

(/)

Displaying Error Messages with Thymeleaf in Spring

Last updated: January 8, 2024



Written by: [baeldung \(https://www.baeldung.com/author/baeldung\)](https://www.baeldung.com/author/baeldung)



Reviewed by: [Michal Aibin \(https://www.baeldung.com/editor/michal-author\)](https://www.baeldung.com/editor/michal-author)

Spring MVC (<https://www.baeldung.com/category/spring/spring-web/spring-mvc>)

Thymeleaf (<https://www.baeldung.com/tag/thymeleaf>)

Get started with Spring and Spring Boot, through the *Learn Spring* course:

>> CHECK OUT THE COURSE ([/ls-course-start](#))

1. Overview

In this tutorial, we'll see **how to display error messages originating from a Spring-based back-end application in Thymeleaf ([/spring-boot-crud-thymeleaf](#)) templates**.

For our demonstration purposes, we'll create a simple Spring Boot User Registration app and validate the individual input fields. Additionally, we'll see an example of how to handle global-level errors.

First, we'll quickly set up the back-end app and then come to the UI part.

2. Sample Spring Boot Application

To create a simple Spring Boot app for User Registration, **we'll need a controller, a repository, and an entity**.

However, even before that, we should add the Maven dependencies.

2.1. Maven Dependency

Let's add all the Spring Boot starters we'll need – Web (https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web) for the MVC bit, Validation (https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-validation) for hibernate entity validation, Thymeleaf (https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-thymeleaf) for the UI and JPA (https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-jpa) for the repository. Furthermore, we'll need an H2 (https://mvnrepository.com/artifact/com.h2database/h2) dependency to have an in-memory database:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
  <version>1.4.200</version>
</dependency>
```

2.2. The Entity

Here's our *User* entity:

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @NotEmpty(message = "User's name cannot be empty.")
    @Size(min = 5, max = 250)
    private String fullName;

    @NotEmpty(message = "User's email cannot be empty.")
    private String email;

    @NotNull(message = "User's age cannot be null.")
    @Min(value = 18)
    private Integer age;

    private String country;

    private String phoneNumber;

    // getters and setters
}
```

As we can see, we've **added a number of validation constraints (//spring-boot-bean-validation) for the user input**. Such as, fields should not be null or empty and have a specific size or value.

Notably, we haven't added any constraint on the *country* or *phoneNumber* field. That's because we'll use them as an example for generating a global error, or an error not tied to a particular field.

2.3. The Repository

We'll use a simple JPA repository (//the-persistence-layer-with-spring-data-jpa) for our basic use case:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {}
```

2.4. The Controller (//)

Finally, to wire everything together at the back-end, let's put together a *UserController*.

```
@Controller
public class UserController {

    @Autowired
    private UserRepository repository;
    @GetMapping("/add")
    public String showAddUserForm(User user) {
        return "errors/addUser";
    }

    @PostMapping("/add")
    public String addUser(@Valid User user, BindingResult result,
        Model model) {
        if (result.hasErrors()) {
            return "errors/addUser";
        }
        repository.save(user);
        model.addAttribute("users", repository.findAll());
        return "errors/home";
    }
}
```

Here we're defining a *GetMapping* (/spring-new-requestmapping-shortcuts) at the path */add* to display the registration form. Our *PostMapping* at the same path deals with validation when the form is submitted, with subsequent save to the repository if all goes well.

3. Thymeleaf Templates With Error Messages

Now that the basics are covered, we've come to the crux of the matter, that is, creating the UI templates and displaying error messages, if any.

Let's construct the templates piecemeal based on what type of errors we can display.

3.1. Displaying Field Errors

Thymeleaf offers an inbuilt *field.hasErrors* method that returns a boolean depending on whether any errors exist for a given field. Combining it with a *th:if* ([/spring-mvc-thymeleaf-conditional-css-classes#using-thif](#)) we can choose to display the error if it exists:

```
<p th:if="${#fields.hasErrors('age')}">Invalid Age</p>
```



Next, if we want **to add any styling, we can use *th:class*** ([/spring-mvc-thymeleaf-conditional-css-classes#using-thclass](#)) **conditionally**:

```
<p th:if="${#fields.hasErrors('age')}"  
th:class="${#fields.hasErrors('age')}? error">  
    Invalid Age</p>
```



Our simple embedded CSS class *error* turns the element red in color:

```
<style>  
    .error {  
        color: red;  
    }  
</style>
```



Another Thymeleaf attribute *th:errors* gives us the ability to display all errors on the specified selector, say *email*:

```
<div>  
    <label for="email">Email</label> <input type="text" th:field="*  
{email}" />  
    <p th:if="${#fields.hasErrors('email')}" th:errorclass="error"  
th:errors="*{email}" />  
</div>
```



In the above snippet, we can also see a variation in using the CSS style. Here **we're using *th:errorclass*, which eliminates the need for us to use any conditional attribute for applying the CSS.**

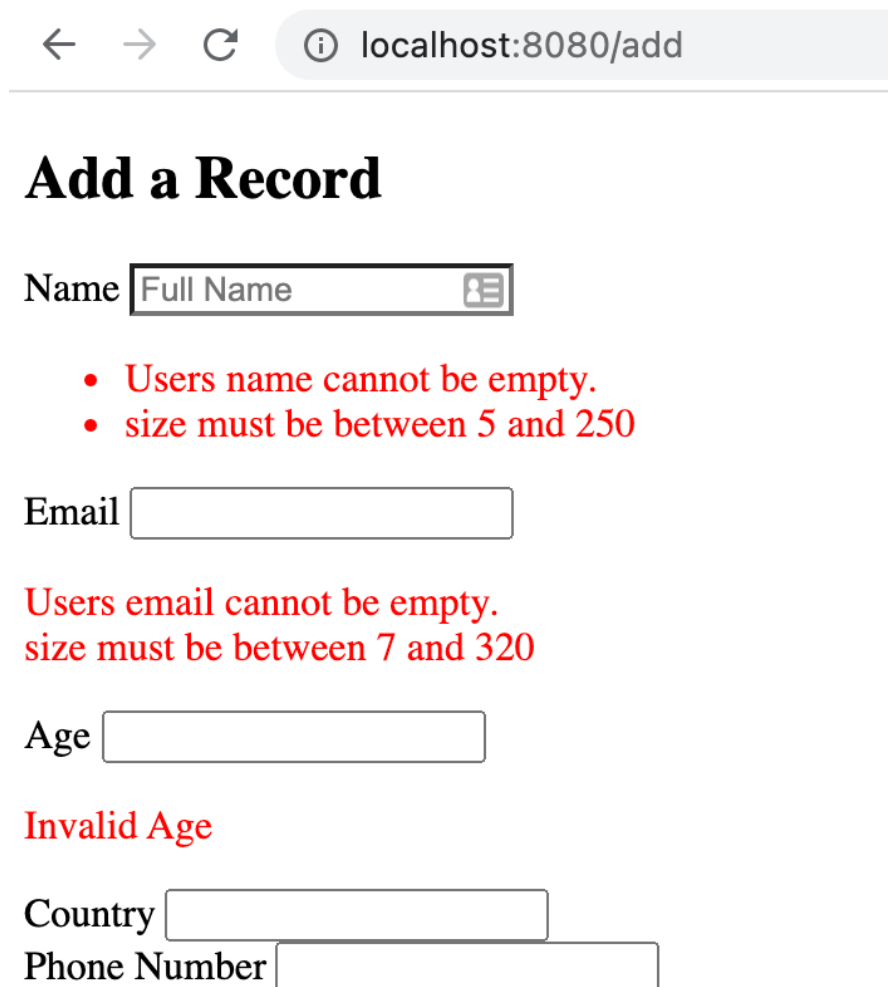
Alternatively, we may choose to iterate over all validation messages on a given field using *th:each* ([/thymeleaf-iteration](#)):

```
<div>                                (/  
    <label for="fullName">Name</label> <input type="text" th:field="*  
{fullName}"  
        id="fullName" placeholder="Full Name">  
    <ul>  
        <li th:each="err : ${#fields.errors('fullName')}}"  
th:text="${err}" class="error" />  
    </ul>  
</div>
```

Notably, we used another Thymeleaf method *fields.errors()* here to collect all validation messages returned by our back-end app for the *fullName* field.

Now, to test this, let's fire up our Boot app and hit the endpoint *http://localhost:8080/add* (*http://localhost:8080/add*).

This is how our page looks when we don't supply any input at all:



The screenshot shows a web browser at the URL *localhost:8080/add*. The page title is "Add a Record". The form contains several input fields with associated validation errors:

- Name**: Input field with placeholder "Full Name".
 - Users name cannot be empty.
 - size must be between 5 and 250
- Email**: Input field.
 - Users email cannot be empty.
 - size must be between 7 and 320
- Age**: Input field.
 - Invalid Age
- Country**: Input field.
- Phone Number**: Input field.

(/wp-content/uploads/2021/04/Thymeleaf_fieldErrs.png)

3.2. Displaying All Errors at Once

Next, let's see how instead of showing every error message one by one, we can show it all in one place.

For that, **we'll use Thymeleaf's *fields.hasAnyErrors()* method:**

```
<div th:if="${#fields.hasAnyErrors()}">
  <ul>
    <li th:each="err : ${#fields.allErrors()}" th:text="${err}" />
  </ul>
</div>
```

As we can see, we used another variant *fields.allErrors()* here to iterate over all the errors on all the fields on the HTML form.

Instead of *fields.hasAnyErrors()*, we could have used *#fields.hasErrors()*. Similarly, *#fields.errors()* is an alternate to *#fields.allErrors()* that was used above.

Here's the effect:

All errors in place:

- Users email cannot be empty.
- Users age cannot be null.
- Users name cannot be empty.
- size must be between 7 and 320
- size must be between 5 and 250

(/wp-content/uploads/2021/04/Thymeleaf_allErrors.png)

3.3. Displaying Errors Outside Forms

Next, let's consider a scenario wherein we want to display validation messages outside an HTML form.

In that case, **instead of using selections or *{!...!}*, we simply need to use the fully-qualified variable name in the format *\${!...!}*:**


```
<h4>Error on a single field:</h4>
<div th:if="${#fields.hasErrors('${user.email}}'"
    th:errors="*{user.email}"></div>
<ul>
    <li th:each="err : ${#fields.errors('user.*')}}" th:text="${err}"
/>
</ul>
```

This would display all error messages on the *email* field.

Now, **let's see how we can display all the messages at once:**

```
<h4>All errors:</h4>
<ul>
<li th:each="err : ${#fields.errors('user.*')}}" th:text="${err}" />
</ul>
```

And here's what we see on the page:

This is outside the form:

Errors on a single field:

Users email cannot be empty.
size must be between 7 and 320

All errors:

- Users age cannot be null.
- size must be between 5 and 250
- Users email cannot be empty.
- Users name cannot be empty.
- size must be between 7 and 320

(/wp-content/uploads/2021/04/Thymeleaf_outsideForm.png)

3.4. Displaying Global Errors

In a real-life scenario, there might be errors not specifically associated with a particular field. We might have a use case where **we need to consider multiple inputs in order to validate a business condition**. These are called global errors.

Let's consider a simple example to demonstrate this. For our *country* and *phoneNumber* fields, we might add a check that for a given country, the phone numbers should start with a particular prefix.

We'll need to make a few changes on the back-end to add this validation.

First, we'll add a *Service* (/spring-component-repository-service) to perform this validation:

```
@Service
public class UserValidationService {
    public String validateUser(User user) {
        String message = "";
        if (user.getCountry() != null && user.getPhoneNumber() !=
null) {
            if (user.getCountry().equalsIgnoreCase("India")
                && !user.getPhoneNumber().startsWith("91")) {
                message = "Phone number is invalid for " +
user.getCountry();
            }
        }
        return message;
    }
}
```

As we can see, we added a trivial case. For the country *India*, the phone number should start with the prefix *91*.

Second, we'll need a tweak to our controller's *PostMapping*.

```
@PostMapping("/add")
public String addUser(@Valid User user, BindingResult result, Model model) {
    String err = validationService.validateUser(user);
    if (!err.isEmpty()) {
        ObjectError error = new ObjectError("globalError", err);
        result.addError(error);
    }
    if (result.hasErrors()) {
        return "errors/addUser";
    }
    repository.save(user);
    model.addAttribute("users", repository.findAll());
    return "errors/home";
}
```

Finally, in the Thymeleaf template, **we'll add the constant *global* to display such type of error:**

```
<div th:if="${#fields.hasErrors('global')}">
    <h3>Global errors:</h3>
    <p th:each="err : ${#fields.errors('global')}" th:text="${err}"
    class="error" />
</div>
```

Alternatively, instead of the constant, we can use methods *#fields.hasGlobalErrors()* and *#fields.globalErrors()* to achieve the same.

This is what we see on entering an invalid input:

Add a Record

Name	<input type="text" value="Jane Doe"/>
Email	<input type="text" value="jdoe@abc.com"/>
Age	<input type="text" value="22"/>
Country	<input type="text" value="India"/>
Phone Number	<input type="text" value="9044334"/>

Global errors:

Phone number is invalid for India

(/wp-content/uploads/2021/04/Thymeleaf_global.png)

4. Conclusion

In this tutorial, we built a **simple Spring Boot Application to demonstrate how to display various types of errors in Thymeleaf**.

We looked at displaying field errors one by one and then all at one go, errors outside HTML forms, and global errors.

As always, source code is available over on GitHub (<https://github.com/eugenp/tutorials/tree/master/spring-web-modules/spring-thymeleaf-3>).

Get started with Spring and Spring Boot, through the *Learn Spring* course:

>> CHECK OUT THE COURSE (/ls-course-end)



Learning to build your API with **Spring**?

Download the E-book (</rest-api-spring-guide>)

Comments are closed on this article!

COURSES

[ALL COURSES \(/ALL-COURSES\)](/all-courses)

[ALL BULK COURSES \(/ALL-BULK-COURSES\)](/all-bulk-courses)

[ALL BULK TEAM COURSES \(/ALL-BULK-TEAM-COURSES\)](/all-bulk-team-courses)

[THE COURSES PLATFORM \(HTTPS://COURSES.BAELDUNG.COM\)](https://courses.baeldung.com)

SERIES

[JAVA "BACK TO BASICS" TUTORIAL \(/JAVA-TUTORIAL\)](/java-tutorial)

[JACKSON JSON TUTORIAL \(/JACKSON\)](/jackson)

[APACHE HTTPCLIENT TUTORIAL \(/HTTPCLIENT-GUIDE\)](#)

[REST WITH SPRING TUTORIAL \(/REST-WITH-SPRING-SERIES\)](#)

[SPRING PERSISTENCE TUTORIAL \(/PERSISTENCE-WITH-SPRING-SERIES\)](#)

[SECURITY WITH SPRING \(/SECURITY-SPRING\)](#)

[SPRING REACTIVE TUTORIALS \(/SPRING-REACTIVE-GUIDE\)](#)

ABOUT

[ABOUT BAELDUNG \(/ABOUT\)](#)

[THE FULL ARCHIVE \(/FULL_ARCHIVE\)](#)

[EDITORS \(/EDITORS\)](#)

[JOBS \(/TAG/ACTIVE-JOB/\)](#)

[OUR PARTNERS \(/PARTNERS\)](#)

[PARTNER WITH BAELDUNG \(/ADVERTISE\)](#)

[TERMS OF SERVICE \(/TERMS-OF-SERVICE\)](#)

[PRIVACY POLICY \(/PRIVACY-POLICY\)](#)

[COMPANY INFO \(/BAELDUNG-COMPANY-INFO\)](#)

[CONTACT \(/CONTACT\)](#)