Validation and Exception Handling in Spring Boot

Let’s validate incoming requests in our APIs

Text

Description automatically generated

Validation plays a very important role when we are building REST APIs or Micro services. This way we can inform the user what data is needed to be sent and how it should be…If the expected format is not found, we can send an error message to the user.

Not only that, we have to manage the **error prone scenarios** also without making an ambiguity for the user. If any breaking situation occurs **like an exception**, we should be able inform something to user to understand what is wrong!

To accomplish both these tasks, Spring Boot provides a very nice and cleaner way. I will show you how to manage this.

**Request Validation 💥**

Usually we send requests to the back-end to perform various things. One case case be considered as **creating a resource**. For that we use POST methods. So, we send some **input data** which is used inside service layer to create that resource. This is the critical part. Let’s say front-end sends unexpected input data which **violates the DTO defined in the back-end**…Then what happens? Program will break with an exception…But the real thing we need to do is **inform the user with the error**. Then user can correct it…Right? Let’s learn how to deal with this situation.

Spring Boot has a validation library to support this. We should add the maven dependency in the POM first.

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-validation</artifactId>  
</dependency>

I’m going to create a REST API endpoint to **create and save a user**in a database. This user has id, name, age, email, phone number properties.

*I’m going to use MySQL database with JPA and Hibernate. I won’t explain that to keep the article short. And for simplification of Pojos, I will use Lombok also.*

**Entity Layer: User**

@Getter  
@Setter  
@ToString  
@AllArgsConstructor  
@NoArgsConstructor  
@Builder  
@Entity(name = "users")  
public class User {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 Long id;  
 String name;  
 String email;  
 String mobile;  
 int age;  
}

Now we are moving to the controller layer. Usually we annotate the request data with **“@RequestBody”** annotation in Spring Boot. And we create a request DTO also. This is the starting point of our validation process. The validation library we have added is providing an annotation called **“@Valid”** to validate the things. We have to put it in front of RequestBody annotation.

**Controller Layer: UserController**

@RestController  
@RequestMapping(value = "/api")  
public class UserController {  
  
 @Autowired  
 private UserService userService;  
  
 @PostMapping(path = "/users")  
 public ResponseEntity<User> saveUser(**@RequestBody @Valid UserRequestDTO userRequest**) {  
 return new ResponseEntity<>(userService.saveUser(userRequest), HttpStatus.CREATED);  
 }  
}

So, now controller is ready to validate incoming request. But we have to decorate our DTO: **UserRequestDTO**alsowith some more annotations. Then we can define **custom message for each field** in the request. Let’s look at the code…

**DTO Layer: UserRequestDTO**

@Data  
@Builder  
public class UserRequestDTO { **@NotBlank(message = "Invalid Name: Empty name")  
 @NotNull(message = "Invalid Name: Name is NULL")  
 @Size(min = 3, max = 30, message = "Invalid Name: Must be of 3 - 30 characters")**  
 String name;  **@Email(message = "Invalid email")**  
 String email;  **@NotBlank(message = "Invalid Phone number: Empty number")  
 @NotNull(message = "Invalid Phone number: Number is NULL")  
 @Pattern(regexp = "^\\d{10}$", message = "Invalid phone number")**  
 String mobile; **@Min(value = 1, message = "Invalid Age: Equals to zero or Less than zero")  
 @Max(value = 100, message = "Invalid Age: Exceeds 100 years")**  
 Integer age;}

There are some constraints I have used. I will explain their purposes. Every validation constraint is having a **separate message** which **should be shown when the constraint is violated**.

* **NotBlank** — Check whether the field value is not empty. Used for string type fields
* **NotNull** — Check whether the field value is not NULL. Can be used for any type of field like string / list / map
* **Size** — Check whether the field value is within the defined ranges(min and max)
* **Email** — Check whether the field is an email(following email common format)
* **Pattern** — Check whether the field is following a defined regex format
* **Min** — Check whether the field value is smaller than the defined min value
* **Max** — Check whether the field value is greater than the defined max value

Okay! Now our DTO is ready to be validated. But the next problem is how to connect this with the flow? Am I right? Before going for that, I will provide the service layer also and proceed.

**Service Layer: UserService**

@Autowired  
private UserRepository userRepository;public User saveUser(**UserRequestDTO userRequest**) {  
 User user = User.builder()  
 .name(userRequest.getName())  
 .email(userRequest.getEmail())  
 .mobile(userRequest.getMobile())  
 .age(userRequest.getAge())  
 .build();  
 return userRepository.save(user);  
}

Now start the service and see what’s happening! Let’s send an invalid request now. I have put age as 288 which is over 100 which violates our DTO field constraint.

{  
 "name": "John",  
 "email": "john@gmail.com",  
 "mobile": "1234567890",  
 **"age": 288**  
}

What is the output??? You should see this now. It’s bad request!

{  
 "timestamp": "2022-07-19T14:18:11.880+00:00",  
 "status": 400,  
 "error": "Bad Request",  
 "path": "/api/users"  
}

Let’s look into console logs…We have this log!!! You can see a clear explanation is there but in a complicated way. I have highlighted the focusing parts.

2022-07-19 19:55:28.759 WARN 850311 --- [nio-5000-exec-7] .w.s.m.s.DefaultHandlerExceptionResolver : Resolved [org.springframework.web.bind.**MethodArgumentNotValidException**: Validation failed for argument [0] in public org.springframework.http.ResponseEntity<com.rest.userservice.entities.User> **com.rest.userservice.controller.UserController.saveUser(com.rest.userservice.dtos.UserRequestDTO): [Field error in object 'userRequestDTO' on field 'age': rejected value [288]; codes [Max.userRequestDTO.age,Max.age,Max.java.lang.Integer,Max]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [userRequestDTO.age,age]; arguments []; default message [age],100]; default message [Invalid Age: Exceeds 100 years]] ]**

Things to understand:

* It is throwing **MethodArgumentNotValidException**exception. We have to handle this exception.
* It has considered the **default message: Invalid Age: Exceeds 100 years** which we have defined to be shown when age is greater than 100 (Hold on and look at the DTO again)

We are now getting closer to our goal! **We have the error, what is wrong and reasons also! Program is also not broke!**

*Only remaining thing is how to deal and inform the user? Am I right???*

Let’s look into that…

**Use a Global Exception Handler**

Spring Boot provides a way to **handle any kind of exception using a centralized way**! That is the beauty of this framework!

*It’s called****Controller Advice****and we have****annotation called “@RestControllerAdvice”****which is used at class level. We need to create a separate class to handle exceptions and annotate it with RestController.*

Ideally this exception handler class should be live with the application starting process. This annotation is an **extension of ControllerAdvice and ResponseBody**, which is considered as a**“@Component”** end of the day. So, Spring framework itself **create the bean** of this class automatically and returns it it the IOC container. I have put their internal implementation to clarify this idea.

// RestControllerAdvice annotation  
@Target({ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
**@ControllerAdvice**  
@ResponseBody  
public @interface **RestControllerAdvice** {  
  
}// ControllerAdvice annotation  
@Target({ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
**@Component**  
public @interface **ControllerAdvice** {  
  
}

Let’s create the exception handler. I have created a separate method called **handleValidationErrors()** which **takes the exception type we expect: (**MethodArgumentNotValidException), when DTO constraints are violated. And then we should annotate the method with the below annotation to let the Spring Boot know: this is the exact method which handles the so called **MethodArgumentNotValidException**always.

@ExceptionHandler(MethodArgumentNotValidException.class)

This **MethodArgumentNotValidException**exceptionclass provides some more methods and fields which are very useful. I have taken the field errors list out of exception and mapped all error messages into another list. Then all errors are written into a Map as a common way. And I have set the status a **BAD\_REQUEST(400 status code) for this kind of exception**. Then this status will be shown exactly in POSTMAN response result.

**Configs Layer: GlobalExceptionHandler**

@RestControllerAdvice  
public class GlobalExceptionHandler {  
  
 **@ExceptionHandler(MethodArgumentNotValidException.class)**  
 public ResponseEntity<Map<String, List<String>>> handleValidationErrors(MethodArgumentNotValidException ex) {  
 List<String> errors = ex.getBindingResult().getFieldErrors()  
 .stream().map(FieldError::getDefaultMessage).collect(Collectors.toList());  
 return new ResponseEntity<>(getErrorsMap(errors), new HttpHeaders(), **HttpStatus.BAD\_REQUEST**);  
 }  
  
 private Map<String, List<String>> getErrorsMap(List<String> errors) {  
 Map<String, List<String>> errorResponse = new HashMap<>();  
 errorResponse.put("errors", errors);  
 return errorResponse;  
 }  
  
}

Following this way, front-end can check **if the response contains “errors” key and show error alerts** with the messages available in the errors list.

Let’s try the same INVALID request again and check the output! The status I have set is highlighted.

Graphical user interface, text, application

Description automatically generated

**Let’s do more errors** and monitor!!! I have given

* Age greater than 100
* Email without “@” sign
* Phone number with 8 numbers
* Name with 2 characters

Now see the result guys!! This is awesome!!! **All errors are crystal clear to the user**.

Graphical user interface, text

Description automatically generated

This is why validation is very important! Now user is not confused if the front-end is managing these errors correctly!

**Handle Other Exceptions 💥**

You already saw how to deal with *MethodArgumentNotValidException* right? Following the same way, we can define some more exception handling methods. Let’s define and understand next.

**@ExceptionHandler(UserNotFoundException.class)**  
public ResponseEntity<Map<String, List<String>>> handleNotFoundException(UserNotFoundException ex) {  
 List<String> errors = Collections.singletonList(ex.getMessage());  
 return new ResponseEntity<>(getErrorsMap(errors), new HttpHeaders(), **HttpStatus.NOT\_FOUND**);  
}  
  
**@ExceptionHandler(Exception.class)**  
public final ResponseEntity<Map<String, List<String>>> handleGeneralExceptions(Exception ex) {  
 List<String> errors = Collections.singletonList(ex.getMessage());  
 return new ResponseEntity<>(getErrorsMap(errors), new HttpHeaders(), **HttpStatus.INTERNAL\_SERVER\_ERROR**);  
}  
  
**@ExceptionHandler(RuntimeException.class)**public final ResponseEntity<Map<String, List<String>>> handleRuntimeExceptions(RuntimeException ex) {  
 List<String> errors = Collections.singletonList(ex.getMessage());  
 return new ResponseEntity<>(getErrorsMap(errors), new HttpHeaders(), **HttpStatus.INTERNAL\_SERVER\_ERROR**);  
}

If any **Exception or RuntimeException** is thrown, it will return response including the error with 500 status code. If the requested **User is not found**, it will come into **UserNotFoundException** handling method and return the response including the thrown message with 404 status code.

I will demonstrate **UserNotFoundException s**cenario. Let’s create another controller method to *fetch a specific user from database*.

@GetMapping(path = "/users/{id}")  
public ResponseEntity<User> getUserByIdPath(@PathVariable Long id) {  
 return ResponseEntity.ok().body(userService.getUserById(id));  
}

In the**Service layer** we can check the*user is available or not*. If user is not found, we will throw our custom exception.

public User getUserById(Long id) {  
 return userRepository.findById(id)  
 .orElseThrow(() -> **new UserNotFoundException("No user by ID: " + id)**);  
}

You got it right? Our custom exception is also a Runtime Exception defined as follows.

**@ResponseStatus(HttpStatus.NOT\_FOUND)**  
public class **UserNotFoundException extends RuntimeException** {  
 public UserNotFoundException(String message) {  
 super(message);  
 }  
 public UserNotFoundException(String message, Throwable cause) {  
 super(message, cause);  
 }  
}

Restart the application and let’s search for a user who is not there…

<http://localhost:5000/api/users/33>

See the result next! The exact message we put in the throw statement is shown! And the status also! How beautiful it is??? Now application user knows that this is an INVALID user!

Graphical user interface, application, email

Description automatically generated

Code Repository on GitHub: <https://github.com/SalithaUCSC/spring-boot-validation-and-exceptions>

That’s all about **validation** and **exception handing** with Spring Boot. Article is little bit lengthy 😃 I had to explain all the necessary details. Hope you would get a great experience after reading this!!! This is pretty cool..We can **validate any kind of requests** any **handle kind of breaking scenarios** in a proper way. The most important thing is **we can inform the end user**! The end user is not alone when an error happens 😎 ❤️

Try this in your implementations and conquer the framework guys…

Bye bye!