* Spring ne ofera posibilitatea de a crea un system de validare a beanurilor, adica a datelor introduse de user in form

**Validation Annotations**

* @NotNull(message=”Mesaj eroare”) – valoarea nu trebuie sa fie Null
* @Min – e pentru numere, sa aiba o valoare minima
* @Max – pentru numere, sa aiba o valoare maxima
* @Size(min=x, message=”Error mess”) – un numar de caractere dintr-un text
* @Pattern
* @Future/@Past – data trebuie sa fie intr-o viitoare or precedenta data
* @NotBlank – nu poate fi fara caractere sau doar cu spatii
* @Email – verifica daca un email e valid

**Tool for Validation**

* Java a creat Java’s standart Bean Validation API(JSR-303) pentru validari
* Dar ea are nevoie de o implementare pentru a putea fi utilizata, si aici ne ajuta Hibernate
* Hibernate ofera si o JSR-303 implementation, ca un proiect total separat pentru validari
* Totusi, Hibernate Validator 7 e bazat pe Jakarta EE9
* Jakarta EE este versiunea comunitara a Java EE. Java EE e o colectie de API enterprise ca Servlets, JSP etc.
* Jakarta EE nu inlocuieste Java EE
* La jakarta nu folosim javax.\* ci jakarta.\* si tot asa
* Hibernate Validator 7 e bazat pe Jakarta EE 9
* Spring 5 e inca bazat pe componente Java EE, deci nu e compatibil cu Hibernate Validator 7
* Insa, Spring 5 e compatibil cu Hibernate Validator 6.2, ce nu e pentru Jakarta projects, si e identic cu cel 7

**Validation System**

* Principiul de functionare la Validation System e ca la form. Odata ce userul trimite o forma, datele se verifica in Controller, si daca ceva nu e bine, Model Object pe care l-am trimis la Pagina si a venit cu datele la Controller, e trimis inapoi la pagina cu mesajul de eroare.

**Create a project with Hibernate**

**Daca dorim sa lucram cu Jakarta, punem in pom.xml doar asta:**

<dependency>  
 <groupId>org.hibernate.validator</groupId>  
 <artifactId>hibernate-validator</artifactId>  
 <version>6.0.21.Final</version>  
</dependency>

**ATENTIE! Nu punem VERSIUNE MAI MARE DE 6!!!**

**SAU**

**Daca cu JavaX,punem ceea ce e sus si mai punem si asa:**

<dependency>  
 <groupId>javax.validation</groupId>  
 <artifactId>validation-api</artifactId>  
 <version>2.0.1.Final</version>  
</dependency>

**Validation Process**

**1.**

* Trebuie sa adaugam anotatiile deasupra membrilor clasei care vrem sa fie supusi procesului de validare.
* In @NotNull, de ex, (message=”ERROR”) indica mesajul de eroare ce va aparea daca validarea esueaza. Daca nu punem message, va aparea unul default.
* Putem pune si mai mule anotatii

public class Client {  
 private String firstName;  
  
 @NotNull(message = "is required")  
 @Size(min=1)  
 private String lastName;

}

**2.**

* In fisierul HTML, facem totul ca deobicei, doar ca vom mai adauga si un nou tag:

<form:errors path=”” cssClass=””/>

Path – este identic cu elementul de deasupra, pentru a arata ca e anume eroarea ce vine de la el

cssClass – e clasa CSS pentru afisarea mesajului de eroare

<**form:form** modelAttribute="client" action="processForm">  
 First Name <**form:input** path="firstName"/>  
 <br><br>  
  
 Last Name(\*) <**form:input** path="lastName"/>  
 <**form:errors** path="lastName" cssClass="error"/>  
  
 <br><br>  
 <input type="submit" value="Submit">  
</**form:form**>

Acest <form:errors/> e afisat doar daca exista erori si deci poate fi aifisat doar dupa ce forma a fost trimisa si verificata.

**3.**

* In metoda care preia requiestul, adica il mapeaza, la parametrul @ModelAttribute(“customer”) adaugam in fata @Valid.

@Valid arata ca se vor aplica regulile de validare stabilite asupra obiectului venit.

* Urmatorul parametru trebuie mereu sa fie BidingResult result;

Acest obiect va contine rezultatul in urma procesului de validare. Spring mereu va introduce acest rezultat dupa obiectul ce are @ModelAttribute din lista de parametri, de aceea el trebuie sa fie mereu dupa el.

@RequestMapping("/processForm")  
public String registerPage(@Valid @ModelAttribute("client") Client client,  
 BindingResult bindingResult){  
  
 if(bindingResult.hasErrors()){  
 return "form";  
 }  
 else {  
 return "register";  
 }  
}

Metoda hasErrors() returneaza True daca sunt errori, si false daca nu. Daca sunt errori, returnam inapoi forma, daca nu, pagina de registrare.

Daca exista erori, obiectul BindingResult va fi trimis catre pagina din return si mesajul de eroare, setat tot de noi, va fi afisat in <form:errors/>. Deja Spring are un mod de a face asta.

**Problema cu Null after Validation cu @Value() la Map**

Avem o problema cand datele nu se valideaza. Deoarece in forma, se creaza un obiect nou, de tipul celui trimis de la Cotroller, cu atributul dat, si se trimite la Controller cu atributul dat, adica il inlocuieste pe obiectul venit, vom avea problema ca anumiti membrii din Obiect vor fi Null, adica nesetati, asa cum a esuat validarea si obiectul, deoarece nu mai e creat de Spring Container, ci de JSP, nu a primit nimic de la anotatii, ca @Value, caci anotatiile pot fi folosite doar cand obiecul e creat de Spring Container. Apoi, noi vom trimtie deja acest obiect inapoi la pagina web, dar el deja va fi cu multe date nesetate, inclusiv Maps, si deci forma va avea eroare la afisarea datelor NULL.

Pentru a rezolva problema data, trebuie sa salvam datele sesiunii cu anotatia deasupra clasei Controller

@SessionAttributes(value = {"client"}, types = {Client.class})

Unde “client” e numele **atributului**, nu al beanului!Anotatia data face ca Forma sa nu creeze un obiect nou de tipul celui venit in care sa puna valorile trimise de forma, dar sa seteze aceste valori anume in obiectul venit de la Controller. Intre {} punem atributele obiectelor care vor face exceptie, adica nu se vor crea in pagina obiecte noi de tipul lor, ci vor fi folosite ele.

**@InitBinder – alternativa la @NotBlank + ceva nou**

In versiunile cand nu era @NotBlank, Chiar daca s-a pus conditia ca numele sa nu fie mai mic de 2 caractere, introducerea de spatii libere oricum va trece de verificare, asta daca avem doar @NotNull. Totusi, metoda asta poate elimina patiile automat. In Controller, punem asta:

@InitBinder  
public void initBinder(WebDataBinder data){  
 StringTrimmerEditor stringTrimmerEditor = new StringTrimmerEditor(true);  
 data.registerCustomEditor(String.class, stringTrimmerEditor);  
}

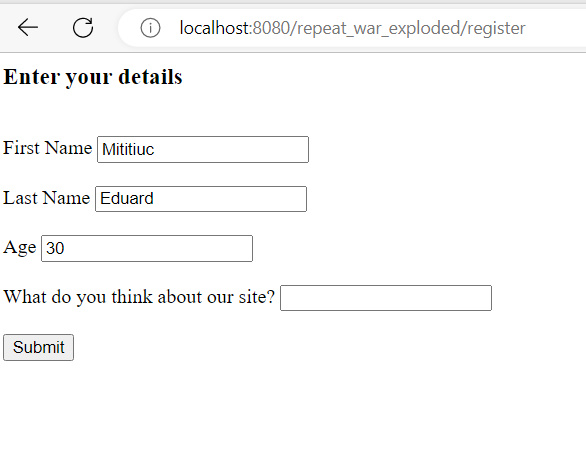
**@InitBinder** – este ca un pre-processor. Ea se ocupa de fiecare web request care vine la Controller pana ca ea sa ajunga la el.

StringTrimmerEditor stringTrimmerEditor = new StringTrimmerEditor(true); - elimina spatiile libere. Daca exista doar spatii libere, returneaza Null, asta e rolul lui true aici.

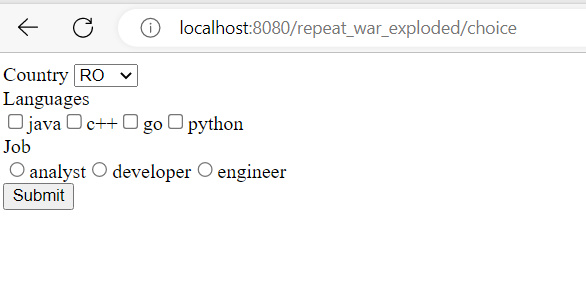
**dataBinder.registerCustomEditor(String.class, stringTrimmerEditor**) – aici spunem ca pentru fiecare String object, aplicam StringTrimmerEditor. Asa va pre-procesa sau verifica fiecare String din form data si daca sunt spatii libere, le elimina, dar daca sunt doar spatii libere, returneaza Null.

**Validation Groups**

1. Procesul de validare poate fi imparit pe etape. De ex, sa zicem ca intai vrem ca userul nostru sa introduca pe o forma niste date de baza, ca numele prenumele si apoi dupa ce trece de forma asta, sa fie dus pe alta, cu alte date, gen job, etc.



apoi



Pentru asta, am creat un obiect Student cu validari:

public class Student {  
  
 @NotNull(message = "Can't be null")  
 @NotBlank(message = "Can't be empty")  
 @Size(min = 3, max = 20, message = "You name is too long or short")  
 private String firstName;  
 @NotNull(message = "Can't be null")  
 @NotBlank(message = "Can't be empty")  
 @Size(min = 3, max = 20, message = "You name is too long or short")  
 private String lastName;  
  
 @Min(value = 18,message = "Too young")  
 @Max(value = 100, message = "Too old")  
 private int age;  
  
 @Value("#{countries}")  
 private HashMap<String,String> country;  
 @Value("#{jobs}")  
 private HashMap<String,String> job;  
 @Value("#{langs}")  
 private HashMap<String,String> lang;  
  
 @NotNull(message = "Can't be null")  
 private String countries;  
 @NotNull(message = "Can't be null")  
 private String jobs;  
 @NotNull(message = "Can't be null")  
 private String[] langs;

.......

}

Si in controller:

@Controller  
@SessionAttributes("student")  
public class MyController {  
  
 @GetMapping("/")  
 public String index(Model model){  
 ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("classpath:applicationContext.xml");  
 model.addAttribute("student",context.getBean("student",Student.class));  
 return "index";  
 }  
  
 @PostMapping("/register")  
 public String register(@Valid @ModelAttribute("student") Student student, BindingResult result, HttpServletRequest request){  
 System.*out*.println(student);  
 System.*out*.println(request.getParameter("opinion"));  
 if(result.hasErrors())  
 return "index";  
 return "choice";  
 }  
 @GetMapping("/choice")  
 public String choice(@ModelAttribute("student")Student student, Model model){  
 return "choice";  
 }  
 @PostMapping("/processChoice")  
 public String processChoice(@Valid @ModelAttribute("student")Student student,BindingResult result){  
 System.*out*.println(student);  
 if(result.hasErrors())  
 return "choice";  
 return "info";  
 }  
}

* Problema e ca, in prima pagina, desi vom introduce corect toate datele, daca apasam pe validare, iarasi vom fi returnati pe ea, si totul pare a fi corect. Insa, asta se intampla din cauza ca au ramas niste fileduri ce nu trec validarea,caci ele nici nu sunt pe prima pagina, si anume

@NotNull(message = "Can't be null")  
@NotEmpty(message = "can't be empty")  
private String countries;  
@NotNull(message = "Can't be null")  
@NotEmpty(message = "can't be empty")  
private String jobs;  
@NotNull(message = "Can't be null")  
private String[] langs;

De ele ne ocupam abea dupa ce trecem de prima pagina, dar Spring nu stie de asta, el crede ca noi nu le-am setat in prima pagina, nu stie ca ele inca urmeaza sa fie setate in a 2 pagina

* Pentru a rezolva problema asta, trebuie sa cream niste Validation Groups. **Un Validation group** este un grup de fielduri ce trebuie validate impreuna, separat de oricare altele existene. Spring va valida doar acel group pe care il vom seta noi, si gata, nu se va atinge de acele fielduri ce nu fac parte din acel group.

**Create Validate Groups**

1. Fiecare grup va fi o interfata, dar nu trebuie sa facem nimic in ea, doar le cream si atat. Avem nevoie doar de numele ei si gata.

public interface Group1 {  
}

public interface Group2 {  
 }

1. Acum, fiecare anotatie pentru validare, are un parametru groups = “”, deci la fiecare anotatie acum vom specifica carui grup apartine:

public class Student {  
  
 @NotNull(message = "Can't be null",groups = Group1.class)  
 @NotBlank(message = "Can't be empty",groups = Group1.class)  
 @Size(min = 3, max = 20, message = "You name is too long or short",groups = Group1.class)  
 private String firstName;  
 @NotNull(message = "Can't be null",groups = Group1.class)  
 @NotBlank(message = "Can't be empty",groups = Group1.class)  
 @Size(min = 3, max = 20, message = "You name is too long or short",groups = Group1.class)  
 private String lastName;  
  
 @Min(value = 18,message = "Too young",groups = Group1.class)  
 @Max(value = 100, message = "Too old",groups = Group1.class)  
 private int age;  
  
 @Value("#{countries}")  
 private HashMap<String,String> country;  
 @Value("#{jobs}")  
 private HashMap<String,String> job;  
 @Value("#{langs}")  
 private HashMap<String,String> lang;  
  
 @NotNull(message = "Can't be null",groups = Group2.class)  
 @NotEmpty(message = "can't be empty",groups = Group2.class)  
 private String countries;  
 @NotNull(message = "Can't be null",groups = Group2.class)  
 private String jobs;  
 @NotNull(message = "Can't be null")  
 private String[] langs;



Numele grupului e inerfata.class

Putem pune si mai multe grupuri

1. Acum, in loc de @Valid, care valideaza toate fieldurile si toate anotatiile, vom folosi @Validated(interfata.class)
2. @Controller  
   @SessionAttributes("student")  
   public class MyController {  
     
    @GetMapping("/")  
    public String index(Model model){  
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("classpath:applicationContext.xml");  
    model.addAttribute("student",context.getBean("student",Student.class));  
    return "index";  
    }  
     
    @PostMapping("/register")  
    public String register(@Validated(Group1.class) @ModelAttribute("student") Student student, BindingResult result, HttpServletRequest request){  
    System.*out*.println(student);  
    System.*out*.println(request.getParameter("opinion"));  
    if(result.hasErrors())  
    return "index";  
    return "choice";  
    }  
    @GetMapping("/choice")  
    public String choice(@ModelAttribute("student")Student student, Model model){  
    return "choice";  
    }  
    @PostMapping("/processChoice")  
    public String processChoice(@Validated(Group2.class) @ModelAttribute("student")Student student,BindingResult result){  
    System.*out*.println(student);  
    if(result.hasErrors())  
    return "choice";  
    return "info";  
    }  
   }



**Asa, fiecare request se va ocupa de validarea doar unei parti din fielduri si anotatii deci, adica doar cele ce fac parte din acest grup. Nu conteaza daca alte fielduri si anotatii din alt grup nu ar trece validarea, ele nu mai sunt luate in seama.**