

CUSTOM VALIDATION IN SPRING BOOT

AGENDA

- ❑ Concept Study
- ❑ Custom Constraint Annotation
- ❑ Validation Error Message
- ❑ Custom Validation Implementation
- ❑ Conclusion

Concept Study

It is necessary to validate user input in any web application to ensure the processing of valid data.

Generally, when we need to validate user input, Spring MVC offers standard predefined validators. However, when we need to validate a more particular type of input, we have the ability to create our own custom validation logic.

In this presentation, we will cover Custom Validation in Spring boot by implementing an CRUD for Users as well as a Custom Validation on the gender field.

Custom Constraint Annotation

Defining a custom constraint validation mainly includes creating a custom validation interface for the annotation and the validator implementation class, which actually implements the validation.

The annotation type is defined using the @interface keyword. The name of the annotation interface defines the annotation itself, which we can use for our validation (e.g., in a field or a bean or DTO, or in a request query parameter). In the annotation interface, we can also define some other parameters such as the target, the retention policy, the real action — validation class with the @Constraint annotation, and so on.

Validation Error Message

The Validation Error Message is defined into the Custom Constraint Annotation Class. The message() is the error message that is showed in the user interface. For example, the message can be “The Integer value is invalid”. We can also provide a custom message for the annotation fields that can be applied the same way as the built-in constraint validations. It is follow by the mostly boilerplate code to conform to the Spring standards.

Custom Validation Implementation

❑ Introduction to the Project

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, gender and so on. as attributes**
- **Create a dto UserRequestException**
- **Create an exception UserNotFound for custom exception handling**
- **Create an Interface UserRepository**
- **Create a class UserService for our business logic**
- **Create the controller UserController**
- **Create an exception advice ApplicationExceptionHandler for custom exception handling**

We can now, implement custom validation into this project.

Custom Validation Implementation

❑ Custom Validation Implementation

We have decided to implement the custom validation for the gender field. Gender should be either Male or Female. We will proceed as follow

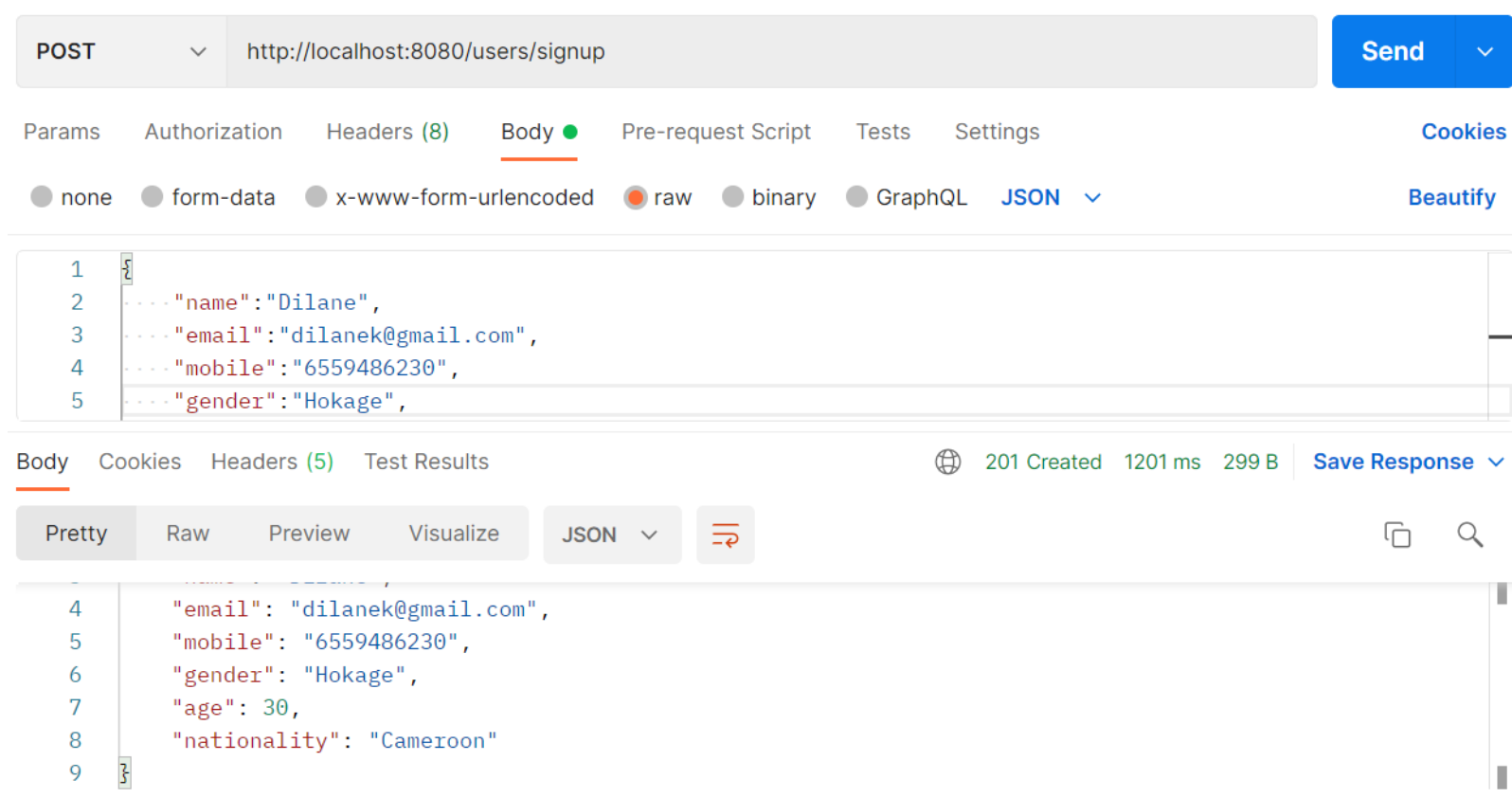
- **Create a package validation.** This will contain our Validator as well as our New Annotation.
- **Create a new @interface called ValidateGender** to define our annotation. This contains the default message "Invalid Gender: It should be either Male or Female".
- **Create a validator class called ValidatorGender** that enforces the rules of our validation i.e test if the gender is either Male or Female.
- **Finally, add the @ValidateGender annotation above the gender field into the UserRequest class into the dto package.**

Now, let's test our application using Postman before and after our custom validation

Custom Validation Implementation

❑ Test Before the Custom Validation Implementation

We don't get an error, but Hokage is not a gender



The screenshot displays a REST client interface with a POST request to `http://localhost:8080/users/signup`. The request body is a JSON object with the following fields: `name` (Dilane), `email` (dilanek@gmail.com), `mobile` (6559486230), and `gender` (Hokage). The response is a 201 Created status with a 1201 ms response time and 299 B of data. The response body is a JSON object with the following fields: `email` (dilanek@gmail.com), `mobile` (6559486230), `gender` (Hokage), `age` (30), and `nationality` (Cameroon).

```
POST http://localhost:8080/users/signup

{
  "name": "Dilane",
  "email": "dilanek@gmail.com",
  "mobile": "6559486230",
  "gender": "Hokage",
}
```

Body Cookies Headers (5) Test Results 201 Created 1201 ms 299 B Save Response

```

  "email": "dilanek@gmail.com",
  "mobile": "6559486230",
  "gender": "Hokage",
  "age": 30,
  "nationality": "Cameroon"
}
```


Custom Validation Implementation

❑ Test After the Custom Validation Implementation

We get an error, because the gender is neither male nor female

The screenshot displays a REST client interface for a POST request to `http://localhost:8080/users/signup`. The request body is a JSON object with the following fields: `name` (Dilane), `email` (dilanek@gmail.com), `mobile` (6559486230), and `gender` (Hokage). The response is a 400 Bad Request with a status of 575 ms and 207 B. The response body is a JSON object with the message: `"gender": "Invalid Gender: It should be either Male or Female"`.

```
POST http://localhost:8080/users/signup
```

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded **raw** binary GraphQL JSON Beautify

```
1 {
2   "name": "Dilane",
3   "email": "dilanek@gmail.com",
4   "mobile": "6559486230",
5   "gender": "Hokage",
6 }
```

Body Cookies Headers (4) Test Results 400 Bad Request 575 ms 207 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "gender": "Invalid Gender: It should be either Male or Female"
3 }
```

Custom Validation Implementation

❑ Test Before the Custom Validation Implementation

We don't get an error, because the gender is male

The screenshot displays a REST client interface for a POST request to `http://localhost:8080/users/signup`. The request body is a JSON object with the following fields: `name` (Dilane), `email` (dilanek@gmail.com), `mobile` (6559486230), and `gender` (Male). The response status is 201 Created, with a response time of 776 ms and a body size of 297 B. The response body is shown in a pretty-printed JSON format, indicating a successful user creation with `userId` 1.

```
POST http://localhost:8080/users/signup
```

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded **raw** binary GraphQL JSON Beautify

```
1 {
2   ... "name": "Dilane",
3   ... "email": "dilanek@gmail.com",
4   ... "mobile": "6559486230",
5   ... "gender": "Male",
6 }
```

Body **Cookies** Headers (5) Test Results 201 Created 776 ms 297 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "userId": 1,
3   "name": "Dilane",
4   "email": "dilanek@gmail.com",
5   "mobile": "6559486230",
6   "gender": "Male",
7 }
```

Conclusion

We have learned how to create custom validators to verify a field or class, and then wire them into Spring MVC. The links of our resources and GitHub repository are attached below

<https://youtu.be/P5sAaFY3O2w>

<https://www.baeldung.com/spring-mvc-custom-validator>

<https://betterprogramming.pub/custom-validation-in-spring-boot-best-explained-part-1-1105a8c2711>

<https://github.com/Dilane-Kamga/Custom-Validation.git>

MERCI
Pour votre attention