Introduction à JSF

Avec Netbeans 4.1

oschmitt@free.fr

Août 2005

Table des matières

1 Qu'est ce que JSF ?

Java **S**erver **F**aces est un framework d'interface utilisateur pour les applications webs, basé sur les technologies JSP et Servlets.

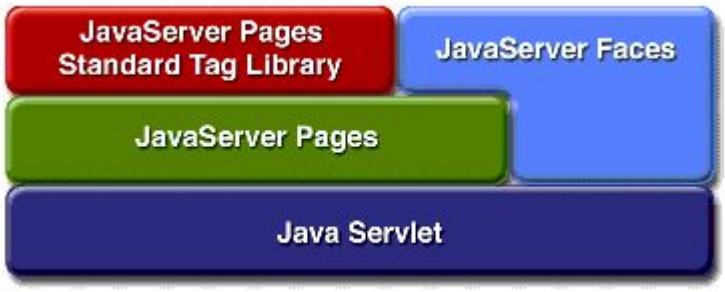
1.1 Pourquoi JSF ?

Le but de JSF est d'accroître la productivité des développeurs dans le développement des interfaces utilisateur tout en facilitant leur maintenance.

1.2 Quelle place dans Java/J2EE ?

JSF est le fruit de la communauté Java via le JCP. Le développement de JSF suit donc la même procédure que les autres technologies Java comme JSP, Servlets, EJB, ...

Cette procédure consiste pour une version donnée de la technologie, en une phase de spécification puis une phase d'implémentation quasiment parallèle. JSF fait partie de J2EE 1.4.



1.3 Pourquoi utiliser JSF ?

JSF est un **standard J2EE**.
Le support de JSF par les éditeurs J2EE est obligatoire. Actuellement, les plus grands éditeurs Java annoncent ou proposent une intégration de JSF dans leurs IDEs.

JSF permet :

- une séparation nette entre la couche de présentation et les autres couches
- le **mapping HTML/Objet**
- un **modèle riche de composants graphiques** réutilisables
- une gestion de l'état de l'interface entre les différentes requêtes
- une liaison simple entre les actions côté client de l'utilisateur et le code Java correspondant côté serveur

- la **création de composants customs** grâce à une API
- le **support de différents clients** (HTML, WML, XML, ...) grâce à la séparation des problématiques de construction de l'interface et du rendu de cette interface

Il existe plusieurs frameworks webs Java dédiés au développement d'interfaces utilisateur mais aucun n'est un standard et va aussi loin que JSF.

Il bénéficie de concepts déjà éprouvés par Java 2 et J2EE (composants graphiques Swing, modèle événementiel, JSP, Servlets) et des apports de Struts dont le concepteur, Craig Mac Clanahan, est aussi le co leader et principal développeur de JSF.

2 Installer l'environnement de développement

2.1 Choix de l'implémentation

Il existe plusieurs implémentations de JSF. Nous utiliserons l'implémentation de référence de SUN.

Téléchargez l'implémentation sur le site dédié à JSF: http://java.sun.com/j2ee/javaserverfaces/index.jsp

Téléchargez la JSTL 1.0 : http://archive.apache.org/dist/jakarta/taglibs/standard-1.0/

2.2 Installer netBeans 4.1

Nous utiliserons netBeans 4.1 pour ce tutorial.

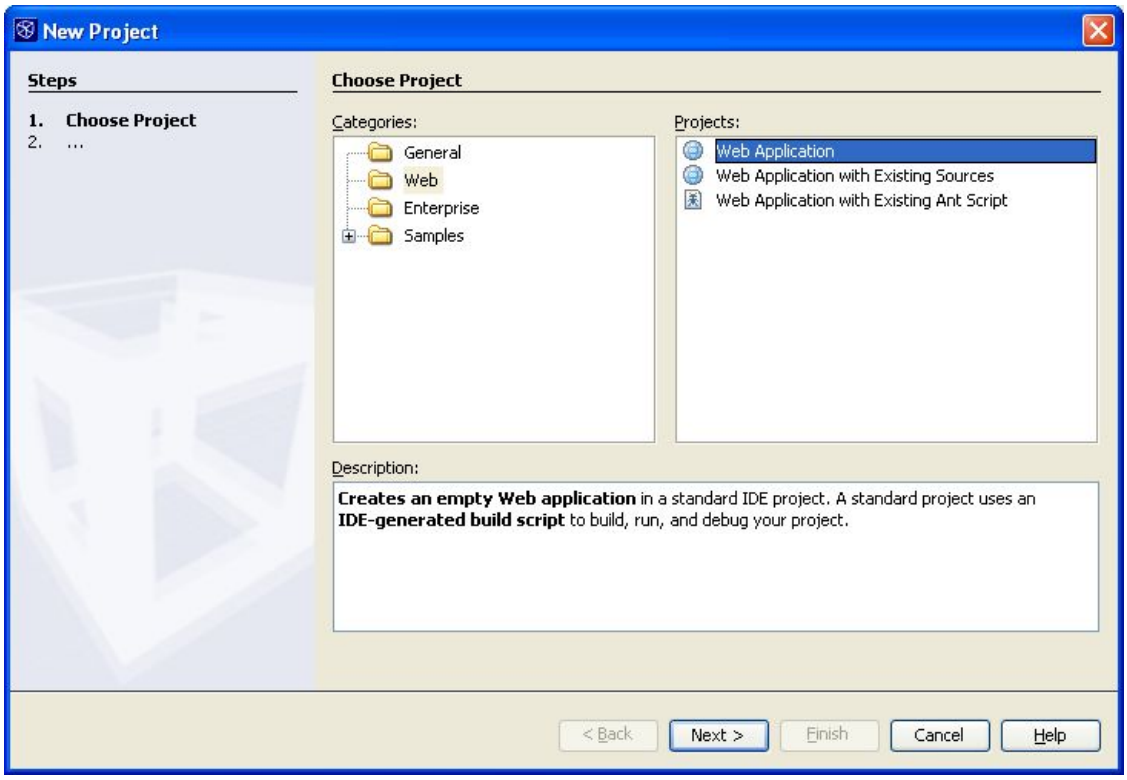
Téléchargez netBeans 4.1 : http://www.netbeans.org/

2.3 Création du projet

Je vous propose la structure suivante :

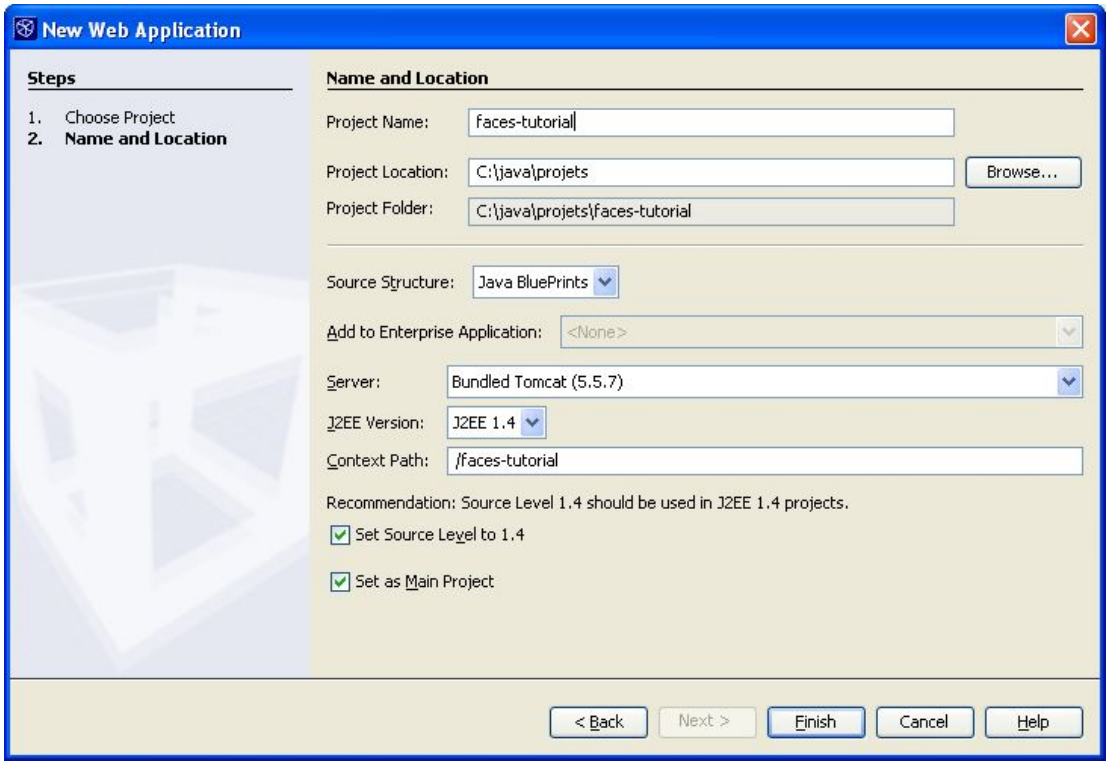
netBeans propose de nombreux assistants de création de projets.

Exécuter l'action : *File* > *New Project*



Choisissez *Web Application* puis *Next*.

Saisissez les informations dans l'assistant comme suit :

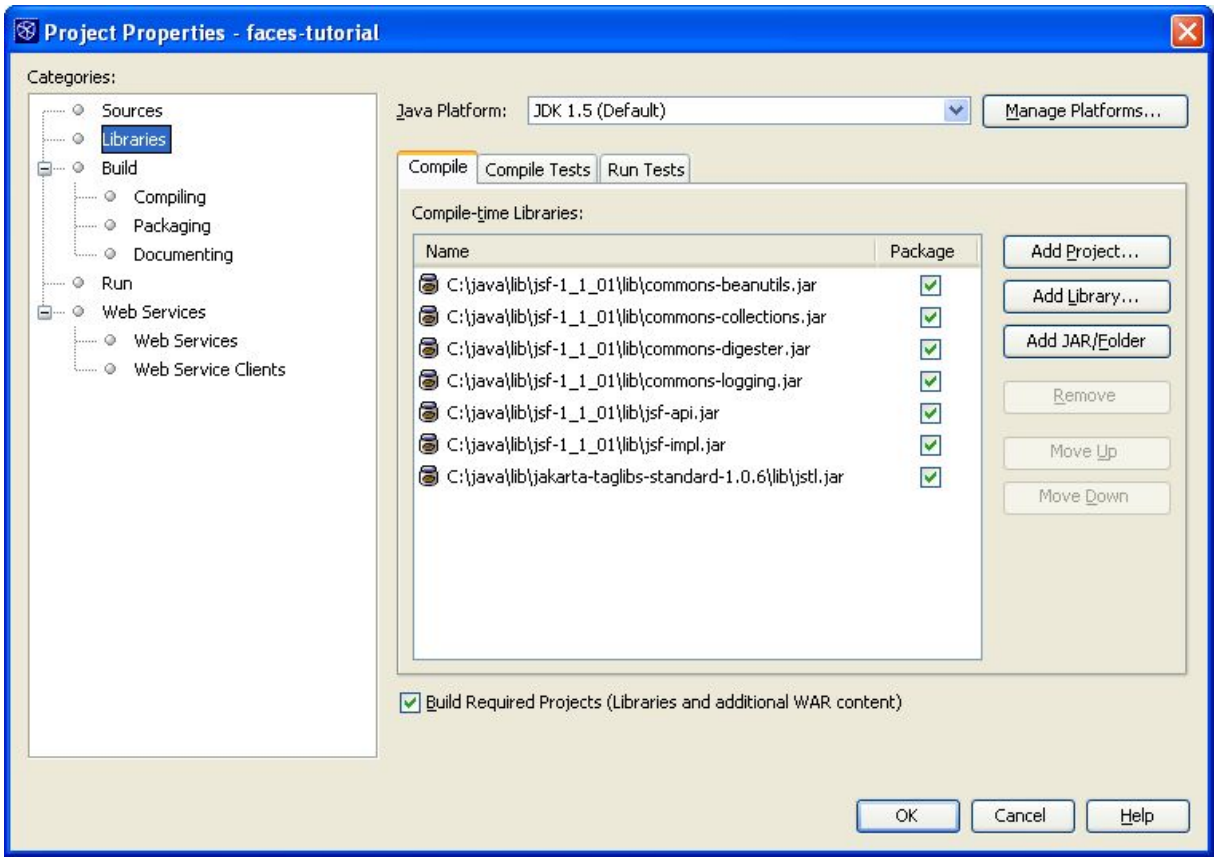


Cliquez sur *Finish*.

2.4 Configuration de l'applicaton web

Vous devez ajouter des librairies Java pour le projet. Dans un premier temps, ouvrez la configuration du projet avec un click droit sur le projet puis *Properties*.

Ensuite conformez votre configuration à celle ci :



Éditez ensuite le fichier faces-tutorial/Web Pages/WEB-INF/web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>


<web-app 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
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
       version="2.4">


<!-- Configuration de JSF -->
    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>client</param-value>
```

```
    </context-param>

    <servlet>
        <servlet-name>FacesServlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>FacesServlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
<!-- Fin de la configuration de JSF -->
```

Enfin, créer un fichier faces-config.xml dans le même répertoire que web.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>

  <application>
    <locale-config>
      <default-locale>fr</default-locale>
    </locale-config>
  </application>

</faces-config>
```

Vous êtes prêt à afficher votre première page JSF !

3 Ma première page JSF : « Hello World ! »

3.1 Hello World !

Cliquez avec le bouton droit sur le dossier Web Pages puis *New* > *JSP*. Créez le fichier avec le nom « hello-world.jsp ».

Sélectionnez l'ensemble du code de la page puis supprimer tout. Voici le code de la 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:outputText value="Hello le World ! (en JSF !)" />

</core:view>
```

Pour visualiser la page, effectuez un click droit puis *Run Project*.

Le serveur Tomcat démarre, votre page est accessible en ajoutant /faces/ avant le nom de la page.

3.2 Anatomie d'une page JSF

`<%@ page contentType="text/html" %>`	Type de contenu : HTML
`<%@ taglib uri="http://java.sun.com/jsf/html" prefix="html" %>` `<%@ taglib uri="http://java.sun.com/jsf/core" prefix="core" %>`	Import des taglibs de base de JSF (requis)
`<core:**view**>`	Définit une vue JSF (requis)
`<html:outputText value="Hello le World ! (en JSF !)" />`	Tag outputText
`</core:**view**>`	Fin de la vue

Une page JSF est donc un assemblage de tags issus des taglibs core et html. Il est cependant possible d'insérer du code HTML ou XML dans la page ou d'utiliser des taglibs supplémentaires.

3.3 Cycle de vie d'une page

Lorsque les tags de la page sont interprétés et exécutés, un arbre d'objets Java est créé côté serveur.

Les objets de cet arbre sont du type **javax.faces.component.UIComponent**, la racine de l'arbre est un objet du type **javax.faces.component.UIViewRoot.**

Cet arbre de composant est ensuite transformé (phase de rendu) dans un flux propre à la technologie du client : HTML, WML, XML, ...

Lorsque le client soumet la vue courante le flux correspondant est décodé pour reconstituer l'arbre de composants associé.

Ainsi, grâce à l'API JSF, il est possible de modifier l'arbre de composants graphiques en utilisant des instructions Java.

4 Les beans managés

4.1 Qu'est ce qu'un bean ?

Un bean est une classe Java qui respecte la spécification JavaBeans : http://java.sun.com/beans/docs/

Cette classe comprend un constructeur vide, des méthodes publique de lecture et de modification de ses attributs (getXXX et setXXX) et aucun attribut public.

Voici un bean AccountDatas qui représente les données d'un compte bancaire :

```
package com.facestut.bean;

public class AccountDatas {

    private long number = 20050000;

    public AccountDatas() {
    }

    public long getNumber(){
        return this.number;
    }

    public void setNumber(long value){
        this.number = value;
    }
}
```

4.2 Qu'est ce qu'un bean managé ?

Un bean managé est un bean dont la vie est gérée par JSF.

En effet, JSF permet de déclarer des beans dans le fichier de configuration **faces-config.xml**.

JSF instancie les beans en fonction de leur **scope** ou champs d'application puis les stocke dans le contexte JSF.

Les beans managés sont utiles pour afficher les données provenant de la couche métier ou pour la saisie de données par l'utilisateur.

Exemple de déclaration de bean managé:

```
<?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>

</faces-config>
```

Un scope égal à « session » indique qu'une instance du bean **com.facestut.bean.AccountDatas** sera présente dans chaque session HTTP d'un utilisateur connecté sur l'application.

Ce bean sera manipulé par son nom : « accountDatas ». Les scopes possibles sont : application, session ou request.

4.3 Exemple d'utilisation d'un bean managé

```
<%@ 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="Le numéro de compte :" />
    <html:outputText value="#{accountDatas.number}" />

</core:view>
```

L'écriture **#{accountDatas.number}** indique à JSF que cette expression est une EL ou **E**xpression **L**anguage.

Une EL est une expression dont le résultat est évaluée au moment ou JSF effectue le rendu de la page. La syntaxe est : **#{expression}**

Pour évaluer l'EL, JSF cherche un objet qui porte le nom **accountDatas** dans son contexte puis invoque la méthode **getNumber()** sur cet objet.

Attention ! Les EL JSF sont différentes des EL JSP qui utilisent la notation ${expression} !

Le futur JSF 1.2 utilise les EL JSP. Cependant, les EL JSF sont très proche des EL JSP. Les EL sont un point clef de JSF, pour plus d'informations consulter les liens :

http://adiguba.developpez.com/tutoriels/j2ee/jsp/el

http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro7.html#wp71019

5 Les formulaires

5.1 Les données

Reprenons notre managed bean et ajoutons lui quelques attributs et méthodes :

```
package com.facestut.bean;

public class AccountDatas {

    private long number = 20050000;
    private float total = (float) 100.0;
    private float amount;
    private Customer customer;

    public AccountDatas() {
    }

    // N'oubliez pas les get et les set !!!!
...

    public void addAmount(){
        this.total = this.total + this.amount;
        this.amount = (float)0.0;
    }

    public void retrieveAmount(){
        this.total = this.total - this.amount;
        this.amount = (float)0.0;
    }

    protected boolean checkAccount(){
        if(this.total < 0.0 ){
            return false;
        }
        else return true;
    }

    public String validate(){
        if(checkAccount()){
            return "AccountDatasOK";
        }
        else {
            return "AccountDatasError";
```

```
        }
    }
}
```

Puis un nouveau bean Customer :

```
package com.facestut.bean;

public class Customer {

    private String name = "DURAND";
    private String forname = "Paul";

    public Customer() {
    }
    // N'oubliez pas les get et les set !!!!
}
```

5.2 Saisie des données

Le tag form définit un formulaire.

Nous avons vu que le tag outputText permettait d'afficher un texte grâce à son attribut value. Le tag inputText permet de saisir la valeur d'un attribut d'une intance de bean.

Voici la page account-form.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>
        <BR/>
        <html:outputText value="Nom : #{accountDatas.customer.name}"/>
        <BR/>
        <html:outputText value="Numéro de compte : #{accountDatas.number}"/>
        <html:outputText value="Total : #{accountDatas.total}"/>
        <BR/>
        <html:outputText value="Montant :"/>
        <html:inputText value="#{accountDatas.amount}"/>
        <BR/>
    </html:form>

</core:view>
```

Lorsque le formulaire est soumis JSF utilise la méthode **setAmount(float value)** du bean AccountDatas pour injecter la valeur du champs inputText dans l'objet « accountDatas » (mapping HTML/Objet !).

Notez au passage que JSF effectue pour vous une conversion de type entre un String et un float !

Pour vous en convaincre, démarrer l'application en mode debug: click droit sur le projet puis *Debug Project*.

Poser un point d'arrêt sur l'unique ligne de la méthode **setAmount(float value)**: click gauche dans la marge de l'éditeur de code Java.

Afficher la page, changer la valeur de amount puis tapez *Entrée*.

La taglib html offre la plupart des composants de saisie connus avec HTML : **inputText**, **inputTextarea**, **inputSecret**, ainsi que des composants de sélections comme **selectBooleanCheckbox**, **selectOneRadio**, ...

6 La navigation

6.1 Principes

JSF implémente un automate à états finis pour gérer la navigation entre les pages.

A l'aide de règle de navigation ou « navigation rule » décrite dans le fichier faces-config.xml, JSF va déterminer en fonction de la page courante quelle est la page suivante.

Une règle de navigation s'écrit de la manière suivante :

```
  ...
  <navigation-rule>

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

    <navigation-case>
        <!-- Si l'outcome renvoyé est  HelloWorldAction
       alors JSF passe à la page /hello-world.jsp -->
        <from-outcome>HelloWorld</from-outcome>
        <to-view-id>/hello-world.jsp</to-view-id>
    </navigation-case>

  </navigation-rule>

</faces-config>
```

Deux composants JSF déclenchent des actions lorsque l'utilisateur clique sur leur représentation graphique : **HtmlCommandButton** (bouton submit de HTML) et **HtmlCommandLink** (hyperlien HTML).

Ces deux composants sont crées par les tags correspondants de la taglib html : **commandLink** et **commandButton**.

Lorsque l'utilisateur clique sur l'un ou l'autre, une clef de navigation ou **outcome** est renvoyée. Cet clef peut être statique, en dur dans le code de la page JSP, ou dynamique, calculée par une EL.

Il est possible d'utiliser des expressions régulières dans la définition des règles de navigation. Cela permet d'inclure un grand nombre de cas en une seule expression.

6.2 Navigation statique

Créons la page index.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:commandLink action="HelloWorld" value="Exemple Hello World"/>
    </html:form>
</core:view>
```

Lorsque l'utilisateur clique sur le lien HTML correpondant alors l'outcome **HelloWorld** est renvoyé au gestionnaire de navigation de JSF (NavigationHandler).

Le gestionnaire cherche une règle de navigation applicable au contexte courant :

- si une correspondance est trouvée alors la page suivante est affichée
- sinon la page courante est rechargée

6.3 Navigation dynamique

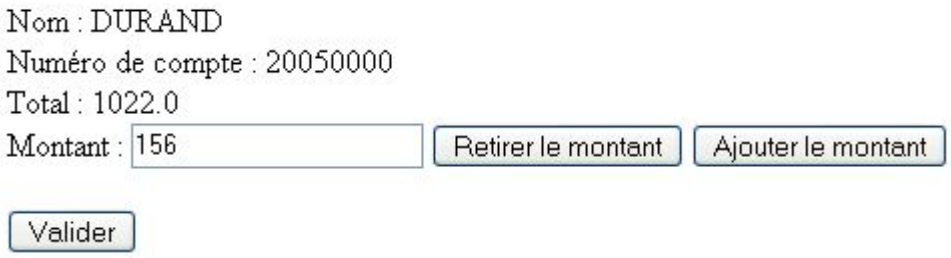
Dans le cas de la navigation dynamique, l'outcome est inconnu au moment ou la page est écrite. Ici, c'est un managed bean qui va calculer l'oucome via l'appel de l'une de ses méthodes.

Pour invoquer une méthode sur un managed bean il suffit d'utiliser une EL. Reprenons notre formulaire de saisie et ajoutons lui quelques boutons :

```
<%@ 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}"/>
        <html:outputText value="Total : #{accountDatas.total}"/>
        <BR/>
        <html:outputText value="Montant :"/>
        <html:inputText value="#{accountDatas.amount}"/>
        <html:commandButton value="Retirer le montant"
             action="#{accountDatas.retrieveAmount}"/>
        <html:commandButton value="Ajouter le montant"
             action="#{accountDatas.addAmount}"/>
        <BR/><BR/>
        <html:commandButton value="Valider" action="#{accountDatas.validate}"/>
        <BR/>
    </html:form>

</core:view>
```



Nous avons ici deux cas de figure.

Si l'utilisateur clique sur le boutons « Retirer le montant » alors la méthode « public void retrieveAmount() » du bean « accountDatas » est appelée. Or cette méthode est définie comme **void** ce qui est interprété comme un outcome null.

Un outcome null provoque le rechargement de la page courante.

Dans le cas du bouton « Valider », la méthode invoquée renvoie un **String**. La valeur de l'outcome varie en

fonction du résultat de la méthode « validate() ».

La navigation qui en résulte est donc variable. Les valeurs possibles sont ici « **AccountDatasOK** » ou «**AccountDatasError** » .

Dans ce cas, nous devons déclarer une nouvelle règle de navigation dans faces-config.xml:

```
...
  <navigation-rule>
    <from-view-id>/account-form.jsp</from-view-id>

    <navigation-case>
        <from-outcome>AccountDatasOK</from-outcome>
        <to-view-id>/success.jsp</to-view-id>
    </navigation-case>

    <navigation-case>
        <from-outcome>AccountDatasError</from-outcome>
        <to-view-id>/error.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

</faces-config>
```

Je vous laisse le soin de créer les pages « success.jsp » et « error.jsp » sur la base de « hello-world.jsp ».

7 Les tables de données

7.1 Une table simple

Pour éxécuter cet exemple il est nécessaire de créer le bean « Bank » et de le déclarer dans faces-config.xml. :

```
package com.facestut.bean;

import java.util.ArrayList;
import java.util.List;

public class Bank {

    private List customers = new ArrayList();
```

```
    public Bank() {
        this.customers.add(new Customer("DURAND","Paul"));
        this.customers.add(new Customer("DUDULE","Michel"));
        this.customers.add(new Customer("MARTIN","Athur"));
        this.customers.add(new Customer("RICARD","Paul"));
    }

    public List getCustomers(){
        return this.customers;
    }

    public void setCustomers(List customers){
        this.customers = customers;
    }
}
```

Puis dans faces-config.xml :

```
...
  <managed-bean>
    <managed-bean-name>bank</managed-bean-name>
    <managed-bean-class>com.facestut.bean.Bank</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
...
```

Nous sommes prêt pour créer et afficher la page data-table.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 value="#{bank.customers}" var="customer" border="1">

            <html:column>
                <html:outputText value="#{customer.name}"/>
            </html:column>

            <html:column>
                <html:outputText value="#{customer.forname}"/>
            </html:column>

        </html:dataTable>
    </html:form>
</core:view>
```

Le tag **dataTable** définit une table de données. L'attribut **value** indique au composant comment récupérer les objets.

Il s'agit d'un bean qui possède une méthode qui renvoie une collection d'objets. Ici la méthode « getCustomers () » du bean « bank » est invoquée.

Le composant va itérer sur la collection d'objets fournie par value. Chaque objet de la collection sera accessible au moment ou le composant itère sur lui au moyen de la variable spécifiée par l'attribut **var**.

Ainsi, **#{customer.forname}** indique que la méthode « getForname() » sera appelée sur chaque objet Customer de la collection.

7.2 Ajout du titre des colonnes

```
<%@ 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 value="#{bank.customers}" var="customer" border="1">
            <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>
    </html:form>
</core:view>
```

Le tag **facet** permet d'associer un composant à un autre sans qu'il y ait une relation de filiation entre les deux composants.

Ici le composant qui représente le titre de la colonne **<core:verbatim>Nom</core:verbatim>** n'est pas un composant fils de **column** ou **dataTable**. Les composants fils sont rendus les uns après les autres.

Un nom de colonne doit être rendu avant de rendre la première ligne de la table. Une facet permet à un composant d'avoir à sa disposition, via le nom de la facet, d'autres composants qu'il pourra rendre indépendamment de ses composants fils.

8 Le binding des composants

8.1 Principes

Le binding d'un composant permet de lier un composant JSF à une propriété d'un bean. Lorsque le binding d'un composant est fait alors il est possible d'agir sur le composant via l'API JSF.

Reprenons notre exemple de table de données :

```
<%@ 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="#{bank.dataTable}" value="#{bank.customers}"
             var="customer" border="1">

            <html:column>
                <html:outputText value="#{customer.name}"/>
            </html:column>

            <html:column>
                <html:outputText value="#{customer.forname}"/>
            </html:column>

        </html:dataTable>
    </html:form>
</core:view>
```

L'attribut **binding** est présent sur la plupart des tags JSF. Lorsque le tag est exécuté alors le composant JSF associé sera accessible à partir du bean spécifié.

Ici, **binding="#{bank.dataTable}"** indique que le bean « bank » possèdera une référence sur le composant de type HtmlDateTable créer par le tag.

Pour accéder à ce composant il convient d'utiliser les méthodes get et set correspondantes.

Voici le bean bank modifié :

```
package com.facestut.bean;

import java.util.ArrayList;
import java.util.List;
import javax.faces.component.UIData;

public class Bank {

...
    // UIData ancêtre de HtmlDataTable
    private UIData dataTable;
...

    public UIData getDataTable(){
        return this.dataTable;
    }

    public void setDataTable(UIData dataTable){
        this.dataTable = dataTable;
    }
...
}
```

8.2 Exemple d'utilisation du binding

Modifions notre exemple, ci dessus pour ajouter deux actions fréquentes sur une table de données : l'ajout et la suppression.

```
...
        <html:dataTable binding="#{bank.dataTable}" value="#{bank.customers}"
            var="customer" border="1">
            <html:column>
                <html:selectBooleanCheckbox binding="#{bank.checkbox}"/>
            </html:column>
...
        </html:dataTable>
        <br>
        <html:commandButton value="Supprimer les clients"
                    action="#{bank.removeSelectedCustomers}"/>
        <html:commandButton value="Ajouter un client"
                    action="#{bank.addCustomer}"/>
```

```
    </html:form>
</core:view>
```

Remarquez l'ajout d'une nouvelle colonne qui contiendra une case à cocher. Modifions le bean pour effectuer le binding entre la case à cocher et le bean :

```
import javax.faces.component.UISelectBoolean;
...
public class Bank {

...
    private UISelectBoolean checkbox;
...

    public UISelectBoolean getCheckbox(){
        return this.checkbox;
    }

    public void setCheckbox(UISelectBoolean checkbox){
        this.checkbox = checkbox;
    }

    public void removeSelectedCustomers(){
        int size = this.dataTable.getRowCount();
        List selectedCustomers = new ArrayList();
        for(int i=0; i < size; i++){
            this.dataTable.setRowIndex(i);
            if(this.checkbox.isSelected()){
                selectedCustomers.add(this.customers.get(i));
            }
        }
        this.customers.removeAll(selectedCustomers);
    }

    public void addCustomer(){
        Customer customer = new Customer();
        customer.setName("Nouveau");
        customer.setForname("client");
        this.customers.add(customer);
    }
}
```

Remarquez l'ajout des deux méthodes **removeSelectedCustomers** et **addCustomer**.

Regardons en détail la méthode **removeSelectedCustomers**.

Cette méthode est déclenchée lorsque l'utilisateur clique sur le bouton « Supprimer les clients ». L'utilisateur agit sur l'interface et soumet la page.

Lorsque JSF reçoit la page il reconstitue l'arbre de composants graphiques et met à jour ces composants avec les agissements de l'utilisateur.

Ici, une ou plusieurs cases à cocher ont été actionnées, cet état se reflète dans l'arbre de composants.

Ainsi, il est possible, via le binding, d'interroger le composant UISelectBoolean qui correspond au tag **selectBooleanCheckbox** pour connaître son état : coché ou non coché (selected).

9 Design patterns pour JSF

Lorsque on analyse la classe Java Bank, il apparaît que cette classe assure plusieurs responsabilités bien différentes :

- modélise une banque (comme son nom l'indique)
- propose des services d'ajout et de retrait de client
- assure le binding des composants JSF
- gère la navigation liée à la deuxième responsabilité

Tout ça pour une seule classe !!!!

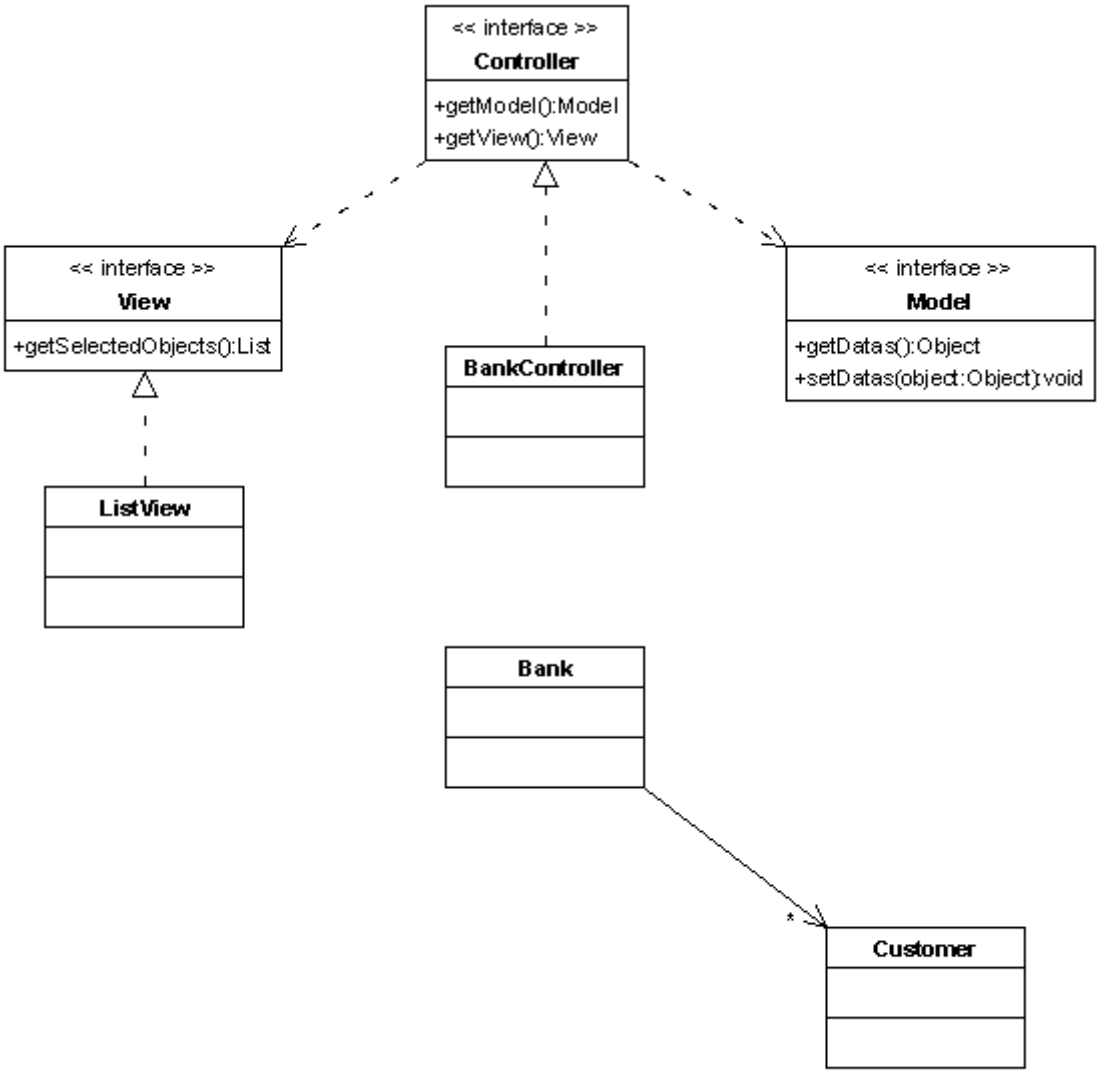
Les préconisations en matière de conception objet nous incitent à séparer les responsabilités.

En effet, si l'interface graphique change je serais peut être amené à changer le binding des composants dans la classe alors même que le concept de banque n'a pas changé.

Si chaque développeur travaille sur un domaine précis, le métier ou l'interface graphique, ils risquent d'être en concurrence sur la modification de la classe Bank.

Il faut donc éclater cette classe en plusieurs autres classes spécialisées. Le pattern Modèle Vue Contrôleur va nous y aider.

Voici un diagramme UML proposant une nouvelle conception:



Le Controller joue les intermédiaires entre la vue et le modèle.

La vue réprésente l'arbre de composants JSF sur lequel l'utilisateur va agir par l'entremise du navigateur et reçoit le binding .

Le modèle représente les données qui sont manipulées par l'utilisateur via la vue.

Les classes Bank et Customer n'assurent plus qu'une responsabilité : modéliser le domaine métier.

Ce pattern vous permettra de réutiliser les fonctionnalités de manipulation de liste d'objets avec JSF, car certaines classes et interfaces du diagramme sont génériques.

```
package com.facestut.bean;

import java.util.ArrayList;
import java.util.List;

public class Bank {

    private List customers = new ArrayList();

    public Bank() {
        this.customers.add(new Customer("DURAND","Paul"));
        this.customers.add(new Customer("DUDULE","Michel"));
        this.customers.add(new Customer("MARTIN","Athur"));
        this.customers.add(new Customer("RICARD","Paul"));
    }

    public List getCustomers(){
        return this.customers;
    }

    public void setCustomers(List customers){
        this.customers = customers;
    }
}
```

Les **View** et **ListView** :

```
package com.facestut.mvc;

import java.util.List;



public interface View {

    public List getSelectedObjects();
}
```

```

package com.facestut.mvc;

import java.util.ArrayList;
import java.util.List;
```

```
import javax.faces.component.UIData;
import javax.faces.component.UISelectBoolean;



public class ListView implements View {

    private UIData dataTable;
    private UISelectBoolean checkbox;

    public ListView() {
    }

     public UIData getDataTable(){
        return this.dataTable;
    }

    public void setDataTable(UIData dataTable){
        this.dataTable = dataTable;
    }

    public UISelectBoolean getCheckbox(){
        return this.checkbox;
    }

    public void setCheckbox(UISelectBoolean checkbox){
        this.checkbox = checkbox;
    }

    public List getSelectedObjects(){

        int size = this.dataTable.getRowCount();
        List datas = (List) this.dataTable.getValue();
        List selectedObjects = new ArrayList();
        for(int i=0; i < size; i++){
            this.dataTable.setRowIndex(i);
            if(this.checkbox.isSelected()){
                selectedObjects.add(datas.get(i));
            }
        }
        return selectedObjects;
    }

}
```

```
package com.facestut.mvc;



public interface Model {

    public Object getDatas();

    public void setDatas(Object object);
}
```

```
package com.facestut.mvc;

import java.util.ArrayList;
import java.util.List;



public class SimpleModel implements Model {

    private Object datas;

    public SimpleModel() {
    }

    public Object getDatas(){
        return this.datas;
    }

     public void setDatas(Object object){
         this.datas = object;
     }
}
```

```

package com.facestut.mvc;

import com.facestut.mvc.Model;
import com.facestut.mvc.View;



public interface Controller {

    public Model getModel();

    public View getView();
}
```

```
package com.facestut.mvc;

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

public class BankListController implements Controller {

    private Model model;
    private View view ;

    public BankListController() {
        this.model = new SimpleModel();
        this.view = new ListView();
        this.model.setDatas(new Bank());
    }

    public View getView(){
        return this.view;
    }

    public void  setView(View view){
        this.view = view;
    }

    public Model getModel(){
        return this.model;
    }

    public void setModel(Model model){
        this.model = model;
    }

     public void removeSelectedCustomers(){
        Bank bank = (Bank) getModel().getDatas();
        bank.getCustomers().removeAll(getView().getSelectedObjects());
    }

    public void addCustomer(){
        Bank bank = (Bank) getModel().getDatas();
        Customer customer = new Customer();
        customer.setName("Nouveau");
        customer.setForname("client");
        bank.getCustomers().add(customer);
    }
}
```

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>
```

Enfin, n'oublions pas de déclarer notre contrôleur dans le fichier faces-config.xml :

```
...
<managed-bean>
    <managed-bean-name>bankCtrl</managed-bean-name>
    <managed-bean-class>com.facestut.mvc.BankListController</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
...
```

10 Pour aller plus loin

Nous avons construit des pages simples grâce aux tags JSF.

Cependant, si vous devez construire une application de gestion complète (validation, message, ...) alors vos besoins dépassent le cadre de ce tutoriel.

JSF permet néanmoins d'aller plus loin: valider la saisie d'un utilisateur grâce aux validators, convertir une valeur saisie vers un type donné grâce aux converters, afficher des messages, ...

JSF offre également le concept de composants customs qui augmentent la productivité tout en diminuant la maintenance. A ce titre, JSF définit un cycle de vie qui détermine rigoureusement l'implémentation et le fonctionnement des composants.

Vous découvrirez tout ceci en détail dans la suite de ce tutoriel.