1.9.3 Ecriture des formulaires

Bon pour relecture

Sommaire :

Exemple

L'écriture des formulaires *JSF* ne déroutera pas l'habitué des formulaires *JSP*. On utilisera par exemple :

```
<h:form id="administratorAddForm">
  <h:messages />
  <h:outputLabel
     for="ldapUid"
     value="#{msgs['ADMINISTRATOR_ADD.TEXT.PROMPT']}" />
  <h:inputText
      id="ldapUid"
      value="#{administratorsController.ldapUid}"
      required="true" />
  <h:message for="ldapUid" />
  <h:commandButton
     value="#{msgs['ADMINISTRATOR_ADD.BUTTON.ADD_ADMIN']}"
     action="#{administratorsController.addAdmin}" />
  <h:commandButton
     value="#{msgs['_.BUTTON.CANCEL']}"
     action="cancel"
     immediate="true" />
</h:form>
```

L'attribut **value** de la balise **<h:inputText>** contient une référence vers un attribut du contrôleur. Cet attribut sera mis à jour lors de la validation du formulaire. L'attribut **action** du bouton **<h:commandButton>** contient une référence vers la *callback* (méthode) du contrôleur. C'est elle qui sera appelée lors de l'appui sur le bouton et dont le résultat sera utilisé par les règles de navigation pour connaître la page que l'application doit afficher en retour.
Le deuxième bouton a un attribut **immediate="true"**. Dans ce cas, les attributs du contrôleur relatifs aux balises **<h:inputText>** ne seront pas mis à jour et les éventuelles vérifications de forme ou de contenu ne seront pas exécutées. Ceci est particulièrement utile sur un bouton d'annulation comme c'est le cas ici.

Voir tous les composants ici http://www.horstmann.com/corejsf/jsf-tags.html

Les balises **<h:messages>** et **<h:message>** sont traitées dans un paragraphe à suivre.

Exercice : Ajouter une entrée dans la barre de navigation Afficher l'énoncé

Exercice : Ajouter une page JSF Afficher l'énoncé

Exercice : Créer une règle de navigation Afficher l'énoncé

Exercice : Créer un contrôleur Afficher l'énoncé

Messages d'erreur

Dans une page *JSP* les balises **<h:messages>** et **<h:message>** permettent d'afficher des messages d'erreur à l'utilisateur.La balise **<h:messages>** permet d'afficher l'ensemble des messages d'erreurs. La balise **<h:message>** permet d'afficher les messages d'erreurs relatifs à une balise **<h:inputText>** particulière. Elle dispose, à cet effet, d'un attribut **for** qui lui permet de faire le lien avec un attribut **id** d'une balise **<h:inputText>** donnée. Ce mécanisme est notamment utilisé quand l'attribut **required** de la balise **<h:inputText>** est positionné à true ou bien quand un validateur (cf. ci-dessous) lui est associé.Il est aussi possible, dans une *callback* d'un contrôleur, de remonter des messages d'erreurs vers la vue. Ces messages seront généralement rendus grâce à la balise **<h:messages>**. Typiquement, la *callback* renverra **null**, c'est-à-dire que la page affichée à l'utilisateur restera la même (celle où s'est produite l'erreur de saisie). Dans l'exemple suivant, on affiche un message si l'utilisateur correspondant à l'identifiant donné est déjà administrateur :

```
User user = getDomainService().getUser(ldapUid);
if (user.getAdmin()) {
  addErrorMessage(
    "form:uid",
    "ADMINISTRATORS.MESSAGE.USER_ALREADY_ADMINISTRATOR",
    ldapUid);
  return null;
}
```

Le premier paramètre de la méthode **addErrorMessage** vaut **"uid"**. Le message ajouté sera alors affiché par la balise **<h:message for="uid">**. Il est également possible de spécifier un premier paramètre **null** ; le message d'erreur sera global, c'est-à-dire affiché par la balise **<h:messages>**.

Exercice : Afficher un message sur une page JSF	Afficher l'énoncé

Validation des formulaires

Validation au niveau des JSP

Utilisation des validateurs prédéfinis

Il existe des validateurs par défaut dans *JSF* (**validateLength**, **validateLongRange** et **validateDoubleRange**), par exemple :

```
<h:inputText id="age" value="#{testController.age}">
  <f:validateLongRange minimum="18"/>
</h:inputText>
<h:message for="age"/>
```

On peut trouver d'autres validateurs que ceux fournis par défaut, *Tomahawk* propose par exemple **validateCreditCard**, **validateUrl**, **validateEmail**, **validateEqual** et **validateRegExpr**.

Exercice : Utiliser un validateur prédéfini	Afficher l'énoncé

Développement de validateurs personnalisés

Il est aussi possible d'écrire ses propres validateurs. Leur mise en œuvre est relativement simple, par exemple :

```
<h:inputText
    id="age"
    value="#{testController.age}"
    validator="#{bean.validateAge}">
</h:inputText>
```

La méthode **validateAge** de **bean** ressemblera à :

```
public void validateAge(
    FacesContext context,
    UIComponent componentToValidate,
    Object value) throws ValidatorException {
  if (...) {
    throw new ValidatorException(getFacesErrorMessage("MESSAGE.ALWAYS_ERROR"));
  }
}
```

Exercice : Écrire un validateur	Afficher l'énoncé

Validation au niveau des beans (JSR 303)

http://www.jmdoudoux.fr/java/dej/chap-validation_donnees.htm
http://musingsofaprogrammingaddict.blogspot.com/2009/01/getting-started-with-jsr-303-beans.html
http://www.touilleur-express.fr/2008/06/06/jsr-303-vous-avez-valide-votre-bean/
http://www.openscope.net/2010/02/08/spring-mvc-3-0-and-jsr-303-aka-javax-validation

Au cours du développement d'une application web on constate la nécessité de valider les objets manipulés à différents niveau :

- Couche métier (au moment de la validation d'un formulaire)
- Dans la couche domaine
- Au niveau du mapping objet-base
- Dans la base (Contrainte sur les colonnes)

On en vient souvent à dupliquer les contraintes à chaque niveau et de façon pas toujours cohérentes (ex: on limite un champs à 50 caractères dans un formulaire alors que la colonne correspondante en base est un varchar(100)

Pour répondre à cela la JSR 303 propose une spécification standard pour la validation des beans.
On utilisera alors les annotations introduites en Java 5 pour exprimer les contraintes sur le bean.

Exemple :

```
package org.esupportail.example.web.beans;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class UserBean{
        /**
         * Id of the user.
         */
        @NotNull
        private String id;

        /**
         * Display Name of the user.
         */
        @Size(max = 10, min = 1)
        private String displayName;


[...]
}
```

Pour utiliser javax.validation il faudra alors ajouter dans le pom.xml la dépendance suivante :

```
<dependency>
 <groupId>org.hibernate</groupId>
 <artifactId>hibernate-validator</artifactId>
 <version>4.0.2.GA</version>
</dependency>
```

Exemple de contraintes :

- @AssertFalse : doit être faux
- @AssertTrue : doit être vrai
- @DecimalMax : doit être un nombre inférieur ou égal à la valeur indiquée (représentation String d'un BigDecimal)
- @DecimalMin : doit être un nombre supérieur ou égal à la valeur indiquée (représentation String d'un BigDecimal)
- @Digits(integer=, fraction=) : doit être un chiffre (pour BigDecimal, BigInteger, String, byte, short, int, long)
- @Future : doit être une date dans le future
- @Max : doit être un nombre inférieur ou égal à la valeur indiquée
- @Min : doit être un nombre supérieur ou égal à la valeur indiquée
- @NotNull : ne doit pas être null
- @Null : doit être null
- @Past : doit être une date dans le passé
- @Size(min=, max=) : doit avoir une taille comprise entre min et max (pour String et tableaux)
- @Pattern(regex=, flag=) : doit respecter la regex indiquée
- @Valid : doit être valide (pour un objet propriété d'un autre)

On peut également imaginer un bean qui cumulerait les annotations nécéssaires à la validation et les annotations nécessaires à la persistance.

Intégration Hibernate

http://www.hibernate.org/subprojects/validator.html
Implémentation hibernate validator intègre la validation du bean au niveau de la persistance.
elle propose quelques contraintes supplémentaires à celles de la spécifictation JSR 303 comme :

- @CreditCardNumber : doit être un numéro de carte bleue
- @Email : doit être un e-mail
- @Length(min=, max=) : doit avoir un taille comprise entre min et max (taille de la colonne en base sera max)
- @NotBlank : ne doit pas être null ou vide ou composée d'espace (pour une chaîne)
- @NotEmpty : ne doit pas être null ou vide
- @URL(protocol=, host=, port=) : doit être une URL
- @Range(min=, max=) : doit avoir une valeur comprise entre Min et Max

Intégration JSF

JSF 2 intègre par défaut la validation des beans (JSR 303)
http://www.mastertheboss.com/web-interfaces/293-jsf-validation-tutorial.html?start=2
Dès lors, les pages JSF n'ont plus besoin de contenir les information de validation du bean. Lorsque une validation de contrainte échoue, les messages d'erreur associés sont automatiquement traduits en *FacesMessage* par l'implementation JSF.

On pourra toutefois les désactiver :

```
<h:inputText value="#{sampleBean.userName}">
    <f:validateBean disabled="true" />
</h:inputText>
```

Gestion des messages

On peut définir le message qui sera affiché en cas d'erreur de validation en précisant un attribut message dans l'annotation

```
@Size(max = 10, min = 1, message="{toto}")
```

Ici on utilise une clé (chaîne entre accolades) afin d'aller chercher le message dans un fichier de propriétés. Ce fichier doit se trouver dans le classpath et porter le nom **ValidationMessages.properties**

On peut le décliner ne fonction des langues **ValidationMessages_en.properties**, **ValidationMessages_fr.properties**

Mise à jour de propriétés par les formulaires (updateActionlistener)

updateActionListener est une balise de la librairie *JSF Tomahawk* qui est particulièrement utile. C'est un *listener* qui est associé à une balise permettant une action (bouton, lien) qui, au moment où ce dernier est activé, va lire le contenu de son attribut **value** pour l'assigner à la référence contenue dans son attribut **property**.On utilisera par exemple :

```
<h:commandButton
    action="deleteUser"
    value="#{msgs['BUTTON.DELETE']}">
  <t:updateActionListener
      value="#{user}"
      property="#{controller.userToDelete}" />
</h:commandButton>
```

Ici, l'action **deleteUser** va diriger l'utilisateur vers une autre page (typiquement une page de demande de confirmation avant d'effacer l'administrateur), via le fichier de règles de navigation. La présence de la balise **updateActionListener** fait que le moteur *JSF* aura, avant d'effectuer le changement de page, positionné la valeur de **#{user}** dans l'attribut **userToDelete** du contrôleur **controller**. La page de confirmation de l'effacement pourra alors faire appel à ce contrôleur pour générer un contenu en fonction de la valeur de cet attribut.

Il est à noter que, dans cet exemple *JSF*, **user** est un objet de type complexe et pas simplement une chaîne de caractères comme on peut en avoir l'habitude en développement web classique.

Exercice : Utiliser un updateActionListener	Afficher l'énoncé

Ordre des opérations en JSF

- Appel des validateurs,
- Affectation des valeurs des entrées de formulaires,
- Appel des **updateActionListeners**,
- Appel de la **callback** d'action du formulaire.

Conversion des types complexes

Il existe des convertisseurs par défaut dans JSF (**DateTimeConverter** et **NumberConverter**). Ils permettent de transformer une date ou un nombre suivant différentes règles, par exemple :

```
<h:outputText value="#{testController.date}">
   <f:convertDateTime dateStyle="short" locale="#{sessionController.locale}"/>
</h:outputText>
```

Dans certains cas, il est aussi nécessaire de définir des convertisseurs manuellement. C'est notamment le cas pour les listes déroulantes. Considérons que l'on veuille afficher une liste déroulante permettant à l'utilisateur de choisir un objet de type **Locale**. Pour la génération du HTML, JSF a besoin d'une représentation textuelle de chaque objet de la liste. De même, il a besoin de pouvoir retrouver un objet depuis un choix de l'utilisateur qui correspond à la valeur textuelle de la balise **<option>** du formulaire HTML. C'est le rôle du convertisseur **converter** de faire ce travail :

```
<h:selectOneMenu id="locale" onchange="submit();"
                 value="#{preferencesController.locale}"
                 converter="#{localeConverter}">
   <f:selectItems value="#{preferencesController.localeItems}"/>
</h:selectOneMenu>
```

Ici, la liste déroulante est constituée d'objets de type **Locale**. Le convertisseur **localeConverter** est défini (par convention dans *esup-commons*) dans le fichier **/properties/web/converters.xml** :

```
<bean id="localeConverter"
      class="org.esupportail.commons.web.converters.LocaleConverter">
  <description>
     A converterforLocale objects.
  </description>
</bean>
```

La classe **LocaleConverter** implémente l'interface **Converter** de JSF.

JSF et accessibilité

Un des objectifs de l'utilisation de JSF est, via un standard de haut niveau, de tendre vers plus d'accessibilité (WAI : *Web Accessibility Initiative*). L'accessibilité des applications doit être une préoccupation constante des programmeurs, qui ne doivent pas hésiter à tester leurs applications en utilisant des navigateurs pauvres tels que Lynx. En règle générale, on évitera absolument d'utiliser les balises **h:commandLink** qui, à cause de l'utilisation de Javascript, brisent toutes les règles de l'accessibilité. On préfèrera dans tous les cas des boutons (**h:commandButton**). Javascript peut néanmoins être utilisé pour améliorer l'IHM des applications, en faisant attention à ce que les navigateurs ne parlant pas Javascript puissent quand même utiliser l'application. Nous montrons ici à titre d'exemple comment on peut soumettre un formulaire par simple changement de la valeur d'une boite déroulante :

```
<h:form id="form">
   <h:messages />
   <h:outputLabel for="value" value="#{msgs['TEXT.VALUE']}"/>
   <h:selectOneMenu id="value" onchange="submit();" value="#{controller.value}">
      <f:selectItems value="#{controller.valueItems}"/>
   </h:selectOneMenu>
   <h:commandButton value="#{msgs['BUTTON.CHANGE']}" id="changeButton"/>
</h:form>
<script type="text/javascript">
   hideButton("form:changeButton");
</script>
```

Le bouton **changeButton** est caché par du codeJavascript.
Les clients qui parlent Javascript ne le voient pas ; ce n'est pas grave puisqu'un changement de valeur de la boite de sélection déroulante soumet automatiquement le formulaire (**onchange="submit();"**). Les clients qui ne parlent pas Javascript voient le bouton, et peuvent donc valider le formulaire.