OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

INFORMATIK UND
MATHEMATIK

Softwareentwicklung (SW)

Validierung des Modells

Prof. Dr. Alixandre Santana
alixandre.santana@oth-regensburg.de

Wintersemester
2024/2025

Lernziele

- Das Konzept der Validierung im Backend und Frontend zu verstehen
- Die Annotation @Valid zu verstehen und anzuwenden
- Validierungsfehlermeldungen in dem View anzuzeigen
- Einen benutzerdefinierten Validator zu implementieren

Agenda

1. Arten der Validierung
2. Annotations für Valiedierung
3. Fehlermeldungen in der Ansicht anzeigen/Dynamic CSS
4. Benutzerdefinierter Validator

1. Validierung

- Die Validierung spielt eine sehr wichtige Rolle, wenn wir Features, REST-APIs oder Microservices erstellen.
- Auf diese Weise können wir den Benutzer darüber informieren, welche Daten gesendet werden müssen und wie sie sein sollten.
- Wenn das erwartete Format nicht gefunden wird, können wir dem Benutzer eine Fehlermeldung senden.

1. Validierung : server-side oder client-side?

388

As others have said, you should do both. Here's why:

Client Side

You want to validate input on the client side first because you can give **better feedback to the average user**. For example, if they enter an invalid email address and move to the next field, you can show an error message immediately. That way the user can correct every field **before** they submit the form.

If you only validate on the server, they have to submit the form, get an error message, and try to hunt down the problem.

(This pain can be eased by having the server re-render the form with the user's original input filled in, but client-side validation is still faster.)

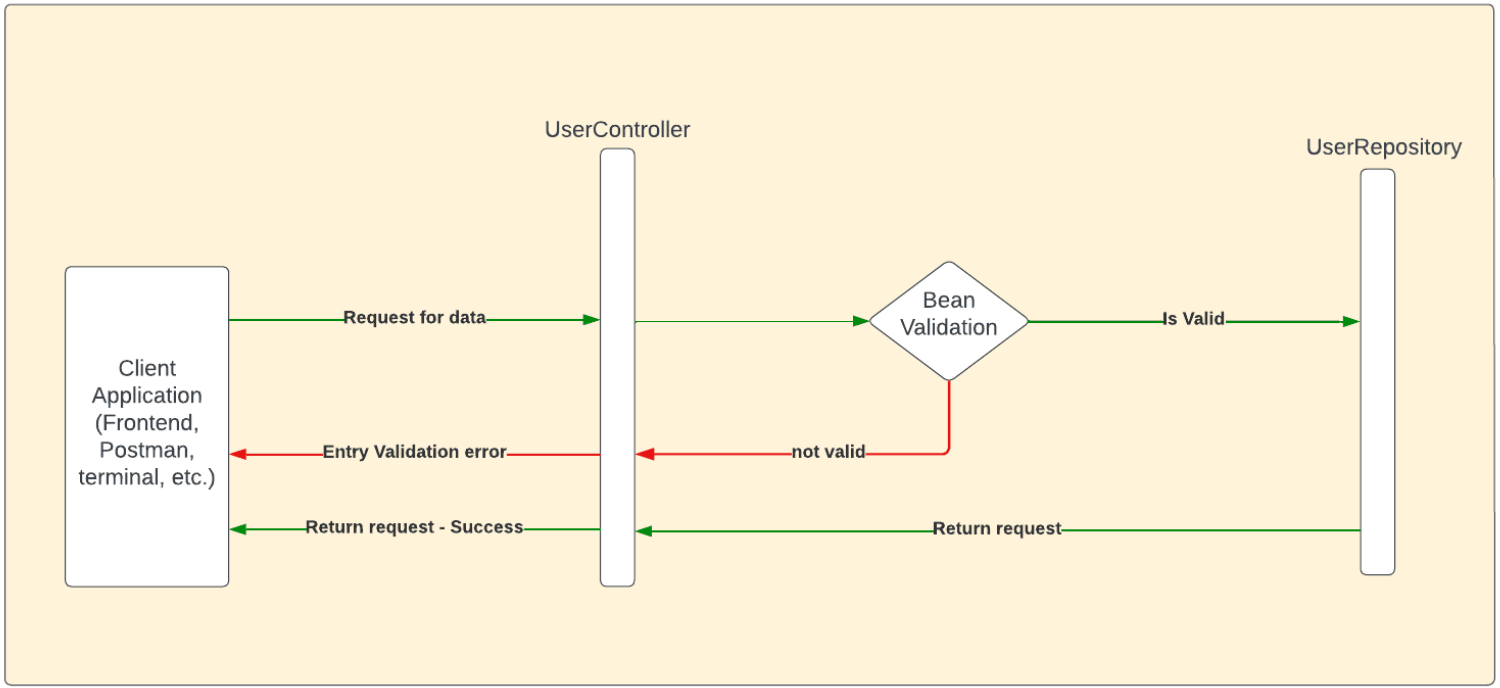
Server Side

You want to validate on the server side because you can **protect against the malicious user**, who can easily bypass your JavaScript and submit dangerous input to the server.

It is very dangerous to trust your UI. **Not only can they abuse your UI, but they may not be using your UI at all, or even a browser**. What if the user manually edits the URL, or runs their own Javascript, or tweaks their HTTP requests with another tool? What if they send custom HTTP requests from `curl` or from a script, for example?

https://stackoverflow.com/questions/162159/javascript-client-side-vs-server-side-validation

1. Validierung : server-side oder client-side?



https://snyk.io/blog/guide-to-input-validation-with-spring-boot/

1. Mit Spring Boot, the JSR-303 Bean Validation API

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

Agenda

1. Arten der Validierung
2. Annotations für Valiedierung
3. Fehlermeldungen in der Ansicht anzeigen/Dynamic CSS
4. Benutzerdefinierter Validator

2. Annotations für Validierung

@Min and @Max

```
@Min(value = 18, message = "Age should not be less than 18")
@Max(value = 150, message = "Age should not be greater than 150")
private int age;
...
```

2. Annotations für Validierung

@NotNull, @NotEmpty and @NotBlank

@NotNull: Erlaubt keinen Nullwert

@NotEmpty: Zusätzlich dazu, dass kein Nullwert zulässig ist, muss die Größe größer als Null sein. Wird für Strings, Arrays, Listen usw. verwendet.

@NotBlank: Zusätzlich dazu, dass kein Nullwert zulässig ist, muss die Größe des Strings nach dem Trimmvorgang (Entfernen von nachgestellten und führenden Leerzeichen) größer als Null sein. Das wird für Strings verwendet.

2. Annotations für Validierung

@Size

```
@Size(min = 10, max = 200, message  = "About Me must be between 10 and 200 characters")

private String aboutMe;
```

2. Annotations für Validierung

@Email

```
@Email(message = "Email should be valid")
    private String email;
```

2. Annotations für Validierung

Und mehr....

@Positive and ***@PositiveOrZero*** gelten für numerische Werte und validieren, dass sie streng positiv oder positiv einschließlich 0 sind.

@Negative and ***@NegativeOrZero*** gelten für numerische Werte und bestätigen, dass sie streng negativ oder negativ einschließlich 0 sind.

@Past and ***@PastOrPresent*** bestätigen, dass ein Datumswert in der Vergangenheit oder in der Vergangenheit einschließlich der Gegenwart liegt; kann auf Datumstypen angewendet werden, einschließlich der in Java 8 hinzugefügten.

@Future *and* ***@FutureOrPresent*** bestätigen, dass ein Datumswert in der Zukunft oder in der Zukunft einschließlich der Gegenwart liegt.

@Digits ....

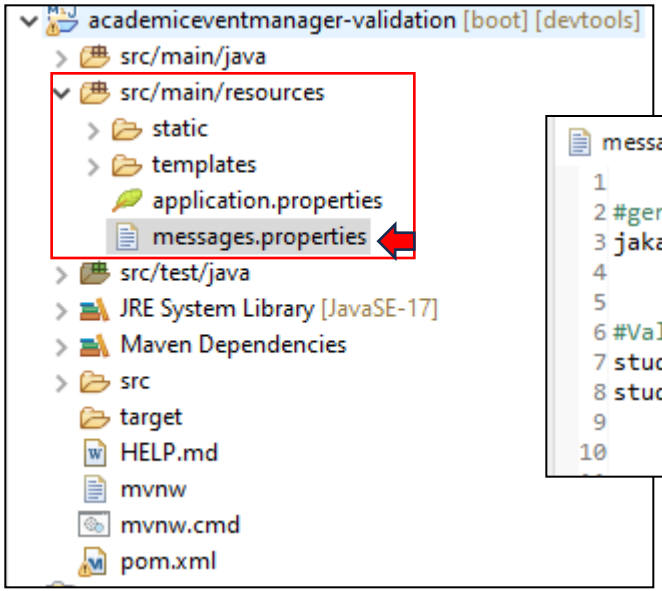
2. 1 die Entität mit Validierungsanmerkungen

```
public class Student {

Long id;

@Size(min = 3, max = 50, message="Name should have between 3 and 50
characters")
String name=null;

@NotBlank(message = "{student.email.not.blank}")
String email;

private LocalDate birthDate;

....
}
```

2. 2 Aktualisierung des Controllers zur Validierung des Modells

```
@RequestMapping(value = "/students/add/process")
public String addStudent(@ModelAttribute @Valid Student studentRequest,
BindingResult result, RedirectAttributes attr){

     if (result.hasErrors()) {
     System.out.println(result.getErrorCount());
     System.out.println(result.getAllErrors());
     return "/students/student-add";
     }

     attr.addFlashAttribute("success", "Student added!");
     return "redirect:/students/list";
}
```

2. 3 messages.properties Datei

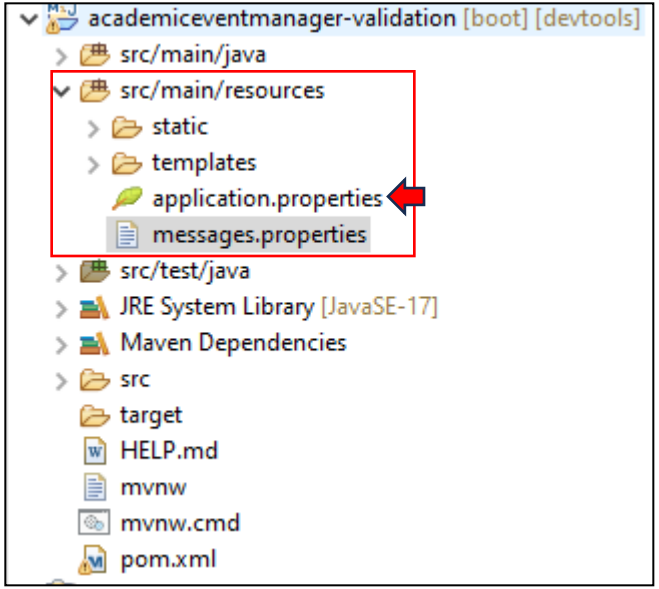


messages.properties

```
1
2 #general messages
3 jakarta.validation.constraints.Size= must have between {min} and {max} characteres.
4
5
6 #Validation for Students:
7 student.email.not.blank= email of the student cannot be blank.
8 student.birthDate.less16 =student should be at least 16 years old.
9
10
```

https://www.baeldung.com/spring-custom-validation-message-source

2. 3 messages.properties Datei



application.properties

```
1  spring.thymeleaf.cache=false
2
3  spring.messages.basename=messages
4  server.error.include-binding-errors=always
5
```

https://www.baeldung.com/spring-custom-validation-message-source

Agenda

1. Arten der Validierung
2. Annotations für Valiedierung
3. Fehlermeldungen in dem View anzeigen/Dynamic CSS
4. Benutzerdefinierter Validator

3. Anzeige der Validierungsmeldungen in dem View

Thymeleaf bietet eine integrierte Methode „fields.hasErrors()“, die einen Booleschen Wert zurückgibt, je nachdem, ob für ein bestimmtes Feld Fehler vorliegen. In Kombination mit einem „th:if“ können wir festlegen, dass der Fehler angezeigt wird, wenn er vorliegt:?

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

Wenn wir Stilelemente hinzufügen möchten, können wir bedingt th:class verwenden:

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

Ein weiteres Thymeleaf-Attribut, th:errors, gibt uns die Möglichkeit, alle Fehler im angegebenen Selektor anzuzeigen, z.B. E-Mail:

```
<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>
```

https://www.baeldung.com/spring-thymeleaf-error-messages
https://www.baeldung.com/spring-mvc-thymeleaf-conditional-css-classes

3. Anzeige der Validierungsmeldungen in dem View

```
<input type="date" class="form-control" id="birthdate" th:field="*{birthdate}"
th:classappend="${#fields.hasErrors('birthdate')} ? 'is-invalid'" />

<div class="invalid-feedback">
    <span th:errors="*{birthdate}"></span>
</div>
```

https://www.baeldung.com/spring-custom-validation-message-source

Agenda

1. Arten der Validierung
2. Annotations für Valiedierung
3. Fehlermeldungen in dem View anzeigen/Dynamic CSS
4. Benutzerdefinierter Validator

4.1 Implementierung eines benutzerdefinierten Validators

```
public class StudentValidator implements Validator {


@Override
public boolean supports(Class<?> clazz) {
// TODO Auto-generated method stub
return Student.class.equals(clazz);
}

@Override
public void validate(Object target, Errors errors) {
// TODO Auto-generated method stub
      Student student = (Student) target;
      long age = ChronoUnit.YEARS.between(student.getBirthDate(), LocalDate.now());
      System.out.println("Age: " + age);
      if (age<16) {
      errors.rejectValue("birthDate", "student.birthDate.less16");
      }
}
}
```

https://docs.spring.io/spring-framework/reference/core/validation/validator.html

4.2 Informieren des Controllers über den Validator

StudentController.java

```
1  package de.othr.aem.controller;
2
3
4  import org.springframework.stereotype.Controller;
17
18 @Controller
19 public class StudentController {
20
21
22     @InitBinder
23     public void initBinder(WebDataBinder binder) {
24         binder.addValidators(new StudentValidator());
25     }
26
```

4.3 Implementierung eines benutzerdefinierten Validators

Package org.springframework.validation

Interface BindingResult

All Superinterfaces:

Errors

All Known Implementing Classes:

AbstractBindingResult, AbstractPropertyBindingResult, BeanPropertyBindingResult, BindException, DirectFieldBindingResult, MapBindingResult, MethodArgumentNotValidException, WebExchangeBindException

---

```
public interface BindingResult
extends Errors
```

General interface that represents binding results. Extends the Errors interface for error registration capabilities, allowing for a Validator to be applied, and adds binding-specific analysis and model building.

Serves as result holder for a DataBinder, obtained via the DataBinder.getBindingResult() method. BindingResult implementations can also be used directly, for example to invoke a Validator on it (e.g. as part of a unit test).

Since:

2.0

Author:

Juergen Hoeller

See Also:

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/validation/BindingResult.html

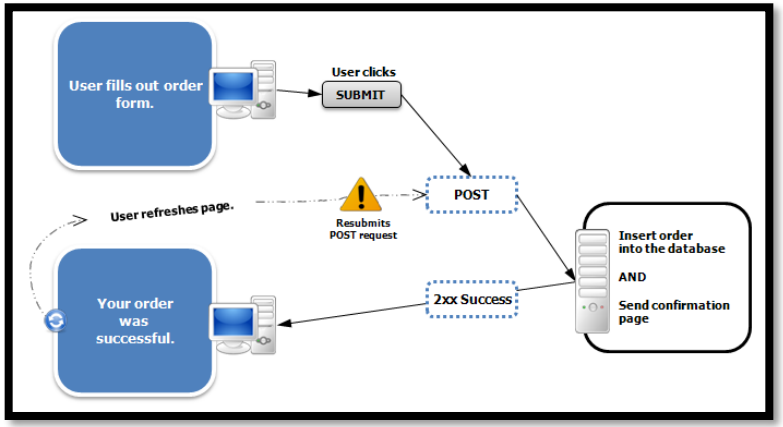
4.4 Implementierung eines benutzerdefinierten Validators

```
@RequestMapping(value = "/students/add/process")
public String addStudent(@ModelAttribute @Valid Student studentRequest,
        BindingResult result,
        RedirectAttributes attr){


    if (result.hasErrors()) {
        System.out.println(result.getErrorCount());
        System.out.println(result.getAllErrors());
        return "/students/student-add";

    }

    //studentService.save(student);
    attr.addFlashAttribute("success", "Student added!");
    return "redirect:/students/list";
}
```

Extra:

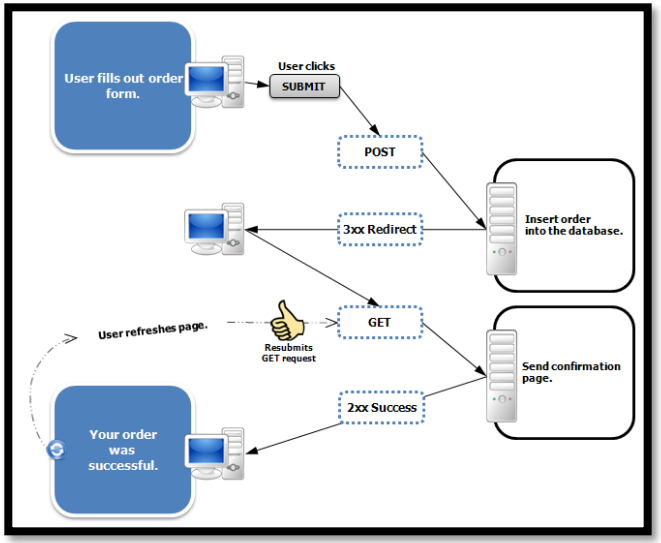
```
@RequestMapping(value = "/students/add/process")
public String addStudent(@ModelAttribute @Valid Student studentRequest,
        BindingResult result,
        RedirectAttributes attr){


    if (result.hasErrors()) {
        System.out.println(result.getErrorCount());
        System.out.println(result.getAllErrors());
        return "/students/student-add";

    }

    //studentService.save(student);
    attr.addFlashAttribute("success", "Student added!");
    return "redirect:/students/list";
}
```

4.4 Implementierung eines benutzerdefinierten Validators



X

Post-Redirect-Get Pattern



https://www.geeksforgeeks.org/post-redirect-get-prg-design-pattern/

Referenzen

https://jeromejaglale.com/doc/spring4_tutorial/form_validation

https://www.baeldung.com/spring-boot-bean-validation

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

https://dzone.com/articles/spring-custom-validations

https://www.baeldung.com/spring-mvc-thymeleaf-conditional-css-classes