JSF, quelques concepts avancés

Avec Netbeans 4.1

oschmitt@free.fr

Février 2006

Table des matières

1 Introduction

Nous avons vus comment construire des pages JSF simples dans le tutoriel précédent. Cependant, ces notions sont insuffisantes pour développer une application de gestion complète. (Voir le tutoriel : http://schmitt.developpez.com/tutoriel/java/jsf/introduction/)

Lors de la saisie de données par l'utilisateur il est souvent indispensable de valider et ou convertir les valeurs saisies avant de les injecter dans un JavaBean .

Dans le cas de développements importants, il est souvent utile de créer des composants réutilisables pour augmenter la productivité et faciliter la maintenance.

Ce tutoriel vous montrera comment compléter vos connaissances pour développer une application de gestion complète avec JSF.

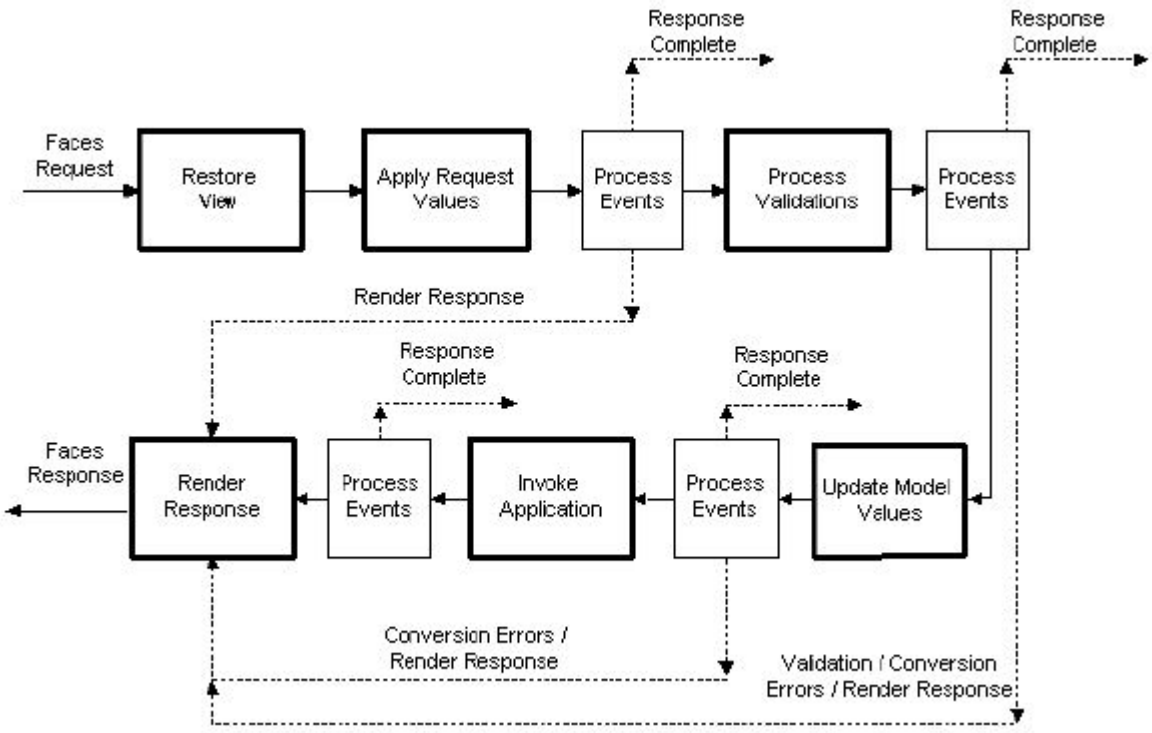
Remarque:

Ce tutoriel s'appuie sur le tutoriel précédent en ce qui concerne les exemples de code. Il est donc fortement recommandé de lire le tutoriel précédent.

2 Le cycle de vie d'une requête JSF

La compréhension du cycle de vie d'une requête JSF est cruciale.

Voici la représentation de ce cycle de vie issue des spécifications de JSF :



2.1 Restore View

Reprenons une page JSF simple telle que « hello-world.jsp » :

```
<%@ page contentType="text/html" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="html" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="core" %>
<core:view>

    <html:outputText value="Hello le World ! (en JSF !)" />

</core:view>
```

Cette page est destinée à un client web qui supporte le langage HTML.

Une page HTML est un fichier texte qui contient un assemblage de balises spécifiées par la norme HTML.

Cependant, nous avons vu que JSF utilise un arbre d'objets pour représenter une vue. La racine de cet arbre est une instance de la classe `UIViewRoot`. Il s'agit d'une structure de données bien différente d'un fichier texte.

Cette phase va donc consister à reconstituer l'arbre de composants qui correspond à la vue HTML de l'utilisateur qui soumet la requête. L'implémentation JSF utilise les informations contenues dans l'objet `HttpServletRequest` pour déterminer le bon algorithme :

- Si la vue est (hello-world.jsp) est demandée pour la première fois, il faut créer une instance de `UIViewRoot` et lui associer un nom (souvent le nom de la page JSP).
- Si la vue existe déjà pour JSF alors l'arbre de composants correspondant devient la vue courante.

Ceci implique que JSF mémorise l'arbre de composants, ce qui revient à sérialiser un objet de type `UIViewRoot`.

Cette sérialisation peut se faire du côté client (un champs caché dans un formulaire) ou du côté serveur.

La méthode `UIComponent.restoreState()` est appelée récursivement sur l'arbre de composants. Elle rétablit l'état du composant à partir de l'état sauvé lors de la sérialisation.

Lorsque JSF a déterminé la vue courante, celle-ci est accessible via la classe `FacesContext`.

```
FacesContext  facesContext = FacesContext.getCurrentInstance();
UIViewRoot viewRoot = facesContext.getViewRoot();
```

Ensuite JSF établit le binding pour tous les composants concernés. (voir chapitre 8 de Introduction à JSF)

2.2 Apply Request Values

Lorsque la phase précédente est terminée nous disposons d'un arbre de composants dont la racine est `UIViewRoot`. La requête HTTP soumise par le navigateur est porteuse des actions de l'utilisateur sur la vue.

Ceci implique qu'il faut synchroniser la vue côté JSF avec la vue côté client. En effet, si l'utilisateur modifie une valeur dans un champs de formulaire il faut que le composant graphique correspondant côté serveur reflète ce changement d'état.

Le but de cette phase est donc de répercuter les actions de l'utilisateur sur l'arbre de composants courant. Il faut décoder les valeurs contenues dans l'objet `HttpServletRequest` et répercuter les changement d'états de la vue sur les composants concernés.

A cet effet, la classe `UIComponent` qui est la classe ancêtre de tout composant JSF dispose de la méthode `UIComponent.processDecodes()`. Cette méthode est appelée récursivement sur chaque composant de l'arbre. Les composants ont la responsabilité de décoder les informations qui les concernent.

Le plus souvent, cela se traduit par la recherche d'une information dans les valeurs de la requête HTTP courante.

Tout composant a un client id (un identifiant client voir l'attribut id dans la page HTML), si un paramètre identifié par le client id du composant existe dans l'objet `HttpServletRequest` alors le composant s'empare de la valeur associée et effectue ce que bon lui semble.

```
/**
* Exemple de composant
*/
public class DummyComponent extends UICommand {

    public void processDecodes(FacesContext facesContext){
      String clientId = getClientId(facesContext);
      Map paramMap = facesContext.getExternalContext().getRequestParameterMap();
      if(paramMap.containsKey(clientId)){
            Object value = paramMap.get(clientId);
            // Faire quelque chose pour changer mon état
        }
    }
}
```

Notez bien que le seul le composant est éventuellement mis à jour ici.

2.3 Process Validations

Certains composants sont parfois liés à des JavaBean via la technique du value binding. Les JavaBeans représentent souvent les données métiers. Il est impératif de valider et ou de convertir les valeurs qui proviennent du client avant de les injecter dans le JavaBean.

En effet, si l'utilisateur saisi la chaîne « zezdsdsd » en lieu et place d'un nombre entier alors cela provoquera une erreur.

JSF offre les concepts de `Validator` et de `Converter` pour effectuer des traitements sur les valeurs soumises par l'utilisateur avant leurs injections dans les JavaBeans. (voir chapitre 5).

Un composant peut posséder des `Validator` et un `Converter` qui sont invoqués lors de l'appel de la méthode `UIComponent.processValidators()`. Cette méthode est appelée récursivement sur l'arbre de composants.

JSF procède d'abord à la conversion de la valeur extraite de la requête HTTP (String) puis valide cette valeur en utilisant les `Validator`.

Si une erreur survient lors de la validation alors JSF saute à la phase `RenderResponse`.

2.4 Update Models Values

Lorsque la requête arrive à cette phase de traitement, les composants sont dans un état qui correspond à la vue du client. Les informations de la requête HTTP qui sont destinées à mettre à jour les données métiers ont été validées.

La méthode `UIComponent.processUpdates()` est appelée récursivement sur l'arbre de composants. Sa responsabilité consiste à mettre à jour le modèle représenté par les JavaBeans.

Si une erreur survient lors de la validation alors JSF saute à la phase `RenderResponse`.

JSF offre un modèle événementiel qui permet de détecter les changements du modèle. L'appel de la méthode `UIComponent.processUpdates()` peut poster des évènements.

2.5 Invoke Application

Lorsque cette phase est atteinte le modèle a été mis à jour, des évènements sont en attentes dans la queue des évènements.

La méthode `UIComponent.processApplication()` est appelée récursivement sur l'arbre de composants. Sa responsabilité consiste à diffuser (broadcasting) certains évènements de la queue vers les écouteurs d'évènements associés ou listeners.

L'attribut phaseId d'un `FacesEvent` indique dans quelle phase il doit être diffusé. Ici, la valeur de cet attribut est `PhaseId.INVOKE_APPLICATION`.

2.6 Render Response

C'est la dernière phase du traitement d'une requête JSF. Sa première responsabilité consiste à encoder l'arbre de composants courant dans un langage compréhensible par le client, ici HTML.

Pour ce faire, le `UIComponent` dispose d'un jeu de méthodes dont la signature commence par `encodeXXX`. Ces méthodes sont particulièrement adaptées à l'encodage dans des langages à balises. Ces méthodes sont au nombre de trois : `encodeBegin()`, `encodeChildren()`, `encodeEnd()`.

Ces méthodes correspondent grossièrement aux étapes d'encodage d'une balise : balise ouvrante, balises filles, balises fermante. (voir chapitre 7).

Sa deuxième responsabilité consiste à sauvegarder l'état des composants. Pour ce faire une deuxième méthode `UIComponent.saveState()` est appelée récursivement . (Voir chapitre 7)

Cette méthode répond à `UIComponent.restoreState()` de la phase RestoreView. Ce comportement est donné par l'interface `StateHolder` qui est implémentée par `UIComponent`.

Le processus de sauvegarde de l'arbre de composants est plus complexe qu'exposé ici, ceci pour des besoins de clarté. Consultez les spécifications de JSF 1.1 ou 1.2 pour plus de détails.

URL des spécifications : http://java.sun.com/j2ee/javaserverfaces/download.html.

Le schéma du traitement de la requête indique qu'il est possible de sauter directement à la phase RenderResponse. Cette fonctionnalité est parfois nécessaire pour des cas d'utilisations complexes.

```
FacesContext  facesContext = FacesContext.getCurrentInstance();
facesContext.renderResponse();
```

3 Les messages

Dans le tutoriel précédent nous avions construit une page de liste de clients. Nous pouvions ajouter ou supprimer de nouveaux clients.

Nous allons ajouter une zone de message dédiée à cette liste.

3.1 Manipuler les messages

JSF dispose d'une API qui permet de manipuler des messages JSF dans votre application.

Tout d'abord le concept de message est modélisé par la classe `FacesMessage`.

Pour créer un message JSF il suffit d'instancier cette classe:

```
FacesMessage facesMessage = new FacesMessage();
facesMessage.setSeverity(FacesMessage.SEVERITY_INFO);
facesMessage.setSummary(summary)
facesMessage.setDetail(detail);
```

Un `FacesMessage` présente plusieurs caractéristiques :

- une sévérité : `SEVERITY_INFO, SEVERITY_WARN, SEVERITY_ERROR, SEVERITY_FATAL`
- un résumé
- un détail

Le code précédent crée un message. Ce code n'est pas suffisant il faut ajouter le message au contexte JSF courant afin que les composants graphiques de message puisse l'afficher. La classe `FacesContext` possède une méthode qui permet cela :

```
FacesMessage facesMessage = new FacesMessage();
facesMessage.setSeverity(FacesMessage.SEVERITY_INFO);
facesMessage.setSummary(« Un résumé de message »  )
facesMessage.setDetail(« Détail du message »);

FacesContext  facesContext = FacesContext.getCurrentInstance();
facesContext.addMessage(null,facesMessage);
```

Vous remarquerez que `FacesContext.addMessage()` possède deux paramètres. Le premier paramètre permet d'attacher un message à un composant particulier, dans ce cas, le paramètre a pour valeur l'identifiant du composant (attribut id sur les tags JSF). Si l'id est `null` alors ce message est global.

3.2 Afficher des messages

Reprenons le cas d'utilisation du chapitre 9 de « Introduction à JSF ».

La page JSP « data-table-mvc.jsp » :

```
<%@ page contentType="text/html" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="html" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="core" %>
<core:view>
    <html:form>
        <html:dataTable binding="#{bankCtrl.view.dataTable}"
             value="#{bankCtrl.model.datas.customers}"
             var="customer" border="1">
            <html:column>
                <html:selectBooleanCheckbox binding="#{bankCtrl.view.checkbox}"/>
            </html:column>
            <html:column>
                <core:facet name="header">
                    <core:verbatim>Nom</core:verbatim>
                </core:facet>
                <html:outputText value="#{customer.name}"/>
            </html:column>

            <html:column>
                <core:facet name="header">
                    <core:verbatim>Prénom</core:verbatim>
                </core:facet>
                <html:outputText value="#{customer.forname}"/>
            </html:column>
        </html:dataTable>
        <br>
        <html:commandButton value="Supprimer les clients"
             action="#{bankCtrl.removeSelectedCustomers}"/>
        <html:commandButton value="Ajouter un client"
             action="#{bankCtrl.addCustomer}"/>
    </html:form>
</core:view>
```

Nous allons ajouter une zone de message au-dessus de la liste. Pour ce faire, JSF propose deux tags dédiés à l'affichage des messages : `<html:messages>` et `<html:message>`.

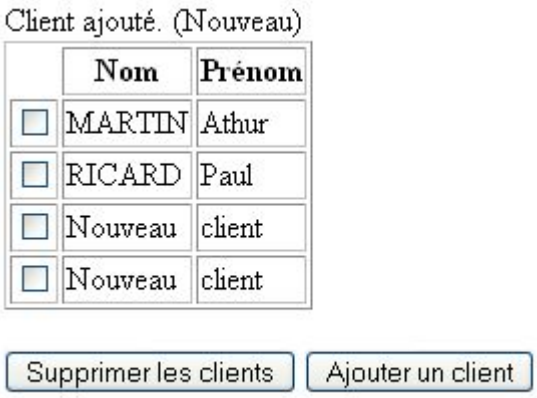
```
<%@ page contentType="text/html" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="html" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="core" %>
<core:view>
    <html:form>

        <html:messages showDetail="true"/>

        <html:dataTable binding="#{bankCtrl.view.dataTable}"
             ....
</core:view>
```

Il nous faut maintenant coder l'ajout de message dans le contexte JSF.

Ici, c'est la responsabilité du `Controller` que d'assumer la cinématique des opérations d'ajout et de suppression.

```
package com.facestut.mvc;

import com.facestut.bean.Bank;
import com.facestut.bean.Customer;

public class BankListController implements Controller {
    ...
     public void removeSelectedCustomers(){
        Bank bank = (Bank) getModel().getDatas();
        Collection selectedObjects = getView().getSelectedObjects();
        bank.getCustomers().removeAll(selectedObjects);
        String summary = "Les clients ont été supprimés.";
        String detail = selectedObjects.toString();
        addMessage(null,summary,detail,FacesMessage.SEVERITY_INFO);
    }

    public void addCustomer(){
        Bank bank = (Bank) getModel().getDatas();
        Customer customer = new Customer();
        customer.setName("Nouveau");
        customer.setForname("client");
        bank.getCustomers().add(customer);
        String summary = "Client ajouté.";
        String detail = "(" + customer.getName() + ")";
        addMessage(null,summary,detail,FacesMessage.SEVERITY_INFO);
    }


      public void addMessage(String id,String summary,String
```

```
detail,FacesMessage.Severity severity){
        FacesContext facesContext = FacesContext.getCurrentInstance();
        FacesMessage facesMessage = new FacesMessage();
        facesMessage.setSeverity(severity);
        facesMessage.setSummary(summary);
        facesMessage.setDetail(detail);
        facesContext.addMessage(id, facesMessage);
    }
}
```



Nous verrons dans les chapitres suivants comment utiliser le tag `<html:message>`.

4 Les converters

Les converters concernent les composants qui publie une de valeur. Ces composants implémentent les interfaces `ValueHolder`. Par exemple, `UIOutput` est le composant ancêtre de ce type de composants.

Les converters sont également utilisés lors de la saisie de valeurs.

4.1 Principes

La conversion se déclenche lors de la phase `ProcessValidation`. Tout d'abord JSF récupère la valeur soumise par le client et la stocke dans l'atrribut `submittedValue` (interface `EditableValueHolder`).

Ensuite, cette valeur qui est une chaîne de caractères (HTTP ne connaît que ce type de données) est convertie dans le type adéquat puis stockée dans l'attribut `localValue` du composant.

JSF choisit un converter par défaut lorsque la valeur du modèle est un type primitif (integer, float, ..,).

Cependant, pour affiner cette conversion ou pour les types non primitifs il faut spécifier un converter.

4.2 Les converters standards

JSF fournit deux converters standards : le premier pour les nombres et le deuxième pour les dates.

Pour utiliser ces converters, il faut importer la librairie de tags core.

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="core" %>
```

Ensuite, les deux converters sont manipulables par le biais de deux tags : `convertNumber` et `convertDate`.

Reprenons notre exemple de formulaire de saisie (chapitre 5 et 6 de Introduction à JSF):

```
<%@ page contentType="text/html" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="html" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="core" %>
<core:view>
    <html:form>
        ...
        <html:outputText value="Total : "/>
        <html:outputText value="#{accountDatas.total}">
            <core:convertNumber type="currency"
                    currencyCode="EUR" minFractionDigits="3"/>
        </html:outputText>
        <BR/>
        <html:outputText value="Montant :"/>
        <html:inputText id="amount" value="#{accountDatas.amount}">
            <core:convertNumber integerOnly="true"/>
```

```
        </html:inputText>
    ...
</core:view>
```

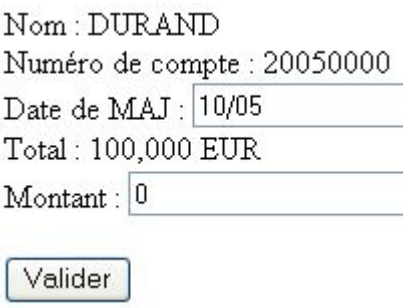
Le tag spécifie quel est le converter que JSF utilisera lors de l'étape de conversion. Ce tag s'insère dans un tag qui construit un composant de type `ValueHolder`. Insérer un tag converter dans un tag qui ne remplit pas cette contrainte n'a aucun sens et provoquera une erreur.

Ajoutons maintenant un attribut de type `Date` dans le bean puis un converter dans la page.

```
package com.facestut.bean;

import java.util.Date;

public class AccountDatas {
...
    // Ne pas oublier le get et le set
    private Date lastModified = new Date();
...
```

Puis :

```
<%@ page contentType="text/html" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="html" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="core" %>
<core:view>
    <html:form>
        <BR/>
        <html:outputText value="Nom : #{accountDatas.customer.name}"/>
        <BR/>
        <html:outputText value="Numéro de compte : #{accountDatas.number}"/>
        <BR/>
        <html:outputText value="Date de MAJ : "/>
        <html:outputText value="#{accountDatas.lastModified}">
            <core:convertDateTime pattern="MM/yy"/>
        </html:outputText>
        <BR/>
        <html:outputText value="Total : "/>
      ...
```



Ces exemples sont valables en saisie.

Lorsque JSF ne peut convertir la valeur saisie par l'utilisateur alors la page courante est affiché de nouveau. En fait, JSF alors en phase `ProcessValidation`, saute directement à la phase `RenderResponse`.

Comme la phase `UpdateModelValues` est sautée le modèle n'est pas modifié. Cela empêche la mise à jour du modèle avec des données non validées.

Pour que l'utilisateur comprenne il est utile d'afficher le message d'erreur lancé par JSF.

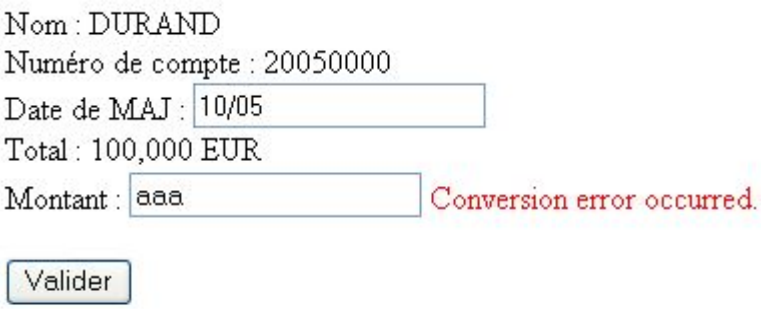
Reprenons notre exemple et saisissons une suite de lettre en lieu et place du montant. Cliquons sur « Valider ». JSF rafraîchit la page mais rien n'indique que quelque chose s'est mal déroulé.

Nous allons utiliser le tag `message` pour afficher l'erreur lié au champs « Montant » :

```
...
  <BR/>
  <html:outputText value="Montant :"/>
  <html:inputText id="amount" value="#{accountDatas.amount}">
      <core:convertNumber integerOnly="true"/>
  </html:inputText>
  <html:message for="amount"/>
...
```

Ainsi, le message d'erreur indique qu'il y eu une erreur de conversion. Par défaut, JSF utilise la locale anglaise.

Pour changer il faut la spécifier la locale dans le *faces-config.xml*. (voir la dtd et les codes des locales)



4.3 Développer un converter

Un converter opérationnel est une classe qui implémente l'interface `javax.faces.Converter`.

Cette interface possède deux méthodes :

- Object **getAsObject**(FacesContext,UIComponent component,String) : convertit la valeur soumise par le client (paramètre String) en Object
- String **getAsString**(FacesContext,UIComponent component,Object) : conevrtit l'objet du modèle en String

La première est appelée dans la phase `ProcessValidation` et la deuxième dans la phase `RenderResponse`.

Nous allons maintenant développer un converter custom.

Changeons un peu le fonctionnel de notre application. Un numéro de compte devra maintenant avoir la structure suivante :

CODEPAYS_CODEAGENCE_NUMEROCLIENT

CODEPAYS : 3 caractères

CODEAGENCE: 5 chiffres

NUMEROCLIENT : 6 chiffres

_ : un caractère d'espacement

Toute valeur qui est structuré de la sorte est un numéro de compte.

Exemple : FRA 00005 123456

Développer un convertisseur de numéro de compte va donc consister à créer une classe qui implémente l'interface Converter.

Changeons notre bean :

```
...
public class AccountDatas {

    // N'oubliez pas le get et le set
    private String number = "FRA 00005 123456";
...
```

Puis notre page :

```
<%@ page contentType="text/html" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="html" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="core" %>
<core:view>
    <html:form>
        <BR/>
        <html:outputText value="Nom : #{accountDatas.customer.name}"/>
        <BR/>
        <html:outputText value="Numéro de compte :"/>
        <html:inputText id="number" value="#{accountDatas.number}">
            <core:converter converterId="com.facestut.AccountNumber"/>
        </html:inputText>
        <html:message for="number" style="color:red;" showDetail="true"/>
        <BR/>
        <html:outputText value="Date de MAJ : "/>
...
```

Vous remarquerez l'usage du tag `converter` qui permet d'utiliser un converter custom grâce à l'attribut `converterId`.

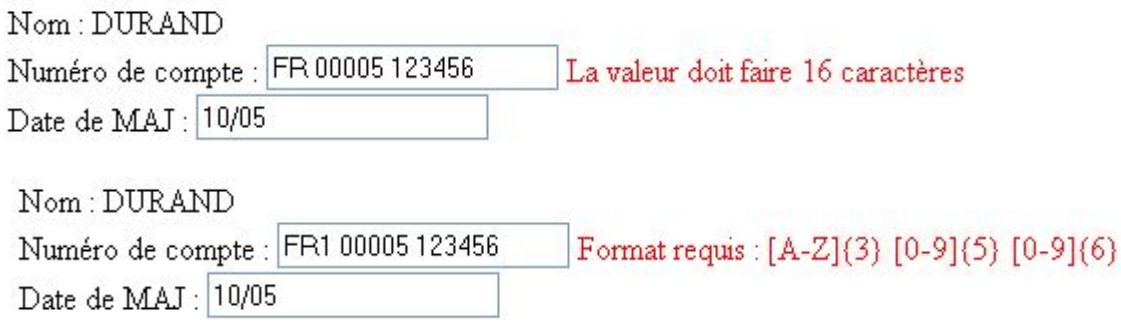
Nous devons déclarer le converter dans le fichier *faces-config.xml* :

```
<?xml version="1.0"?>
<!--
 Copyright 2003 Sun Microsystems, Inc. All rights reserved.
 SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
-->
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
...
  <converter>
    <converter-id>com.facestut.AccountNumber</converter-id>
    <converter-class>com.facestut.converter.AccountNumberConverter</converter-class>
  </converter>
...
```

Enfin, la classe du converter : AccountNumberConverter.java

```
package com.facestut.converter;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;

public class AccountNumberConverter implements Converter {

    public AccountNumberConverter() {
    }

    public String getAsString(FacesContext facesContext, UIComponent uIComponent, Object obj) {
        return obj.toString();
    }

    public Object getAsObject(FacesContext facesContext, UIComponent uIComponent, String str) {
        if(str.length() == 16){
            String regex = "[A-Z]{3} [0-9]{5} [0-9]{6}";
            if(str.matches(regex)){
                return str;
            } else {
                FacesMessage facesMessage = new FacesMessage();
                facesMessage.setSeverity(FacesMessage.SEVERITY_ERROR);
                facesMessage.setSummary("Format incorrect");
                facesMessage.setDetail("Format requis : [A-Z]{3} [0-9]{5} [0-9]{6}");
                throw new ConverterException(facesMessage);
            }
        } else {
            FacesMessage facesMessage = new FacesMessage();
            facesMessage.setSeverity(FacesMessage.SEVERITY_ERROR);
            facesMessage.setSummary("Format incorrect");
            facesMessage.setDetail("16 caractères attendus");
            throw new ConverterException(facesMessage);
        }
    }
}
```

Notez bien que dans un contexte internationalisé, les messages ne doivent pas être en « dur » dans le code mais dans un bundle prévu pour chaque langue.



4.4 Passer outre les erreurs de conversions

Dans certains cas, il est utile de passer outre les erreurs de conversions.

En effet, imaginons que vous vouliez ajouter un bouton « Annuler » sur votre formulaire. Le comportement standard empêchera l'utilisateur de quitter la page si il existe un erreur de conversion dans la page.

Pour outre passer les erreurs de conversions : l'attribut `immediate` positionné à `true` sur un tag de commande.

```
...
        <BR/> <BR/>
        <html:commandButton value="Valider" action="#{accountDatas.validate}"/>
        <html:commandButton value="Annuler" action="Index" immediate="true"/>
        <BR/>
    </html:form>

</core:view>
```

L'attribut `immediate` modifie quelque peu le cycle de traitement de la requête. JSF va exécuter la phase `RenderResponse` puis sauter à la phase `RenderView`. Ainsi, la phase `ProcessValidation` n'est pas exécutée et donc aucune erreur de saisie ne peut bloquer l'utilisateur.

Remarquez également que la phase `UpdateModel` n'est pas exécutée : le modèle n'est pas modifié.

Modifier le fichier *faces-config.xml* pour ajouter la règle de navigation suivante :

```
  <navigation-rule>
    <from-view-id>*</from-view-id>

    <navigation-case>
        <from-outcome>Index</from-outcome>
        <to-view-id>/index.jsp</to-view-id>
    </navigation-case>

  </navigation-rule>
```

5 Les validators

Les validators concernent les composants qui permettent la saisie de valeurs. Ces composants implémentent les interfaces `ValueHolder` et `EditableValueHolder`. Par exemple, `UIInput` est le composant ancêtre de tous les composants de saisie.

5.1 Principes

La validation se déclenche lors de la phase `ProcessValidation`. Tout d'abord JSF récupère la valeur soumise par le client et la stocke dans l'atrribut `submittedValue` (interface `EditableValueHolder`).

Ensuite, cette valeur qui est une chaîne de caractères (HTTP ne connaît que ce type de données) est convertie dans le type adéquat. (voir chapitre 4). La validation peut commencer.

Le but d'un validator est de protéger le modèle. En effet, la phase `UpdateModel` n'est atteinte que si la donnée est valide, ce qui évite de positionner le modèle dans un état incohérent.

5.2 Les validators standards

JSF est fourni avec trois validators standards : le premier valide la longueur d'une chaîne de caractères, les deux autres valident des numériques.

Pour utiliser ces validators, il faut importer la librairie de tags core.

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="core" %>
```

Ensuite, les trois converters sont manipulables par le biais de trois tags : `validateLength`, `validateLongRange`. `validateDoubleRange`.

Reprenons notre exemple de formulaire de saisie (chapitre 5 et 6 de Introduction à JSF):

```
<%@ page contentType="text/html" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="html" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="core" %>
<core:view>
    <html:form>
        <BR/>
        <html:outputText value="Nom : #{accountDatas.customer.name}"/>
        <BR/>
        <html:outputText value="Numéro de compte :"/>
        <html:inputText id="number" value="#{accountDatas.number}">
            <core:validateLength minimum="16" maximum="16"/>
        </html:inputText>
        <html:message for="number" style="color:red;" showDetail="true"/>
```

```
        <BR/>
        <html:outputText value="Date de MAJ : "/>
        <html:inputText value="#{accountDatas.lastModified}"/>
        <BR/>
        <html:outputText value="Total : "/>
        <html:outputText value="#{accountDatas.total}"/>
        <BR/>
        <html:outputText value="Montant :"/>
        <html:inputText id="amount" value="#{accountDatas.amount}">
            <core:validateDoubleRange minimum="0" maximum="9999"/>
        </html:inputText>
        <html:message for="amount" style="color:red;"/>
...
</core:view>
```

Lorsque la valeur saisie est invalide , on observe le même comportement que pour les converters.

5.3 Développer un validator

Le développement d'un validator ressemble fortement à celui d'un converter (voir chapitre précédent).

Ici un validator est une classe qui implémente l'interface `javax.faces.Validator`.

Cette interface possède une méthode : `void` **`validate`**`(FacesContext,UIComponent,Object)`.

Le paramètre `UIComponent` représente le composant JSF qui détient la donnée à valider.

Le paramètre `Object` représente la donnée qui est validée.

Reprenons le fonctionnel du chapitre 4.3 et ajoutons un validator qui va vérifier que le numéro de compte est bien valide. Nous ajouterons que le numéro client est un nombre impair.

Notre page de formulaire va combiner le converter custom du chapitre 4.3 et notre validator:

```
<%@ page contentType="text/html" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="html" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="core" %>
<core:view>

    <html:form>
        <BR/>
        <html:outputText value="Nom : #{accountDatas.customer.name}"/>
        <BR/>
        <html:outputText value="Numéro de compte :"/>
        <html:inputText id="number" value="#{accountDatas.number}">
            <core:converter converterId="com.facestut.AccountNumber"/>
            <core:validator validatorId="com.facestut.AccountNumber"/>
```

```
        </html:inputText>
        <html:message for="number" style="color:red;" showDetail="true"/>
        <BR/>
...
```

Vous remarquerez l'usage du tag `validator` qui permet d'utiliser un validator custom grâce à l'attribut `validatorId`.

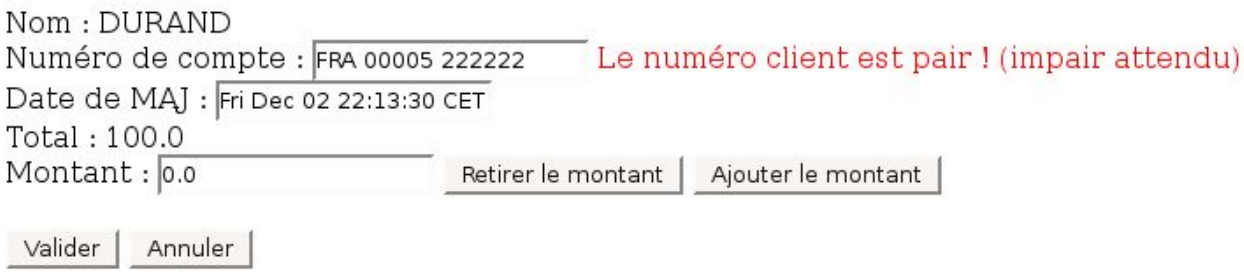
Nous devons déclarer le validator dans le fichier *faces-config.xml* :

```
<?xml version="1.0"?>

<!--
 Copyright 2003 Sun Microsystems, Inc. All rights reserved.
 SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
-->

<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
...
  <validator>
    <validator-id>com.facestut.AccountNumber</validator-id>
    <validator-class>com.facestut.validator.AccountNumberValidator</validator-class>
  </validator>
```

Puis enfin, notre classe AccountNumberValidator :

```
package com.facestut.validator;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

public class AccountNumberValidator implements Validator {

    public AccountNumberValidator(){
    }
```

```
    public void validate(FacesContext facesContext,UIComponent component,Object value){

        String accountNumber  = (String)value;
        String[] parts = accountNumber.split(" ");
        String customerNumberAsString = parts[2];
        long customerNumber = Long.valueOf(customerNumberAsString).longValue();
        boolean even = ((customerNumber / 2)*2) == customerNumber;
        if(even){
            FacesMessage facesMessage = new FacesMessage();
            facesMessage.setSeverity(FacesMessage.SEVERITY_ERROR);
            facesMessage.setSummary("Le numéro client est pair ! (impair attendu)");
            throw new ValidatorException(facesMessage);
        }
    }
}
```



5.4 Passer outre les erreurs de validations

Voir le chapitre 4.4.

6 Développer un composant JSF

6.1 Principes

Un composant JSF consiste en trois éléments dont un est optionnel :

- la classe du tag : définit une classe qui va faire le lien entre la page JSP et le composant.
- la classe du composant : implémente le composant aux travers ses attributs et méthodes.
- la classe du renderer : optionnelle, cette classe est spécialisée dans le rendu du composant (HTML,XML,XUL, ...)

JSF établit des conventions de nommage pour ces classes. Ainsi, la classe du tag aura pour nom `<nomducomposant>Tag`, la classe du composant `UI<nomducomposant>` et la classe du renderer `<nomducomposant>Renderer`.

6.2 Le composant UIPager

Nous avons vu comment construire des listes. Nous voulons maintenant paginer nos listes.

Le composant `UIPager` aura les caractéristiques suivantes :

- un contrôle page suivante
- un contrôle page précédente
- le nombre d'éléments dans la liste
- paramétrable (nombre d'éléments par page)
- composant fils d'un `UIData` via le tag `dataTable` et le tag `facet` (header ou footer)

Commençons par la classe du composant `UIPager` :

```
package com.facestut.component;

import java.io.IOException;
import javax.faces.component.UIComponentBase;
import javax.faces.component.UIData;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;

public class UIPager extends UIComponentBase {
```

```
    public String getFamily(){
        return "facestut";
    }

    public void encodeBegin(FacesContext facesContext) throws IOException {
        ResponseWriter responseWriter = facesContext.getResponseWriter();

        // Encode le contrôle page précédente
        responseWriter.startElement("a", this);
        responseWriter.writeAttribute("href","#","href");
        responseWriter.write("<<");
        responseWriter.endElement("a");
    }

    public void encodeChildren(FacesContext facesContext) throws IOException {
    }

    public void encodeEnd(FacesContext facesContext) throws IOException {
        ResponseWriter responseWriter = facesContext.getResponseWriter();

        // Encode le contrôle page suivante
        responseWriter.startElement("a", this);
        responseWriter.writeAttribute("href","#","href");
        responseWriter.write(">>");
        responseWriter.endElement("a");

        // Encode les informations de taille
        UIData uiData  = getUIData();
        responseWriter.startElement("p", this);
        responseWriter.write(uiData.getRowCount() + " éléments." );
        responseWriter.endElement("p");

    }

    public boolean getRendersChildren(){
        return true;
    }

    protected UIData getUIData(){
        return (UIData) this.getParent();
    }
}
```

Notre composant hérite de `UIComponentBase`. Cette classe est la classe de base pour créer un

composant JSF. Notez que `UIComponentBase` a des classes filles qui modélise des types de composants plus spécifiques : *`UIPanel`*, *`UICommand`*, ...

Hériter de `UIComponentBase` implique l'implémentation de la méthode `getFamily()`.

Cette méthode renvoie la famille de votre composant. Une famille permet de regrouper des composants. Cette notion est utilisée en conjonction avec le concept de Renderer que nous verrons plus tard.

Un composant JSF est responsable d'un certain nombre de tâches dont :

- effectuer son rendu auprès de l'utilisateur ou encodage
- se décoder pour changer son état lorsque l'utilisateur interagit avec lui côté client
- sauvegarder son état
- restaurer son état
- ...

La classe `UIComponentBase` implémente différentes méthodes qui sont directement liées au cycle de traitement d'une requête JSF (voir chapitre 2). Dans notre cas, nous ne modifierons pas le comportement de ces méthodes.

Les méthodes d'encodage vont donc se charger du rendu de notre composant. La présence de trois méthodes encode est liée à l'utilisation de langage à balises pour générer le rendu. En effet, tous les langages à balises sont issus de XML. Une balise XML à un début une fin et éventuellement contient des balises filles. Nous avons la correspondance suivante :

<xxx>: `encodeBegin`

`encodeChildren` :

<yyy></yyyy>

<yyy></yyyy>

.....

</xxx>: `encodeEnd`

Notez bien que ce découpage est théorique.

Dans certains cas un composant ne rend pas ces fils, la méthode `encodeChildren` n'est pas implémentée et `getRenderChildren` retourne `false`.

Dans d'autres cas, la notation abrégée d'une balise est utilisée car le composant ne peut avoir de fils, `encodeBegin` n'est pas implémentée et tout le code de rendu est dans `encodeEnd` (<xxx />).

Notre composant n'est pas terminé mais nous pouvons tout de même tester son rendu grâce au tag `pager`.

Avant cela nous devons indiquer à JSF qu'un nouveau composant est disponible. Le fichier de configuration f*aces-config.xml* va nous permettre cela :

```
<?xml version="1.0"?>
<!--
 Copyright 2003 Sun Microsystems, Inc. All rights reserved.
 SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
-->
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>

...
  <managed-bean>
    <managed-bean-name>accountDatas</managed-bean-name>
    <managed-bean-class>com.facestut.bean.AccountDatas</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
...
  <converter>
    <converter-id>com.facestut.AccountNumber</converter-id>
    <converter-class>com.facestut.converter.AccountNumberConverter</converter-class>
  </converter>
...
  <navigation-rule>

    <!-- Indique pour quelle vue courante la règle s'applique -->
    <from-view-id>/index.jsp</from-view-id>

...
  </navigation-rule>

  <component>
    <component-type>facestut.component.Pager</component-type>
    <component-class>com.facestut.component.UIPager</component-class>
  </component>

</faces-config>
```

6.3 Le tag pager

Un tag JSF permet de créer un composant particulier via une page JSP.

Pour cela nous devons créer une classe `PagerTag` :

```
package com.facestut.tag;

import javax.faces.component.UIComponent;
import javax.faces.webapp.UIComponentTag;

public class PagerTag extends UIComponentTag {



    public String getRendererType(){
        return null;
    }

    public String getComponentType(){
        return "facestut.component.Pager";
    }
}
```

Cette classe étends `UIComponentTag` qui est la classe ancêtre de tout tag JSF. Un tag JSF n'a d'autre but que de créer une instance de composant.

La méthode `getComponentType()` renvoie le type du composant à créer tandis que la méthode `getRendererType()` renvoie le nom du renderer qui effectue le rendu du composant Lorsque cette méthode renvoie `null` le composant effectue lui même son rendu dans les méthodes `encodeXXX`.

Nous devons maintenant créer le fichier */Web Pages/META-INF/facestut.tld* afin de déclarer notre tag :

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd">
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>ft</short-name>
  <uri>http://facestut.developpez.com/jsf</uri>
  <tag>
    <name>pager</name>
    <tag-class>com.facestut.tag.PagerTag</tag-class>
    <body-content>empty</body-content>
  </tag>
</taglib>
```

6.4 Utiliser le composant UIPager

Notre composant et notre tag sont désormais connus grâce aux fichiers de configuration. Nous devons maintenant les utiliser dans la page de liste (chapitre 9 intro JSF data.table-mvc.jsp) :

```
<%@ page contentType="text/html" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="html" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="core" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="core" %>
<%@ taglib uri="/META-INF/facestut.tld" prefix="ft" %>
<core:view>
    <html:form>
        <html:dataTable id="customers"
             binding="#{bankCtrl.view.dataTable}"
             value="#{bankCtrl.model.datas.customers}"
            var="customer" border="1">

            <core:facet name="header">
                <ft:pager/>
            </core:facet>

            <html:column>
                <html:selectBooleanCheckbox binding="#{bankCtrl.view.checkbox}"/>
            </html:column>
            <html:column>
                <core:facet name="header">
                    <core:verbatim>Nom</core:verbatim>
                </core:facet>
                <html:outputText value="#{customer.name}"/>
            </html:column>
            <html:column>
                <core:facet name="header">
                    <core:verbatim>Prénom</core:verbatim>
                </core:facet>
                <html:outputText value="#{customer.forname}"/>
            </html:column>
        </html:dataTable>
        <br>
        <html:commandButton value="Supprimer les clients"
             action="#{bankCtrl.removeSelectedCustomers}"/>
        <html:commandButton value="Ajouter un client"
             action="#{bankCtrl.addCustomer}"/>
    </html:form>
</core:view>
```

Nous obtenons le résultat suivant :

6.5 Ajout d'attributs

Dans les spécifications du composant `UIPager` il est indiqué que le nombre d'éléments par page est paramétrable.

Nous devons ajouter cet attribut dans le composant :

```
package com.facestut.component;

import java.io.IOException;
import javax.faces.component.UIComponentBase;
import javax.faces.component.UIData;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;

public class UIPager extends UIComponentBase {

    private Integer itemsByPage;
....
    public Integer getItemsByPage(){
        return this.itemsByPage;
    }

    public void setItemsByPage(Integer integer){
        this.itemsByPage = integer;
    }

    public Object saveState(FacesContext context) {
        Object values[] = new Object[2];
        values[0] = super.saveState(context);
        values[1] = this.itemsByPage;
        return values;
```

```
    }

   public void restoreState(FacesContext context, Object state) {
        Object values[] = (Object[]) state;
        super.restoreState(context, values[0]);
        this.itemsByPage = (Integer)values[1];
   }
}
```

Le cycle de vie JSF prévoit une phase de sauvegarde et une phase de restauration de l'état de tout composant JSF.

Ces phases appellent les méthodes `saveState` et `restoreState` sur chaque composant de la vue JSF courante (ou `UIViewRoot`). Voir le chapitre 2 à ce sujet.

Ensuite nous devons modifier la classe du tag :

```
package com.facestut.tag;

import javax.faces.component.UIComponent;
import javax.faces.webapp.UIComponentTag;

public class PagerTag extends UIComponentTag {

    private Integer itemsByPage;

    public String getRendererType(){
        return null;
    }

    public String getComponentType(){
        return "facestut.component.Pager";
    }

    public Integer getItemsByPage(){
        return this.itemsByPage;
    }

    public void setItemsByPage(Integer integer){
        this.itemsByPage = integer;
    }

    public void setProperties(UIComponent component){
        super.setProperties(component);
        if(getItemsByPage() != null){
            component.getAttributes().put("itemsByPage",getItemsByPage());
        }
    }
}
```

La méthode `setProperties(UIComponent)` est une méthode de `UIComponentTag`. Elle a pour responsabilité de mettre à jour les attributs du composant nouvellement crée. A cet effet, la classe `UIComponent` offre un mécanisme intéressant via une `Map` d'attributs.

Lorsque vous invoquez la méthode `put(String,Object)` sur celle-ci, deux comportements sont possibles:

- Soit la classe du composant possède un attribut `itemsByPage` avec ces getter et setter : la méthode `put` mets à jour cet attribut via le setter
- Soit l'attribut n'existe pas dans la classe et la `Map` se comporte comme une `Map` normale.

Remarque : on observe le même algorithme pour la lecture de l'attribut mais la méthode `get(String)`.

Il nous faut également déclarer l'attribut `itemsByPage` dans la TLD :

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd">
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>ft</short-name>
  <uri>http://facestut.developpez.com/jsf</uri>

   <tag>
    <name>pager</name>
    <tag-class>com.facestut.tag.PagerTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
        <name>itemsByPage</name>
        <type>java.lang.Integer</type>
    </attribute>
   </tag>

</taglib>
```

Nous pouvons désormais utiliser cet attribut dans nos pages.

Cependant, l'attribut `itemsByPage` est mis à jour mais nous n'avons pas effectué de traitement spécifique pour indiquer au composant `UIData` de ne rendre qu'une partie de la liste d'objets.

```
package com.facestut.tag;

import com.facestut.component.UIPager;
import javax.faces.component.UIComponent;
import javax.faces.component.UIData;
import javax.faces.webapp.UIComponentTag;
import javax.servlet.jsp.JspException;

public class PagerTag extends UIComponentTag {
...
    public int doEndTag() throws JspException {
        if(getCreated()){
           UIPager uiPager = (UIPager)getComponentInstance();
           UIComponent uiComponent = uiPager.getParent();
           if(uiComponent instanceof UIData){
               UIData uiData = (UIData)uiComponent;
```

```
                if(getItemsByPage() != null){
                 int rows = getItemsByPage().intValue();
                 uiData.setRows(rows);
                }
            }
            else {
               throw new JspException("Le parent du pager n'est pas un htmlDataTable !");
            }
        }
        return super.doEndTag();
    }
}
```

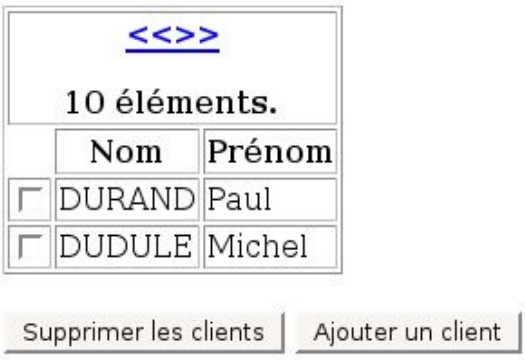
La méthode `doEndTag()` est appelée lorsque la balise de fin de tag est rencontrée. Il est souvent utile de redéfinir cette méthode pour effectuer des traitements.

Ici, nous utilisons la méthode `getCreated()` pour savoir si le composant `UIPager` vient d'être créer ou si il existe déjà.

En effet, lorsque vous rechargez une page JSF, la méthode `setProperties(UIComponent)` du tag n'est plus invoquée à moins que la page JSP ait changé. Cette méthode est invoquée lorsque le composant est créé pour la première fois. La méthode `getCreated()` nous donne cette information.

Ici, nous allons contrôler que le composant parent est bien un `UIData` et changer l'attribut `rows` afin de limiter le nombre d'objets visibles. Notez bien que `UIData` ne gère pas la notion de page.

Remarque importante : lorsque le composant est créée et initialisé dans `setProperties(UIComponent)`, il n'est pas encore inséré dans l'arbre de composants, ce qui veut dire que sa méthode `getParent()` renvoie `null`.



6.6 Gestion des actions de l'utilisateur

Notre composant commence à prendre forme, néanmoins il manque l'essentiel : prendre en compte les actions de l'utilisateur.

Les méthodes `encodeXXX` nous permettent de rendre l'instance du composant.

La méthode `decode (FacesContext)` est utilisée pour le décodage des informations contenues dans la requête HTTP envoyée par le navigateur lors d'une action de l'utilisateur sur le pager.

Cependant, au vu de la méthode `encodeEnd(FacesContext)` de `UIPager` , le click de l'utilisateur sur page suivante ou page précédente ne sera pas détecté.

Nous devons modifier ce code pour faire en sorte que nous sachions si l'action de l'utilisateur concerne l'instance du composant `UIPager`.

Tout d'abord les méthodes `encodeBegin` et `encodeEnd` :

```
...
    public void encodeBegin(FacesContext facesContext) throws IOException {
        ResponseWriter responseWriter = facesContext.getResponseWriter();
        String clientId = getClientId(facesContext);
        // Encode le contrôle page précédente
        responseWriter.startElement("a", this);
        responseWriter.writeAttribute("href","javascript:document.forms[0].submit();
","href");
        responseWriter.writeAttribute("onclick","document.getElementById('" + clientId
+"').value = 'prev';return true;","onclick");
        responseWriter.write("<<");
        responseWriter.endElement("a");
    }

    public void encodeChildren(FacesContext facesContext) throws IOException {

    }

    public void encodeEnd(FacesContext facesContext) throws IOException {
        ResponseWriter responseWriter = facesContext.getResponseWriter();

        String clientId = getClientId(facesContext);

        // Encode le contrôle page suivante
        responseWriter.startElement("a", this);
        responseWriter.writeAttribute("href","javascript:document.forms[0].submit();
","href");
        responseWriter.writeAttribute("onclick","document.getElementById('" + clientId
+"').value = 'next';return true;","onclick");
        responseWriter.write(">>");
        responseWriter.endElement("a");
```

```
        // Encode les informations de taille
        UIData uiData  = getUIData();
        responseWriter.startElement("p", this);
        responseWriter.write(uiData.getRowCount() + " éléments." );
        responseWriter.endElement("p");

        responseWriter.startElement("input", this);
        responseWriter.writeAttribute("id",clientId,"id");
        responseWriter.writeAttribute("name",clientId,"name");
        responseWriter.writeAttribute("type","hidden","type");
        responseWriter.writeAttribute("value","","value");

    }
...
```

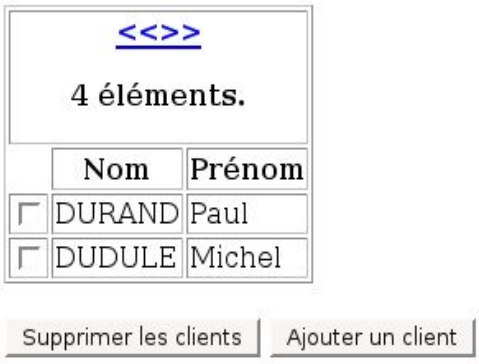
Nous utilisons ici un champs caché dont le nom est le `clientId` du composant `UIPager` courant.

En effet, le client id a pour rôle d'identifier de manière unique une instance de composant qui est rendu côté client. Lorsque l'utilisateur clique sur suivant ou précédent un bout de code Javascript mets à jour le champs qui représente le composant et son état. Le formulaire est ensuite soumis.

Nous pouvons maintenant décoder les paramètres de la requête HTTP afin d'y retrouver notre champs caché et sa valeur :

```
    public void decode(FacesContext facesContext){
        String clientId = getClientId(facesContext);
        Map parametersMap = facesContext.getExternalContext().getRequestParameterMap();
        Object value = parametersMap.get(clientId);
        if(value != null){
            String cmd = (String)value;
            UIData uiData = (UIData) getParent();
            if(itemsByPage!= null){
                if(cmd.equals("prev")){
                    int computedFirst = uiData.getFirst()-itemsByPage.intValue();
                    if(computedFirst < 0){
                        computedFirst = 0;
                    }
                    uiData.setFirst(computedFirst);
                }
                else if(cmd.equals("next")){
                    int computedFirst = uiData.getFirst()+itemsByPage.intValue();
                    if(computedFirst >= uiData.getRowCount()){
                        computedFirst = uiData.getFirst();
                    }
                    uiData.setFirst(computedFirst);
```

```
                }
            }
        }
    }
```



Le composant `UIPager` n'est pas très user friendly: il pourrait afficher par exemple la page courante ou une liste déroulante contenant les numéros de page. A vous de jouer !

6.7 Le renderer

Un renderer est une classe spécialisée dans l'encodage et le décodage d'un type de composant particulier.

En effet, le composant JSF définit le composant en dehors de toute représentation. Cependant, la présence des méthodes d'encodage et de décodage brise cette indépendance.

Dans le cas de `UIPager` ces méthodes contiennent du code HTML et un algorithme de décodage particulier. Il est possible de déléguer ces responsabilités à un renderer.

Un renderer est une classe Java qui étends la classe `javax.faces.Renderer`. Voici la classe PagerRenderer :

```
package com.facestut.renderer;

import com.facestut.component.UIPager;
import java.io.IOException;
import java.util.Map;
import javax.faces.component.UIComponent;
import javax.faces.component.UIData;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.render.Renderer;

public class PagerRenderer extends Renderer {

    public void encodeBegin(FacesContext facesContext,UIComponent component) throws IOException {
```

```
        ResponseWriter responseWriter = facesContext.getResponseWriter();
        String clientId = component.getClientId(facesContext);
        responseWriter.startElement("a", component);
        responseWriter.writeAttribute("href","javascript:document.forms[0].submit();
","href");
        responseWriter.writeAttribute("onclick","document.getElementById('" + clientId
+"').value = 'prev';return true;","onclick");
        responseWriter.write("<<");
        responseWriter.endElement("a");
    }


    public void encodeEnd(FacesContext facesContext,UIComponent component) throws IOException {

        ResponseWriter responseWriter = facesContext.getResponseWriter();
        String clientId = component.getClientId(facesContext);

        responseWriter.startElement("a", component);
        responseWriter.writeAttribute("href","javascript:document.forms[0].submit();
","href");
        responseWriter.writeAttribute("onclick","document.getElementById('" + clientId
+"').value = 'next';return true;","onclick");
        responseWriter.write(">>");
        responseWriter.endElement("a");

        UIPager uiPager = (UIPager)component;
        UIData uiData  = uiPager.getUIData();
        responseWriter.startElement("p", component);
        responseWriter.write(uiData.getRowCount() + " Ã©lÃ©ments." );
        responseWriter.endElement("p");

        responseWriter.startElement("input", component);
        responseWriter.writeAttribute("id",clientId,"id");
        responseWriter.writeAttribute("name",clientId,"name");
        responseWriter.writeAttribute("type","hidden","type");
        responseWriter.writeAttribute("value","","value");

    }

    public void decode(FacesContext facesContext,UIComponent component){
        String clientId = component.getClientId(facesContext);
        Map parametersMap = facesContext.getExternalContext().getRequestParameterMap();
        Object value = parametersMap.get(clientId);
        if(value != null){
            String cmd = (String)value;
            UIData uiData = (UIData) component.getParent();
```

```
            Integer itemsByPage = (Integer)component.getAttributes().get("itemsByPage");
            if(itemsByPage!= null){
                if(cmd.equals("prev")){
                    int computedFirst = uiData.getFirst()-itemsByPage.intValue();
                    if(computedFirst < 0){
                        computedFirst = 0;
                    }
                    uiData.setFirst(computedFirst);
                }
                else if(cmd.equals("next")){
                    int computedFirst = uiData.getFirst()+itemsByPage.intValue();
                    if(computedFirst >= uiData.getRowCount()){
                        computedFirst = uiData.getFirst();
                    }
                    uiData.setFirst(computedFirst);
                }
            }
        }
    }
}
```

L'intérêt du renderer est de pouvoir déléguer des responsabilités du composant à une classe spécialisée pour une configuration particulière.

En effet, le renderer d'un composant est valable dans une configuration donnée ce qui permet de changer de renderer en fonction de la technologie de rendu utilisée (WAP, HTML, XUL, ...).

L'association d'un renderer est d'un composant se fait au niveau du tag. La méthode getRendererType() indique le type de renderer utilisé :

```
...
public class PagerTag extends UIComponentTag {

    private Integer itemsByPage;

    public String getRendererType(){
        return "facestut.renderer.Pager";
    }

    public String getComponentType(){
        return "facestut.component.Pager";
    }
    ...
```

La correspondance entre le type et la classe du renderer est réalisée par *faces-config.xml* :

```
...
  <component>
    <component-type>facestut.component.Pager</component-type>
    <component-class>com.facestut.component.UIPager</component-class>
  </component>

  <render-kit>

    <renderer>
        <component-family>facestut</component-family>
        <renderer-type>facestut.renderer.Pager</renderer-type>
        <renderer-class>com.facestut.renderer.PagerRenderer</renderer-class>
    </renderer>

  </render-kit>

</faces-config>
```

La balise `render-kit` définit un kit de rendu. Un kit de rendu est un ensemble de renderer.

Ainsi vous pouvez changer le rendu d'un composant en changeant son renderer. Ici, il suffit de changer la classe `com.facestut.renderer.PagerRenderer` en un autre nom de classe.

7 Conclusion

Nous avons vu dans ce tutoriel comment aborder le développement d'une application JSF avec des concepts nouveaux, pour un résultat plus professionnel :

- la validation et la conversion de la saisie utilisateur
- les messages
- les composants customs

Cependant, JSF propose des fonctionnalités encore plus avancées qui permettent de customiser son fonctionnement même, comme par exemple :

- Utiliser XML pour construire les vues
- Utiliser un gestionnaire de navigation custom pour par exemple produire des applications type workflow
- ...

Et JSF 1.2 qui sortira au printemps 2006 apporte son lot de nouveautés ...