ERRORS IN SPRING BOOT

AGENDA

- Concept Study
- Custom Errors
- Response Exception Mapper
- Exception Mapper
- Conclusion

Concept Study

Building REST APIs with Spring became the standard approach for Java developers. Using Spring Boot helps substantially, as it removes a lot of boilerplate code and enables auto-configuration of various components.

The ability to handle errors correctly in APIs while providing meaningful error messages is a desirable feature, as it can help the API client respond to issues. The default behavior returns stack traces that are hard to understand and ultimately useless for the API client. Partitioning the error information into fields enables the API client to parse it and provide better error messages to the user.

In this presentation, we cover Spring Boot exception handling for REST API

Custom Errors

REST applications developed in Spring Boot automatically take advantage of its default error handling logic. Specifically, whenever an error occurs, a default response containing some information is returned. The problem is that this information may be poor or insufficient for the API callers to deal with the error properly. This is why implementing custom error handling logic is such a common and desirable task. Achieving it requires more effort.

For this presentation, we will first talk about Default Error Handling in Spring Boot, and then about Custom Error Handling in Spring Boot using the @ControllerAdvice annotation while implementing an example.

Custom Errors

❑ Default Error Handling in Spring Boot

Spring Boot offers an error-handling response in case of REST requests. Spring Boot looks for a mapping for the /error endpoint during the start-up. When no valid mappings can be found, Spring Boot automatically configures a default fallback error page.
Similarly, when dealing with REST requests, Spring Boot automatically returns a default JSON response in case of errors. It looks as follow

```
{
  "timestamp": "2021-15-08T14:32:17.947+0000",
  "status": 500,
  "error": "Internal Server Error",
  "path": "/test"
}
```

Custom Errors

- Default Error Handling in Spring Boot

As we see before, the default Spring Boot error handling responses for REST does not provide much information. This can quickly become a problem, especially when trying to debug. It is also problematic for front-end developers, who need detailed information coming from API error response messages to be able to explain to the end users what happened properly.

Next, we will see how to replace this default response with custom-defined messages. While this may appear like an easy task, this is actually a tricky one. To achieve it, we first need to create a CRUD and handling errors into it.

Custom Errors

❑ Custom Error Handling in Spring Boot

There are two different approaches to custom error handling in Spring Boot REST applications. Both are based on a @ControllerAdvice annotated class handling all exceptions that may occur. So, let's first see what a @ControllerAdvice annotated class is, why to use it, how, and when. Then, we will talk about the two different approaches in detail. And then implement the most suitable.

❑ Handling Exceptions with @ControllerAdvice

The @ControllerAdvice annotation was introduced in Spring 3.2 to make exception handling logic easier and entirely definable in one place. It allows you to address exception handling across the whole application. Classes annotated with @ControllerAdvice are powerful and flexible tools. Not only do they allow you to centralize exception-handling logic into a global component, but also give you control over the body response, as well as the HTTP status code.

Custom Errors

❑ Defining Many Custom Exceptions

This approach involves having as many methods in your @ControllerAdvice as many HTTP error status codes you want to handle. These methods will be related to one or more exceptions and return an error message with a particular HTTP status code.

❑ Defining a Single Custom Exception Carrying All Data

This approach involves defining a custom exception carrying the HTTP status to use, and all the data required to describe the error that occurred. The idea is to turn every exception you want to handle, or you would like to throw under special circumstances, into an instance of this particular exception.

Custom Errors

❑ Pros and Cons of Each Approach

The first approach should be used when you do not want to spread error handling logic all over your codebase. In fact, the HTTP status code is only associated with errors in your @ControllerAdvice annotated class. It respects the principle of least privilege, it does involve boilerplate code. But, you may easily end up with dozens of custom exceptions.

The second approach is a less restricting approach. It is scalable and quicker to be implemented, it allows you to achieve the desired result with little effort. it is more maintainable than the first approach because it involves only a custom exception. Unfortunately, this one is definitely dirtier. It requires you to spread detail about error handling logic in many different points of your code.

For the reasons mentioned above, we have decided to implement our solution with the first approach.

Custom Errors

❑ Building our App

We have decided to implement an user registration CRUD with just the Create and Read functionalities. We will proceed as follow

- **Create an entity class User with name, email, etc. as attributes**
- **Create a dto UserRequest**
- **Create an Interface UserRepository**
- **Create a class UserService for our business logic**
- **Create the controller UserController**

We can now, handling exception in our App.

Custom Errors

❑ Implementation of the First Approach

We have decided to implement the first approach we have seen above. We will proceed as follow

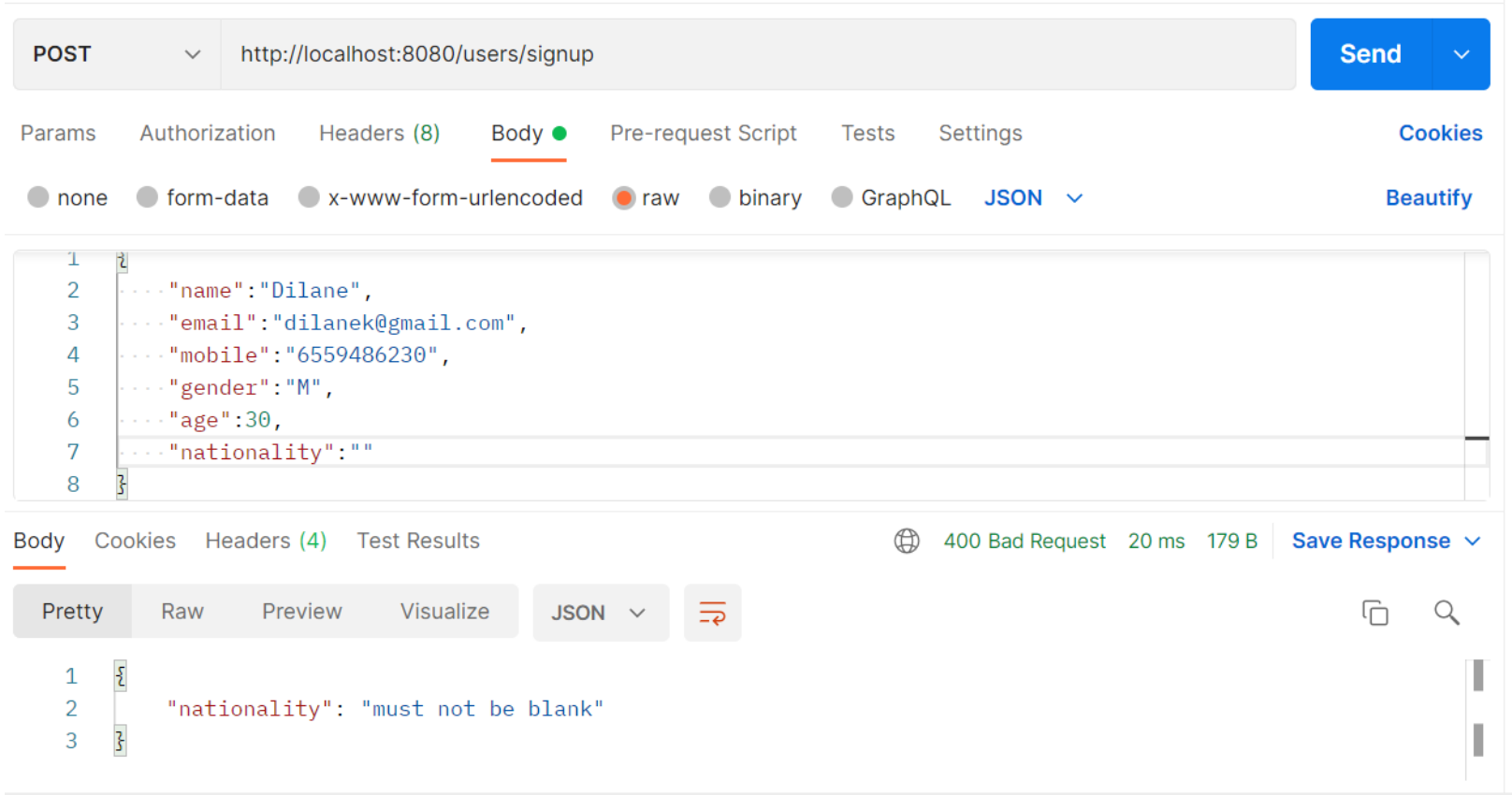
- **Create a class ApplicationExceptionHandler with the annotation @ControllerAdvice. This class will contain 2 methods : handleInvalidArgument for invalid argument exception and handleBusinessException for user not found exception.**
- **Create a class UserNotFoundException. It will help us to customize specific a message for a specific user id.**

Now, let's test our application using Postman

Custom Errors

❑ Bad Request Error or Invalid Argument Exception

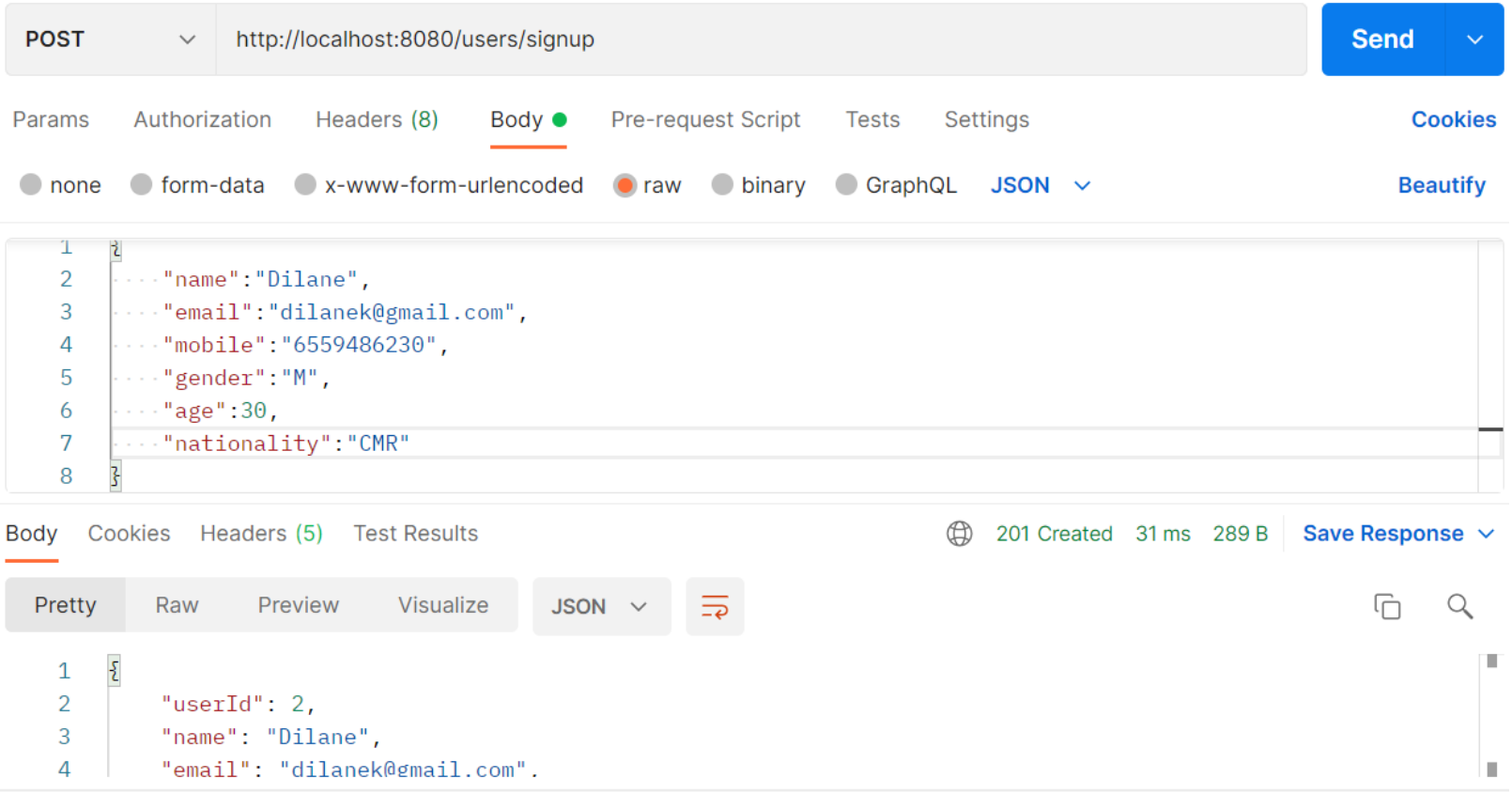
We have an error, because we missed the nationality field



Custom Errors

❑ Bad Request Error or Invalid Argument Exception

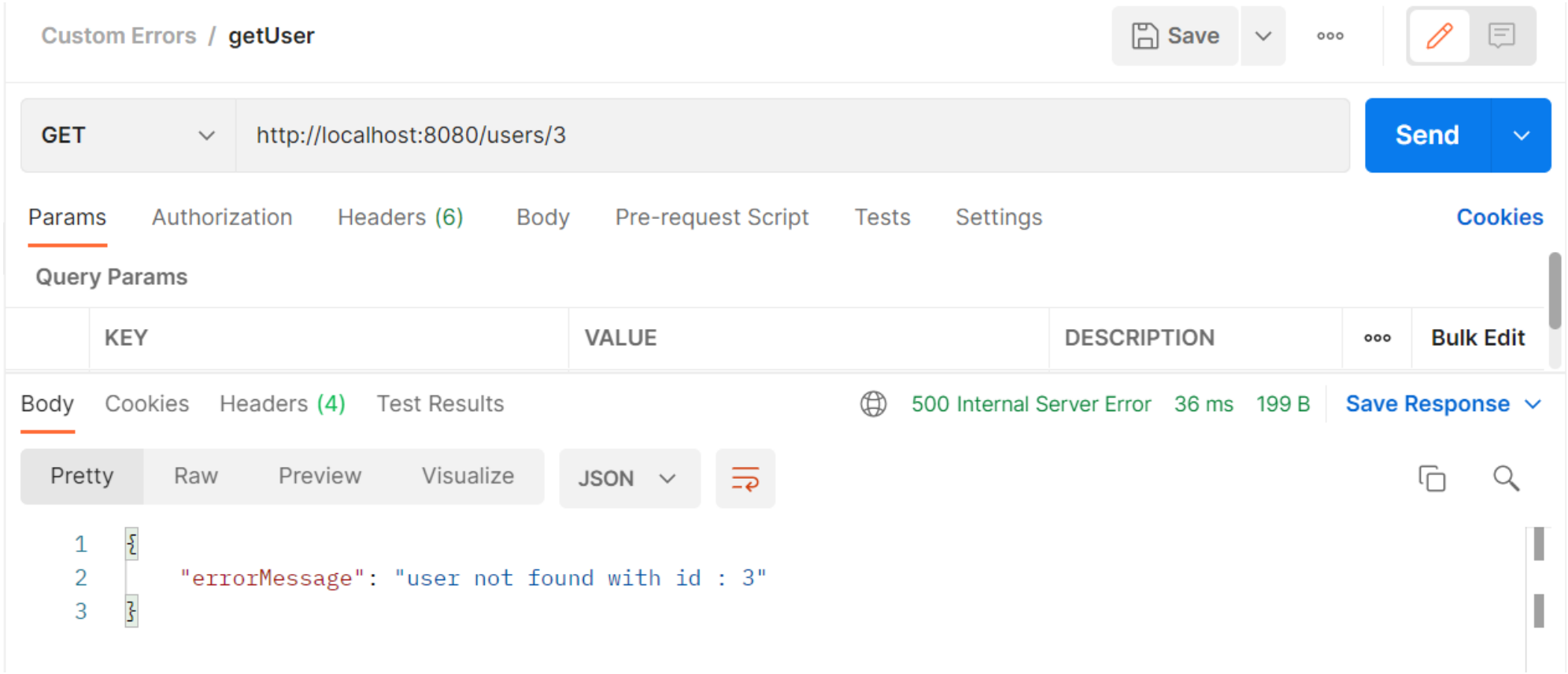
We don’t have an Error anymore, because we have filled the nationality field



Custom Errors

❑ Internal Server Error or User Not Found Exception

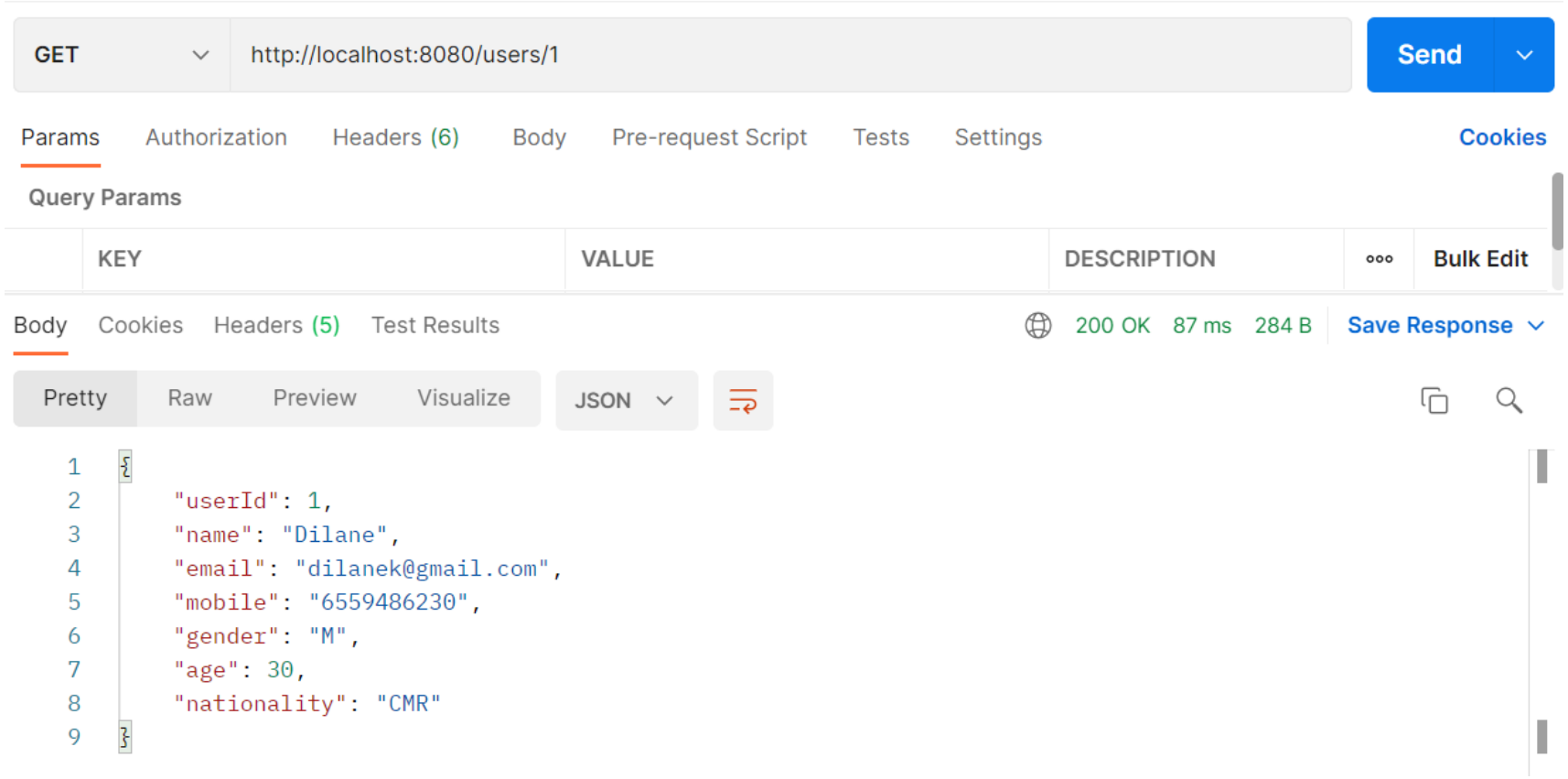
We have an Error, because the user with the id 3 doesn't exist



Custom Errors

❑ Internal Server Error or User Not Found Exception

We don’t have an Error anymore, because the user with the id 1 exist



Response Exception Mapper

For this , we will implement a scenario where our system gets data by calling an external API. We will handling exceptions on the response provide by this public API.

We will do it using the @RestControllerAdvice annotation. First, let's talk a little bit about the @RestControllerAdvice.

- @RestControllerAdvice

The @RestControllerAdvice annotation is specialization of @Component annotation. We will do it using the @RestControllerAdvice annotation.

Rest Controller Advice's methods (annotated with @ExceptionHandler) are shared globally across multiple @Controller components to capture exceptions and translate them to HTTP responses. The @ExceptionHandler annotation indicates which type of Exception we want to handle. The exception instance and the request will be injected via method arguments.

@RestControllerAdvice is the combination of both @ControllerAdvice and @ResponseBody

Response Exception Mapper

❑ Building our System

We have decided to implement a simple system based on 2 services (the todo service and the dashboard service). Our system works as follow: gets data from a public API available on this url https://jsonplaceholder.typicode.com/todos

- **The todo service gets data (tasks) from a public API available on this url https://jsonplaceholder.typicode.com/todos by sending an Http request, it reserves an Http response in JSON format. Then it saves it to a database. It can gets one task or all tasks. If everything is okay, It sends these data to the dashboard service.**

- **The dashboard service gets one task or all tasks from the todo service, save into its own data base. So the user of this service can display these data, without where it comes from.**

We can now, handling exception in our System.

Response Exception Mapper

❑ Handling Exceptions in our system

Let's see how we have implemented the todo service

- **Create a Todo record class model with id, title, and so on.**
- **Create a class JsonPlaceHolder. It will help us to communicate with the public api using Rest Template.**
- **Create an Interface TodoRepository**
- **Create a class TodoService for our business logic**
- **Create the controller TodoController**

Let's see how we have implemented the dashboard service

We can now, handling exception in our App.

Response Exception Mapper

❑ Handling Exceptions in our system

Let's see how we have implemented the todo service

- **Create a Todo record class model with id, title, and so on.**
- **Create a class JsonPlaceHolder. It will help us to communicate with the public api using Rest Template.**
- **Create an Interface TodoRepository**
- **Create a class TodoService for our business logic**
- **Create the controller TodoController**

Let's see how we have implemented the dashboard service

- **Create a Todo record class model with id, title, and so on.**
- **Create an Interface TodoRepository**
- **Create the controller DashboardController**

We can now, handling exception in our App.

Response Exception Mapper

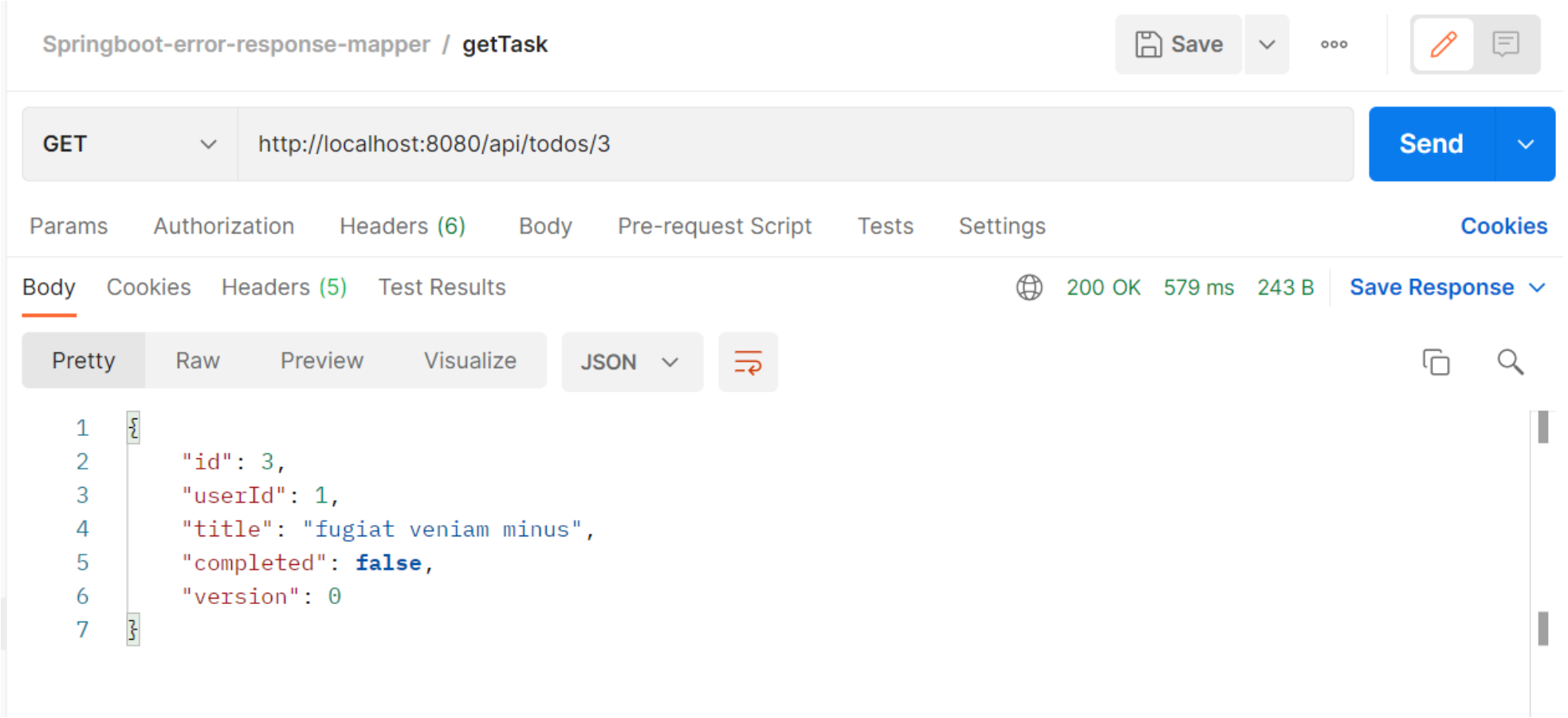
- Internal Server Error or User Not Found Exception

We have an error, because we missed the nationality field

Response Exception Mapper

❑ Internal Server Error or User Not Found Exception

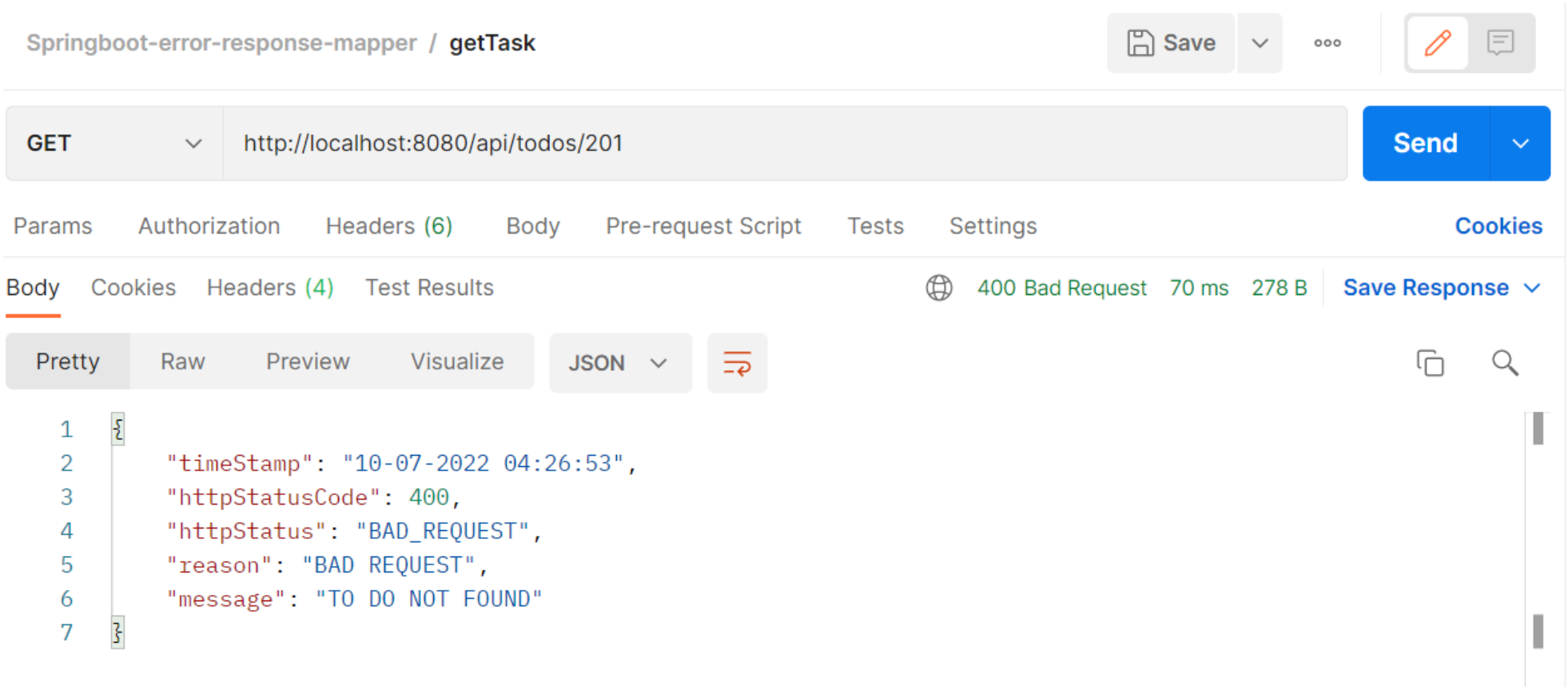
We don’t have an error, because task with id 3 exist in our public api



Response Exception Mapper

❑ Internal Server Error or User Not Found Exception

We have an error, because task with id 201 is not exist in our public api



Conclusion

We have learned how to create custom errors in Spring Boot as well as how to handle errors and responses. The links of our resources and GitHub repository are attached below

https://www.toptal.com/java/spring-boot-rest-api-error-handling

https://auth0.com/blog/get-started-with-custom-error-handling-in-spring-boot-java/

https://www.bezkoder.com/spring-boot-restcontrolleradvice/

https://www.youtube.com/watch?v=gPnd-hzM_6A&t=76s

https://www.youtube.com/watch?v=XEtPVm_SL2Q

https://github.com/Dilane-Kamga/ERRORS-IN-SPRING-BOOT.git

MERCI

Pour votre attention