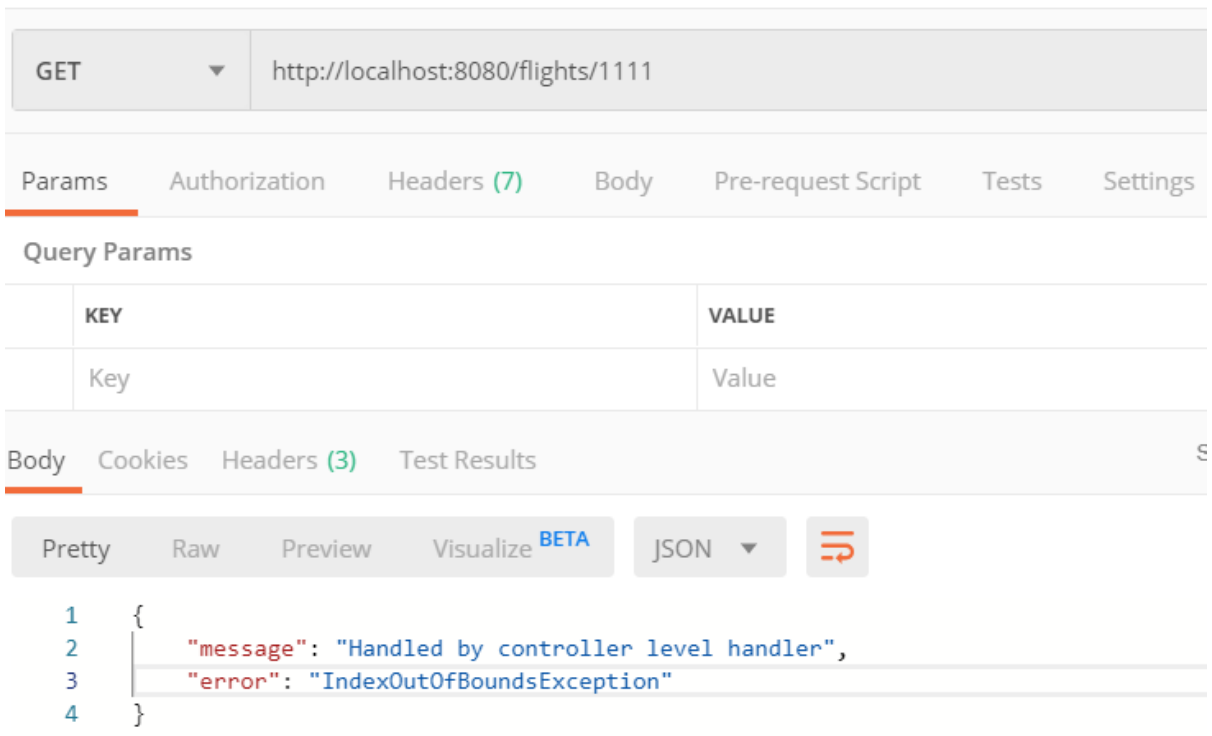
We can manually set the order of handlers with the `@Order` annotation. For example, let's get back the `@ExceptionHandler` for `IndexOutOfBoundsException` to a controller.

```
@Order(Ordered.HIGHEST_PRECEDENCE)
@ExceptionHandler(IndexOutOfBoundsException.class)
public HashMap<String, String> handleNotFoundFlight(Exception e) {
   HashMap<String, String> response = new HashMap<>();
   response.put("message", "Handled by controller level handler");
   response.put("error", e.getClass().getSimpleName());
   return response;
}
```

And still, let's keep the global `@RestControllerAdvice` annotated class for the same type of exception.

```
@RestControllerAdvice
public class ControllerExceptionHandler {

    @ExceptionHandler(IndexOutOfBoundsException.class)
    public HashMap<String, String> handleNotFoundFlight(Exception e) {
        HashMap<String, String> response = new HashMap<>();
        response.put("message", "Handled by global handler");
        response.put("error", e.getClass().getSimpleName());
        return response;

    }

}
```

We set the highest priority to the controller-level handler. Let's test it:



@Order test

As the message in a response says, it handled the exception with the most top priority.

§7. Creating custom Exceptions

There is one more way to set the response code and status to the exception. You can add the annotation `@ResponseStatus` to the custom exception like this :

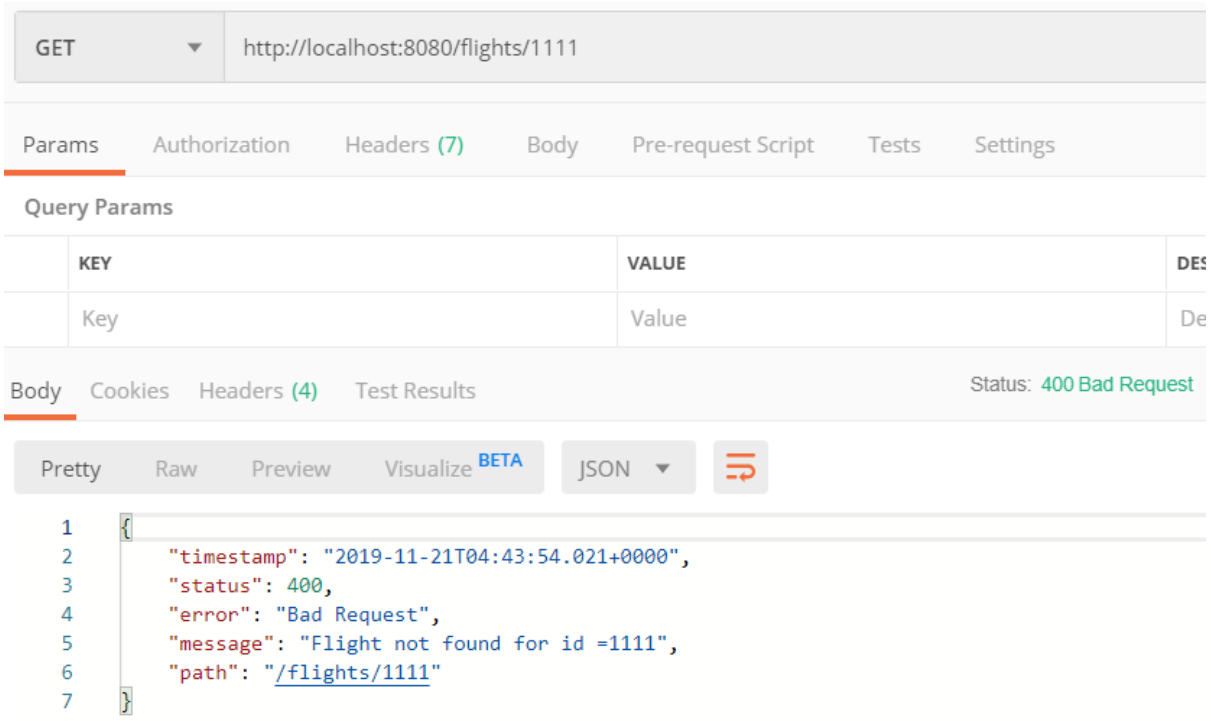
```
@ResponseStatus(code = HttpStatus.BAD_REQUEST)
class FlightNotFoundException extends RuntimeException {}
```

Now you can throw this exception like the `ResponseStatusException` one, and the status will be set automatically.

For example, in a flight controller:

```
@GetMapping("flights/{id}")
public FlightInfo getFlightInfo(@PathVariable int id) {
    if (id > flightInfoList.size()) {

throw new FlightNotFoundException("Flight not found for id =" + id);
    }
    return flightInfoList.get(id);
}
```

If we test this exception with id=1111, we will get a response with the new status code 400.



Custom exception test

§8. Conclusion

Here you have learned different ways of handling exceptions. You can make it:

1. globally with `@RestControllerAdvice`;
2. on controller level with `@ExceptionHandler`;
3. in the most flexible way by throwing `ResponseStatusException` at any place you want.

You can also set the order of handlers with the `@Order` annotation to control the system of them.

Report a typo

84 users liked this theory. **6** didn't like it. **What about you?**

Start practicing

Comments (19) Hints (1) Useful links (0) Show discussion