
JSF 2

(framework java/web)

Table des matières

I - Présentation de JSF	4
1. Framework JSF.....	4
2. Implémentations de JSF.....	6
3. Fonctionnement interne de JSF.....	7
4. Modèle de composants abstraits et kit de rendus.....	8
5. Intégration de JSF au sein d'une application Web	10
6. Configuration maven pour MyFaces 2 (JSF2).....	11
II - Managed Beans	12
1. Relations entre formulaire et JavaBean.....	12
2. Configuration au sein de faces-config.xml.....	12
3. Code type d'un JavaBean JSF.....	13
4. Configuration via annotations (depuis JSF 2).....	13
5. Accès aux résultats pour affichage.....	14
III - Navigations JSF (règles , contrôleur).....	15
1. Principes généraux de la navigation JSF.....	15
2. Configuration au sein de faces-config.xml.....	16

3. Bean "mini contrôleur" pour navigation.....	17
4. Navigations implicites depuis la version 2 de JSF.....	18

IV - Validations JSF et Contexte..... 19

Bon paramétrage des formulaires jsf.....	19
1. Validation automatique et implicite.....	20
2. Validation automatique et explicite.....	21
3. Messages globaux / erreurs de traitement.....	23
4. Annotations compatibles JSF2 pour la validation.....	24
5. Validations avancées (en annexes).....	24

V - Facelet (templates)..... 25

1. Présentation des facelets pour JSF2.....	25
2. Pages ".xhtml" et plus de pages ".jsp".....	25
3. Notion de "template".....	27
4. Mise en oeuvre des "templates" de jsf/facelet.....	28
5. Définition d'une page s'appuyant sur un template.....	29
6. Configurations conseillées (pour facelets).....	29
7. Templates évolués (avec sous templates).....	31

VI - Composants JSF standards (de base)..... 32

1. Éléments de base (zone de saisie, bouton , ...).....	32
2. Case à cocher simple.....	32
3. Lien hypertexte.....	32
4. Zone de Liste (à sélection simple).....	33
5. Liste de bouton radios exclusifs (SelectOneRadio).....	34
6. Alignement via h:panelGrid	34
7. Sélections multiples (via cases à cocher ou liste).....	35
8. Tableaux (<h:dataTable > , <h:column>).....	36

VII - Événements JSF..... 38

1. Catégories d'événements et gestionnaires.....	38
2. Gestionnaires d'événements (listener).....	38

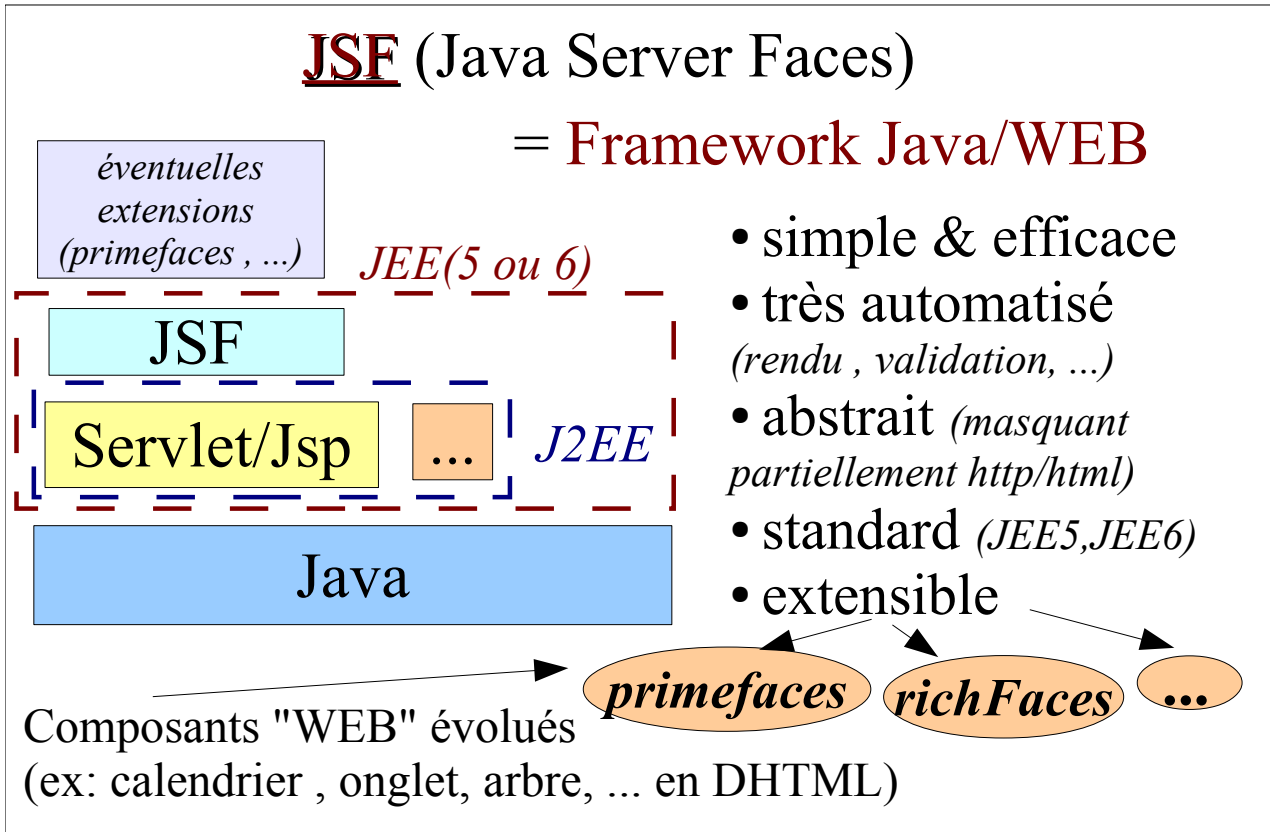
VIII - Principaux apports de JSF2..... 41

1. Gestion du mode "GET" (depuis JSF2).....	41
2. Autres apports de JSF2.....	42

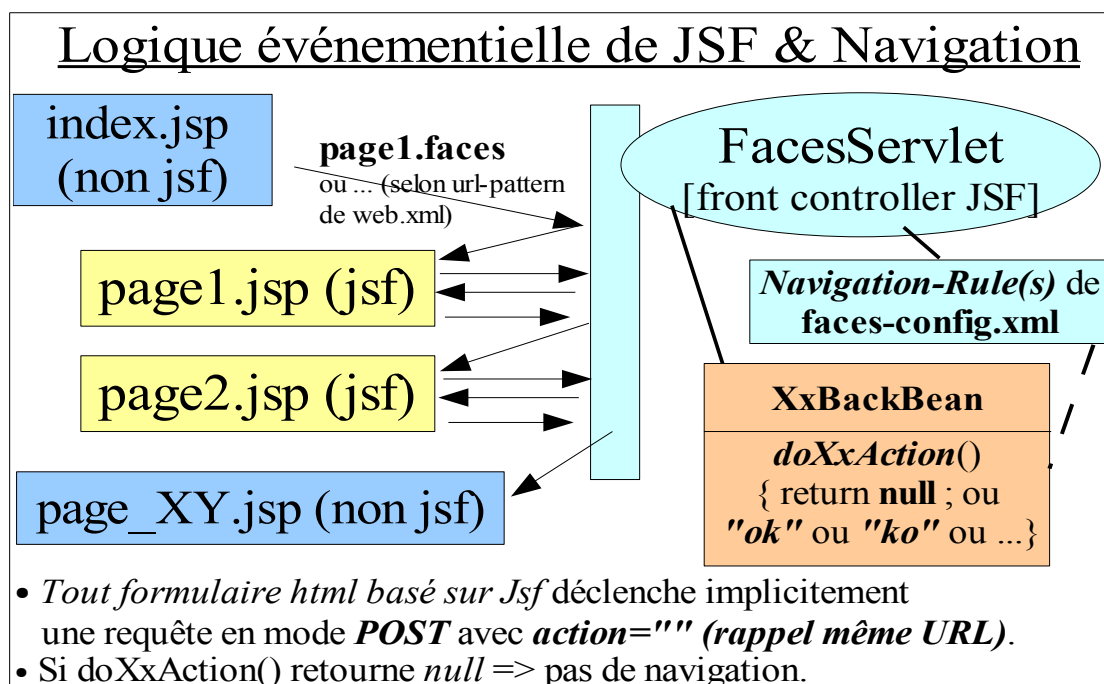
IX - Extensions JSF (tomahawk , richFaces, ...)	44
1. Principales extensions pour JSF (aperçu)	44
2. richFaces	44
3. PrimeFaces	48
X - Annexe – Validations avancées	51
Différents styles pour les validations	51
1. Validation manuelle (libre mais longue)	51
2. Eventuelle personnalisation des messages JSF	52
3. Valideur spécifique (custom validator)	53
XI - Annexe - Properties & internationalisation	54
1. Fichier de propriétés (une version par langue)	54
2. Prise en compte au sein des pages JSP	54
3. Ressource globale (à déclarer dans faces-config.xml)	55
XII - Annexe – astuces diverses , évolutions JSF	56
1. Apports de EL 2.2 (depuis Tomcat 7 / JEE6)	56
2. DataModel (JSF)	56
NB: bien que la solution "f:setPropertyActionListener" présentée ci après soit en générale plus simple (plus directe) que la solution "DataModel" pour récupérer l'élément sélectionné dans un tableau , la solution "DataModel" offre l'intérêt d'être assez bien intégrée dans le framework "spring-web-flow"	57
3. Astuces diverses pour JSF	58
4. Redirection automatique en cas de session terminée/expirée	59
XIII - Annexe - paramètres jsf & backing Bean	60
1. Backing Bean	60
2. Passage de paramètre(s)	60
XIV - Annexe - Intégration IOC (jsf , jee , spring)	62
1. Injection IOC via JSF : plusieurs beans	62
2. Liens entre JSF et Spring	64
3. Eventuelles injections par annotations	66
XV - Annexe - composants composites	72
1. Composants composites (depuis JSF2)	72

I - Présentation de JSF

1. Framework JSF



1.1. Logique événementielle et navigation



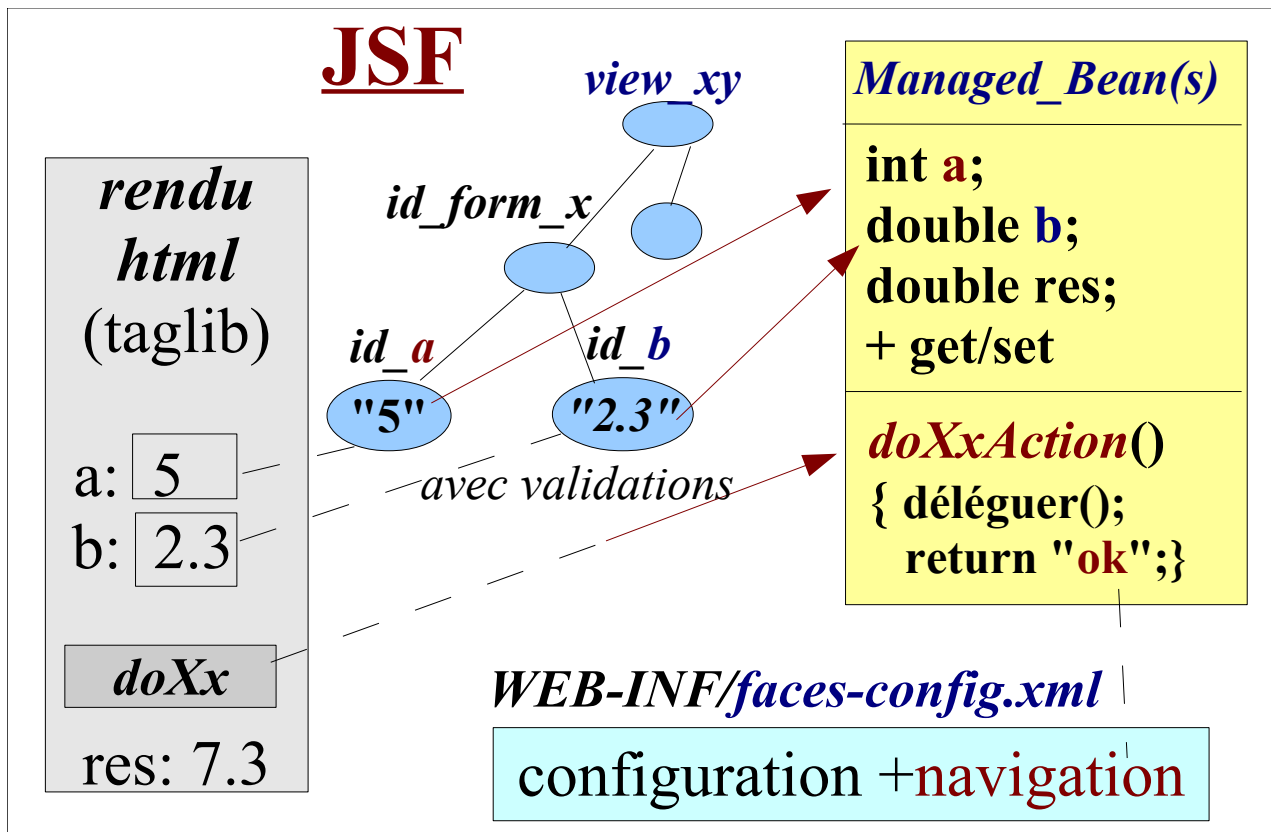
1.2. Quelques traits de JSF

JSF ==> accès aux composants via leurs noms , pas d'héritage imposé (contrairement à STRUTS)

JSF ==> fichier de configuration simple : *faces-config.xml* ou *annotations* avec **JSF2**

JSF ==> orienté "Drag & Drop" + propriétés + événements

1.3. Arbre de composants JSF (racine=view)



Un servlet prédéfini (`javax.faces.webapp.FacesServlet`) interne au framework JSF intercepte la requête http (`url = "....../faces/xxx.faces"`) pour effectuer les choses suivantes:

- Tenir compte du modèle de composants (ex: form html, zone de texte html , ...) décrit par les "taglib" de xxx.jsp pour mettre en place en mémoire un **arbre de composants** . Chacun de ces composants de bas niveau va ensuite être rempli avec l'une des données véhiculées par les paramètres de la requête http.
- En fonction des vérifications qui sont alors automatiquement déclenchées (valeurs numériques ?, valeurs comprises entre un minimum et un maximum ? , ...), des messages d'erreurs seront s'il le faut automatiquement générés et renvoyés au sein du formulaire de saisie qui réapparaît alors dans le navigateur (en même temps que les anciennes valeurs saisies).
- Si aucune erreur n'est détectée , un peuplement automatique d'un JavaBean de données/traitements (précisé par xxx.jsp ou xxx.xhtml) est alors déclenché puis un mécanisme événementiel prend ensuite le relais pour continuer les opérations.

2. Implémentations de JSF

L'api **JSF** (et sa documentation officielle) est à **télécharger** depuis le site de SUN.

Le package officiel de l'API JSF est *javax.faces* (javax.faces.event , javax.faces.component.html , ...).

Versions de JSF:

Versions de l'API <i>JSF</i>	Caractéristiques
1.0 (2004)	basé sur Servlet 2.3 & Jsp 1.2 (pour J2EE 1.3 ou 1.4 ou +)
1.1	comme 1.0 mais avec moins de bugs et avec de meilleures performances (pour J2EE 1.3 ou 1.4 ou +)
1.2	basé sur Servlet 2.5 & Jsp 2 (pour Java5 /JEE5) [nécessite Tomcat 6, ...] Ajouts de la version 1.2 : <ul style="list-style-type: none"> - unified expression language (EL commun à JSP2 & JSF) - ajax support , , fichier de config avec xsd (et plus de dtd)
2.0 , 2.1 , 2.x (depuis JEE6)	Avec annotations , navigations par défaut et intégration des " facelets " <i>Attention</i> : certaines nouveautés de JSF2 exigent ". xhtml " ("jsp") !

Implémentations:

Implémentations de JSF	Editeur	Caractéristiques
MyFaces	Jakarta/Apache	s'intègre facilement dans une appli. Java/web (pour Tomcat ou ...). http://myfaces.apache.org/
Jsf-Sun-RI	Sun	http://java.sun.com/j2ee/javaserverfaces/download.html
ibm-jsf	IBM	intégré à RAD6 , RAD7 et WebSphere 6.0 , 6.1

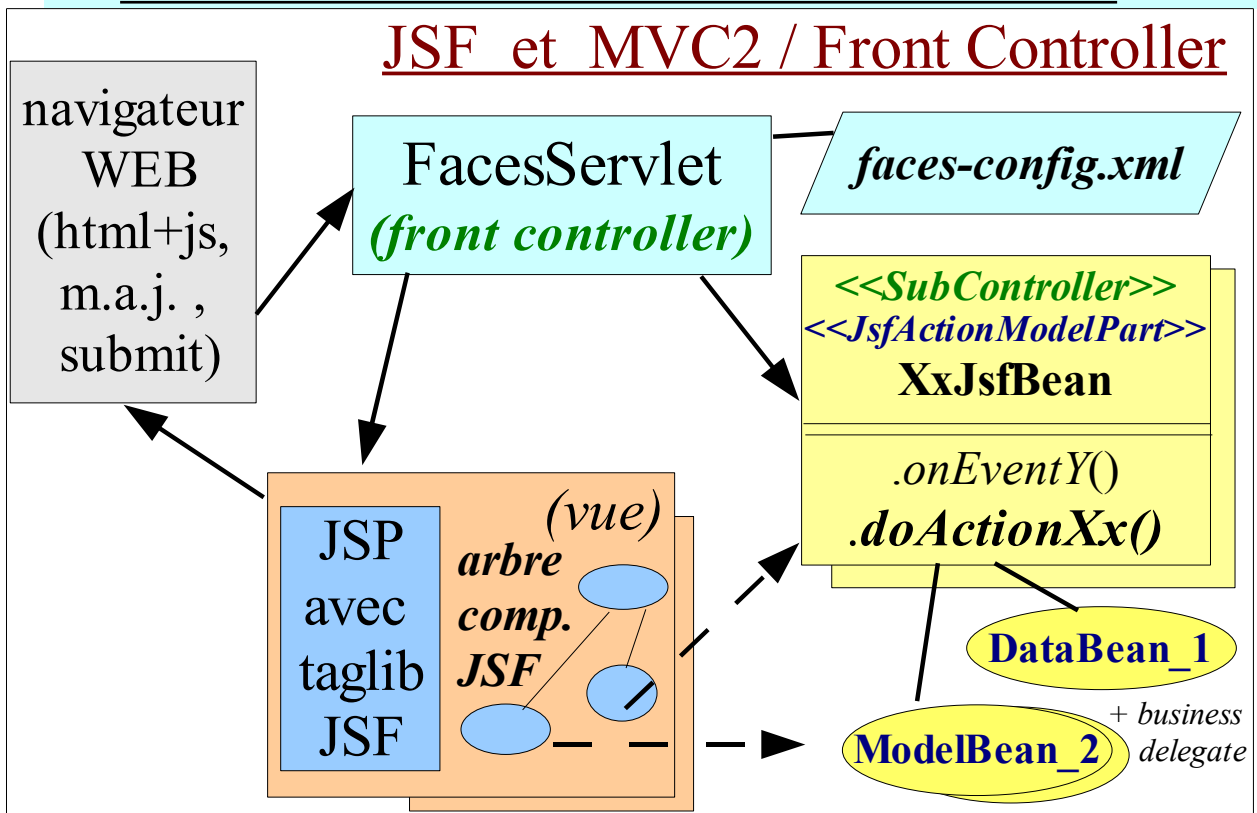
Extensions graphiques pour JSF:

composants graphiques JSF évolués (ex: calendrier , onglets , arbres , menus déroulants) [*rendu en Html + javaScript*]

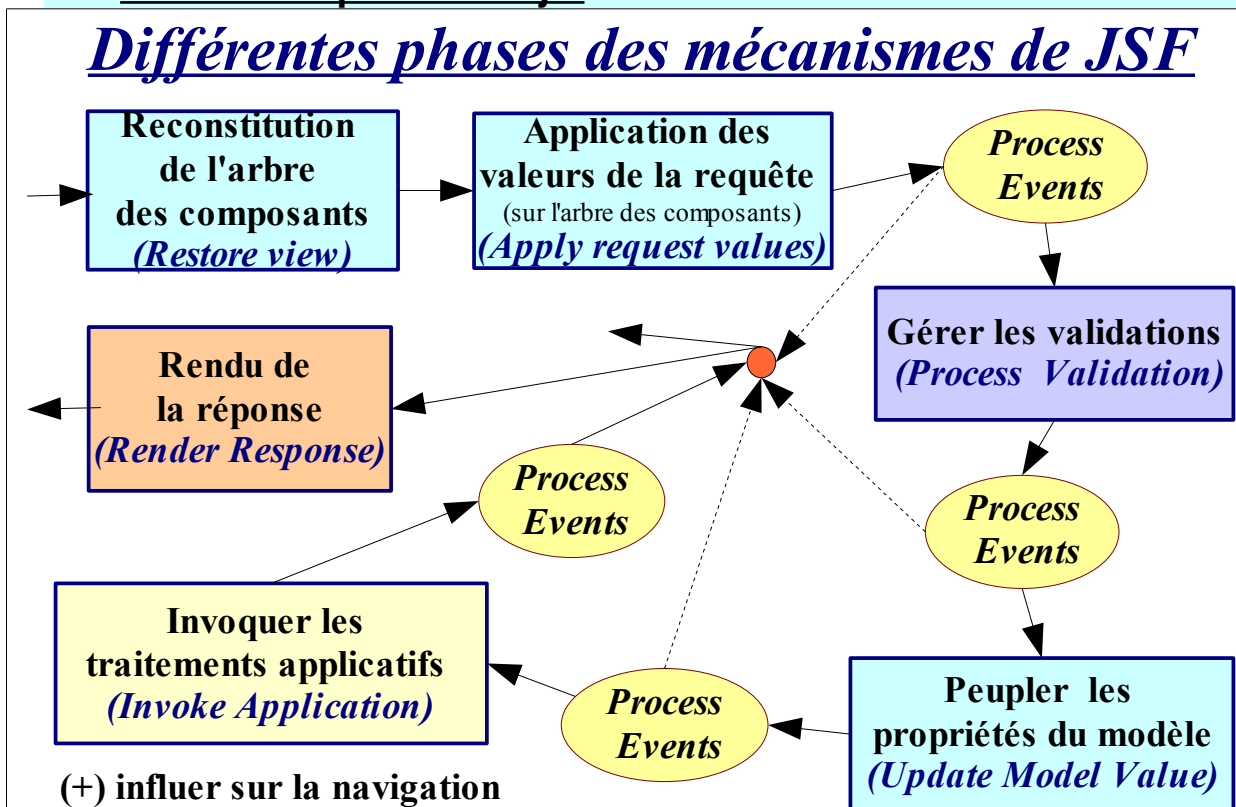
Implémentations de JSF	Editeur	Caractéristiques
Tomahawk	Jakarta/Apache	<i>Historiquement première extension pour JSF (sans ajax)</i>
RichFaces 3	JBoss Labs	<i>extension pour Ajax + composants graphiques évolués (intégration très simple dans JSF 1.2)</i>
RichFaces 4 et 5	JBoss Labs	<i>Versions (complexes) de richfaces pour JSF2</i>
icesFaces	ICEsoft	Ajax + composants graphiques évolués
primeFaces	primefaces.org	Très bonne extension (simple) optimisée pour JSF2
		http://www.jsfmatrix.net/ = URL avec liste extensions

3. Fonctionnement interne de JSF

3.1. Lien entre JSF et le modèle "MVC2 / Front Controller"



3.2. Différentes phases de jsf



<i>Phases</i>	<i>Traitements effectués</i>
Reconstitution de l'arbre des composants	Construire en mémoire (coté serveur) un arbre de composants en fonction des informations de la requête (type et id des composants).
Application des valeurs de la requête	Stocker les valeurs saisies (param HTTP) au sein des composants de l'arbre
Validations	Construire et stocker des messages d'erreurs en fonction des erreurs de conversions et des contraintes explicites (mini , maxi , ...). En cas d'erreur de validation , empiler un événement spécial qui routera automatiquement la navigation vers la page d'origine (en court-circuitant les traitements applicatifs).
Peupler les propriétés du modèle	Recopier les valeurs des composants dans les propriétés du modèle (JavaBean).
Gérer les événements	Déclenchement des méthodes événementielles. Ces déclenchements peuvent (selon les cas) intervenir avant ou après les phases de validation et de peuplement du modèle.
Invoquer les traitements applicatifs et piloter navigation	- Déclenchement des traitements applicatifs (actions) - Gestion de la navigation (redirection vers d'autres pages)
Phase de rendu de la réponse	Prise en compte des tags de rendu JSF au sein d'une page JSP pour représenter concrètement (en HTML/JavaScript) les composants abstraits de la page de réponse . Les données à afficher sont puisées dans le modèle. En cas d'erreur(s) survenue(s) préalablement lors de la validation, le rendu s'effectue sur la page d'origine.

4. Modèle de composants abstraits et kit de rendus

Classe abstraite de base : `javax.faces.component.UIComponent`

Identifiant sous forme de chaîne courte ==> `getId()` / `setId(String)`

Imbrication de composants ==> arbre d'identificateurs

Type (De rendu): *variante pour le rendu*

Famille : `getFamily()`

Arbre des composants:

`getParent()` / `setParent(UIComponent...)`

List `getChildren()`

`getChildCount()`

Sous classe abstraite (avec plus grande partie programmée) : `UIComponentBase`

NB:

Un composant abstrait (**UIComponent**) représente avant tout un **ensemble de fonctionnalités** .
Sa représentation graphique concrète sera gérée par un **objet annexe de rendu** (d'un kit ...).

NB : les classes de **UIComponent** (UIInput , UIForm , UICommand , ...) correspondent à des familles de composants.

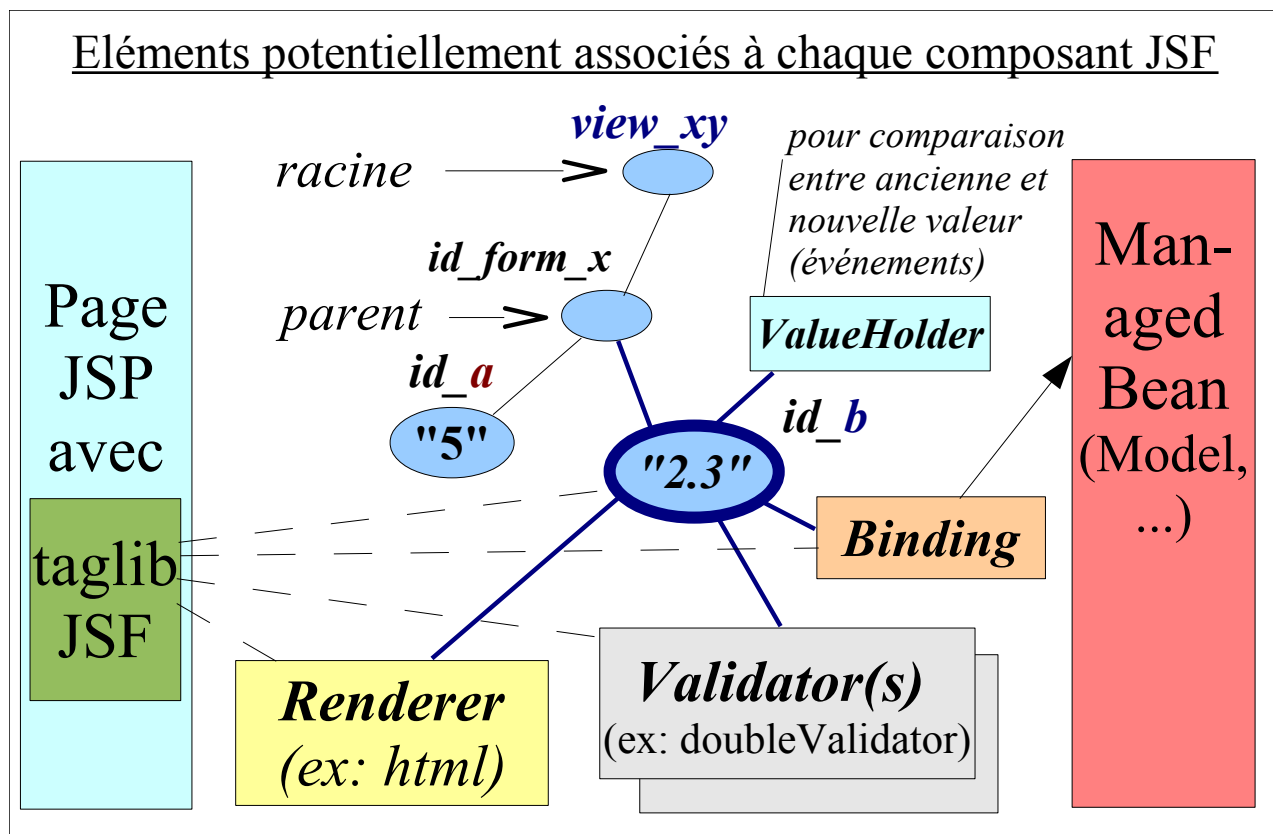
Certaines familles (ex: "UICommand ") supportent **plusieurs variantes de rendus** appelées "Types de rendu" (ex: "javax.faces.Button" , "javax.faces.Link" , ...).

Renderer rendu = KitDeRendu.getRenderer(family_string , rendererType_string);

...

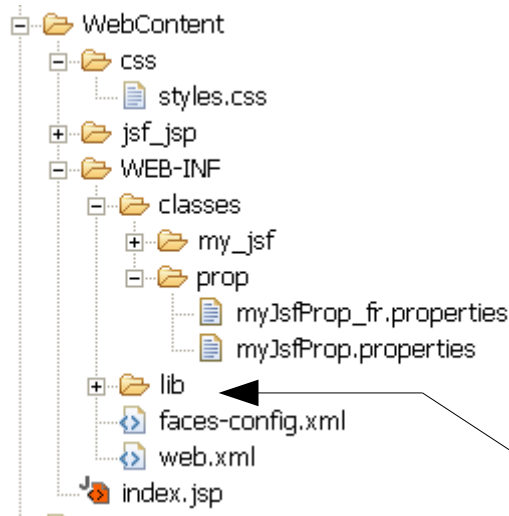
---> détails secondaires dans une annexe externe (pdf supplémentaire)

4.1. Éléments gravitants autour d'un composant JSF



5. Intégration de JSF au sein d'une application Web

Organisation classique liée aux spécifications de J2EE:



myfaces-api-.....jar + myfaces-impl...jar

Cette configuration utilise l'implémentation **myfaces** de la communauté Apache.

myfaces-impl...jar comporte directement en lui les ".tld" et "Messages.properties" nécessaires. D'autres configurations sont envisageables .

Extraits importants de WEB-INF/web.xml :

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

<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">
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

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

  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern> <!-- ou *.faces ou /faces/* ou ...-->
  </servlet-mapping>
</web-app>
```

Remarque importante pour l'IDE eclipse:

Au sein d'un projet **eclipse** de type "**Dynamic Web Project**" , on peut activer le menu "**project / properties / project Facet**" puis cocher et paramétrer "**Java Server Faces**" (v 1.2 ou 2.1 , *.faces ou *.jsf) de façon à **générer automatiquement un début de fichier faces-config.xml et la partie "Faces Servlet" de web.xml** . NB: choisir "**disable librairies configuration**" .

6. Configuration maven pour MyFaces 2 (JSF2)

pom.xml

```
...
    <properties>
        ...
        <org.apache.myfaces.version>2.0.5</org.apache.myfaces.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.1</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version> <!-- old: 1.1.2 not for jsf2 -->
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.myfaces.core</groupId>
            <artifactId>myfaces-api</artifactId>
            <version>${org.apache.myfaces.version}</version>
            <scope>compile</scope>
        </dependency>

        <dependency>
            <groupId>org.apache.myfaces.core</groupId>
            <artifactId>myfaces-impl</artifactId>
            <version>${org.apache.myfaces.version}</version>
            <scope>runtime</scope>
        </dependency>
        ...
    </dependencies>
    ...
```

Ceci est utile pour Tomcat 6 ou 7 .

Jboss 7 comporte déjà une implémentation de JSF2 et n'a donc pas besoin de MyFaces2

II - Managed Beans

1. Relations entre formulaire et JavaBean

La syntaxe `value="#{nomJavaBean.propriété}"` permet d'associer une zone graphique JSF (`<h:inputText>`, ...) d'un formulaire à une propriété d'un JavaBean.

Cette association servira :

- à récupérer initialement certaines *valeurs à afficher par défaut* (celles qui proviennent du *constructeur par défaut*).
- à *peupler automatiquement les propriétés d'un JavaBean en fonction des valeurs saisies* suite à la soumission d'un formulaire (submit) .

param_calcul.jsp:

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

<html>
<head>
  <title>Param_calcul</title>
  <LINK rel="stylesheet" href="../css/styles.css" type="text/css" />
</head>
<body>
<f:view>
  <h:messages styleClass="RedCssClass"/>
  <hr/>
  <h:form>
    Montant:<h:inputText value="#{calculBean.montant}"/> <br/>
    Taux: <h:inputText value="#{calculBean.taux}"/> <br/>
    <h:commandButton value="Calcul!"
                      action="#{calculBean.calculer}"/>
  </h:form>
</f:view>
</body>
</html>
```

2. Configuration au sein de faces-config.xml

WEB-INF/faces-config.xml:

```
<faces-config>

  <managed-bean>
    <managed-bean-name>calculBean</managed-bean-name>
    <managed-bean-class>my_jsf.CalculBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>

  ...

</faces-config>
```

3. Code type d'un JavaBean JSF

CalculBean.java:

```
package my_jsf;

public class CalculBean {
    private Double montant;    private Double taux;
    private Double resultat; //NB: Double (avec null par défaut) mieux que double (0.0 par défaut)
                               //--> zone de texte vide (si rien de saisi) , mieux que 0.0

    public CalculBean() { super() ; }

    public String calculer() {    String suite=null;
        try{    resultat = ..... montant ..... taux ..... ; suite="page2" ;    }
        catch(Exception ex) { ex.printStackTrace(); }
        return suite;
    }

    public Double getResultat() { return resultat; }

    public Double getMontant() { return montant; }
    public void setMontant(Double montant) {    this.montant = montant; }

    public Double getTaux() { return taux; }
    public void setTaux(Double taux) {    this.taux = taux; }
}
```

==> pur JavaBean (style "POJO") sans héritage imposé .

4. Configuration via annotations (depuis JSF 2)

```
<?xml version="1.0"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee" version="2.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig\_2\_0.xsd">
    <!-- presque vide en V2 (si utilisation des annotations et navigations implicites) -->
</faces-config>
```

```
import javax.faces.bean.*;
@ManagedBean // nom par défaut = NomClasse avec première lettre en minuscule
@SessionScoped // ou bien @RequestScoped ou bien ...
public class CalculBean { ...
}
```

Remarque : @ManagedBean , @SessionScoped et @ManagedProperty (du package "javax.faces.bean") sont des annotations spécifiques à JSF2 .

Dans certains cas (selon le contexte), on pourra éventuellement préférer @Named, @Inject et @SessionScoped de (javax.inject et javax.enterprise.context) et liés à DI/CDI/JEE6 ou bien encore @Scope , @Autowired de Spring >= 2.5 .

5. Accès aux résultats pour affichage

resultats.jsp:

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

<html>
<head> <title>resultat calcul</title> </head>
<body>
<f:view>
  Pour un montant de
    <b><h:outputText value="#{calculBean.montant}"/></b>
  et avec un taux de
    <b><h:outputText value="#{calculBean.taux}"/></b>
  le résultat est de
    <b><h:outputText value="#{calculBean.resultat}"/></b>

  <br/>
  Affichage via JSP 2 (et non plus h:outputText) : <br/>
  Resultat = <b> #{calculBean.resultat} </b>
</f:view>
</body>
</html>
```

NB: La syntaxe `<h:outputText value="#{nomBean.propriété}"/>` est utilisable dans n'importe quelle page JSP ayant fait référence aux bibliothèques de tags de JSF.

Une telle jsp peut alors être prise en charge par la quasi totalité des serveurs aujourd'hui en production (Tomcat 4, WebSphere 4 et 5, JBoss 3, ...).

Par contre, l'utilisation directe de la syntaxe **JSP2** (`#{nomBean.propriété}`) n'est envisageable que si l'on est certain d'effectuer un déploiement sur un serveur récent (ex: Tomcat >=5, ...).

NB: L'utilisation de JSTL (`<c:out value="#{requestScope.nomBean.propriété}"/>`) ou des anciens standards (`<jsp:useBean ...>`, `<jsp:getProperty ...>`) est également envisageable.

L'essentiel est d'utiliser un style homogène sur l'ensemble des pages d'un projet.

Remarque très importante:

Les balises de JSTL (en `#{...}`) et de JSF (en `#{...}`) sont interprétées à des moments différents:

- * *interprétations des TagLib JSP* pour JSTL
- * *phase de rendu HTML* pour JSF

==> **L'imbrication de balises JSTL à l'intérieur de balises JSF ne fonctionne donc pas.**

Par contre, l'imbrication de balises JSF au sein de balises JSTL fonctionne quelquefois (à tester)

Le plus proche équivalent JSF de `<c:forEach />` est `<h:dataTable>`

Le plus proche équivalent JSF de `<c:if />` correspond à l'attribut `rendered="#{xxBean.prop ...}"` qui a pour effet d'activer ou désactiver le rendu d'un composant JSF en fonction d'une expression booléenne (condition vérifiée ou pas).

III - Navigations JSF (règles , contrôleur)

1. Principes généraux de la navigation JSF

1.1. Fichiers ".jsp" ou ".xhtml" mais URL en ".faces" ou ".jsf"

Bien que les **pages jsp de JSF** aient une **extension classique (.jsp ou .xhtml)** au niveau du code source, les **URL** qui doivent être utilisées au sein d'un site JSF doivent se terminer par **".faces" (ou ".jsf")** selon la configuration choisie au sein de **WEB-INF/web.xml** :

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

<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
....
```

==> vers page2 depuis page1

1.2. <h:form> ayant d'office method="POST" et action="" (self)

Un **formulaire JSF (<h:form>)** ne supporte pas de soumission (*submit*) en mode GET; le paramètre *method* est donc toujours implicitement fixé à **"POST"**.

D'autre part, le paramètre *action* de **<h:form>** est d'office fixé à **" "**. Ce qui a pour effet de **ré-invoquer l'URL actuelle** : Une page JSF se rappelle elle même très souvent (post-back).

1.3. Navigation paramétrée via <h:commandButton action="..." >

De façon à paramétrer la navigation attendue suite à la soumission d'un formulaire , on utilise généralement le paramètre **action** de **<h:commandButton >**

La valeur résultant (directement ou pas) de ce paramètre sera comparée aux valeurs des balises **<from-outcome>** d'un bloc **<navigation-rule>** / **<navigation-case>** du fichier **"faces-config.xml"** :

- Si le paramètre **action** de **<h:commandButton ..>** a une valeur prenant la forme d'une simple chaîne de caractères , cette valeur sera alors directement prise en compte au sein du fichier **"faces-config.xml"**.
- Si par contre la valeur de ce paramètre *action* est du type **"#{nomContrôleur.methode}"** , c'est alors le **résultat de la méthode appelée** qui sera prise en compte au sein du fichier central **"faces-config.xml"**.

Exemple de "commandButton" au sein d'un formulaire:

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

<html>
<head> <title>Param_xxx</title> </head>
<body>
<f:view>
    <h:form id="formXxx">
        Montant: .....</br>
        <h:commandButton value="Retour !" action="retour"/>
        <h:commandButton value="Calcul Xxx !"
                        action="#{myCalculController.calculer}"/>
    </h:form>
</f:view>
</body>
</html>
```

2. Configuration au sein de faces-config.xml

WEB-INF/faces-config.xml

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
    "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config>
    <managed-bean>...</managed-bean>
    ...
    <navigation-rule>
        <from-view-id>/pages/param_xxx.jsp</from-view-id>
        <navigation-case>
            <from-action>#{myCalculController.calculer}</from-action>
            <from-outcome>ok</from-outcome>
            <to-view-id>/pages/suite_xxx.jsp</to-view-id>
        </navigation-case>
        <navigation-case>
            <from-outcome>retour</from-outcome>
            <to-view-id>/pages/p1.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>

    </navigation-rule>
        <navigation-case>
            <from-outcome>accueil</from-outcome>
            <to-view-id>/pages/welcome.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
    ...
</faces-config>
```

- L'information encadrée par **<from-view-id>** correspond à la page actuelle coté serveur (liée au précédemment "submit" coté navigateur).
NB: Si la balise **<from-view-id>** n'est pas renseignée , la règle de navigation s'appliquera alors pour n'importe quelle page jsp/jsf de départ .
- La valeur d'une balise **<from-outcome>** est comparée à la valeur découlant directement du paramètre *action* de `<h:commandButton>` ou bien indirectement d'une méthode invoquée.
- L'information encadrée par **<to-view-id>** correspond à la page qui sera affichée (redirection interne via `rd.forward(req,resp)` = futur résultat immédiat de la navigation).
- La balise facultative **<from-action>** permet en outre de préciser à partir de quelle action la règle de navigation doit être évaluée (ceci est utile dans le cas d'une page comporte plusieurs formulaires et/ou plusieurs actions) .

3. Bean "mini contrôleur" pour navigation

Rappel: un formulaire JSF peut préciser une méthode d'aiguillage via une syntaxe du type `#{nomContrôleur.methodeAiguillage}` .

Exemple:

```
<h:commandButton value="Calcul Xxx !"
                 action="#{myCalculController.calculer}"/>
```

Pour que le mécanisme d'aiguillage fonctionne bien , il faut :

- **Coder la méthode d'aiguillage au sein d'une classe java (de type "JavaBean")**
- **Déclarer un nom de contrôleur associé à cette classe au sein de *faces-config.xml***

Exemple de code Java pour une classe "contrôleur" avec méthode d'aiguillage:

```
package pck;

public class MyCalculController {

    public String calculer()
    {
        String res=null;
        try{
            ....; res="ok";
        } catch(Exception ex) {
            ex.printStackTrace();
        }
        return res; // si return null ==> pas de navigation déclenchée
    }
}
```

Rappel : Déclaration d'un contrôleur (ou "Managed Bean") au sein de *faces-config.xml* :

```

...
<faces-config>
  <managed-bean>
    <managed-bean-name>myCalculController</managed-bean-name>
    <managed-bean-class>pck.MyCalculController</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
...

```

Remarques:

- La sous balise facultative **<redirect>** de **<navigation-case>** permet de demander une navigation pas redirection http (et pas comme un simple forward vers une page jsp). Ceci peut quelquefois être utile pour des navigations vers l'extérieur ou pour ne pas rencontrer de problème dans l'interprétation des chemins relatifs lorsque l'on souhaite naviguer vers une page d'un autre répertoire.
- Des navigations simples peuvent être déclenchées par de simples liens hypertextes (** vers page N **) . C'est plus direct mais moins souple.
- On peut éventuellement utiliser **<h:commandLink .../>** à la place de **<h:commandButton />** pour déclencher une navigation via un lien hypertexte inséré dans un formulaire. Les paramètres sont exactement les mêmes.

4. Navigations implicites depuis la version 2 de JSF

Depuis la version 2 de JSF , une navigation implicite est maintenant automatiquement activée:

Si une méthode de traitement/aiguillage au niveau d'un "ManagedBean" (généralement déclenchée par `commandButton`) retourne la chaîne de caractère "**pageX**" alors une navigation vers la page "**pageX.jsp**" ou bien "**pageX.xhtml**" (selon ce qui existe en relatif) est alors automatiquement effectuée.

==> conséquence , si l'on souhaite ne pas changer de page --> **"return null;"**

IV - Validations JSF et Contexte

Bon paramétrage des formulaires jsf

Bien que préciser l'id d'un composant JSF soit facultatif au niveau d'une page JSP ou XHTML , il est très fortement recommandé de **préciser l'id de toutes les zones JSF importantes pour les raisons suivantes**:

- un message d'erreur pourra être rattaché à une zone mal saisie (message qui s'affiche à coté)
- le message d'erreur par défaut sera plus compréhensible
- l'id d'une zone jsf sera constant/stable (et non pas variable / généré dynamiquement) et ceci permettra d'automatiser des tests de pages JSF avec des technologies telles que Selenium ou JMeter .

Exemple:

Si l'on oublie de préciser l'id d'un `<h:form >` , celui-ci se voit attribué un id dynamiquement (ex : `j_id_jsp_1850302668_1`) et les messages d'erreurs s'affichent par défaut comme ceci :

Montant: j_id_jsp_1850302668_1:montant: La donnée n'est pas un nombre valide.
 Nb années: j_id_jsp_1850302668_1:nbAnnees: La donnée n'est pas un nombre valide.
 Taux: j_id_jsp_1850302668_1:taux: La donnée n'est pas un nombre valide.

En spécifiant l'id du formulaire , le message d'erreur devient un peu plus parlant :

```
<h:form id="formEmprunt">
    Montant: <h:inputText id="montant" value="#{empruntBean.montant}" />
    <h:message for="montant" styleClass="RedCssClass"/> <br/>
    ....
```

Montant: formEmprunt:montant: La donnée n'est pas un nombre valide.

Ceci dit , l'affichage "`id_formulaire:id_champSaisi : raison_erreur`" est un peu *long/verbeux* .

On pourra préférer un **préfixe court et personnalisé** en ajoutant l'attribut `label="xxx"` au niveau du champ de saisie :

```
<h:form id="formEmprunt">
    Montant: <h:inputText id="montant" label="montant"
    value="#{empruntBean.montant}" />
    <h:message for="montant" styleClass="RedCssClass"/> <br/> ...
```

Montant: montant: La donnée n'est pas un nombre valide.

Avec ou sans détails:

`<h:messages />` affiche par défaut une liste de messages sans détails :

- Erreur de conversion
- Erreur de conversion
- Erreur de conversion

En ajoutant l'attribut `showDetail="true"` , `<h:messages .../>` affiche :

- Erreur de conversion j_id_jsp_2046816173_2:j_id_jsp_2046816173_3: La donnée n'est pas un nombre valide.
- Erreur de conversion j_id_jsp_2046816173_2:j_id_jsp_2046816173_4: La donnée n'est pas un nombre valide.
- Erreur de conversion j_id_jsp_2046816173_2:j_id_jsp_2046816173_5: La donnée n'est pas un nombre valide.

---> Encore une fois , sans id et/ou sans label , les messages sont peu compréhensibles !!!!

1. Validation automatique et implicite

En mode automatique :

- rien de fastidieux à coder au niveau du JavaBean .
- la classe du JavaBean peut directement encoder les propriétés sous formes numériques (int , double, ...).

Les erreurs de conversions (String ==> double , int , ...) remontent automatiquement sous forme de messages d'erreurs qu'il suffit d'afficher.

La balise `<h:messages .../>` affiche toute la liste (si non vide) des messages d'erreurs en mode résumé (par défaut avec `showDetail="false"` et `showSummary="true"`)

La balise `<h:message for="idXxx" .../>` affiche en mode détaillé (par défaut avec `showDetail="true"` et `showSummary="false"`) le message d'erreur associé à la zone de saisie dont l'*id* est précisé via l'attribut *for* .

Les balises `<h:messages />` et `<h:message />` acceptent un attribut facultatif *styleClass* qui permet de préciser la classe de style CSS qui sera utilisée pour mettre en forme un message d'erreur.

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

<html>
<head>
  <title>Param_calcul</title>
<LINK rel="stylesheet" href="../css/styles.css" type="text/css" />
</head>
<body>
<f:view>
  <h:form id="formXY" >
    Montant: <h:inputText id="montant" label="montant"
                  value="#{calculBean.montant}" />
    <h:message for="montant" styleClass="RedCssClass"/> <br/>

    Taux: <h:inputText id="taux" label="taux"
                  value="#{calculBean .taux}" />
    <h:message for="taux" styleClass="RedCssClass"/> <br/>

    Commentaire (required): <h:inputText id="commentaire"
                  label="commentaire" required='true'
                  value="#{calculBean .commentaire}" />
    <h:message for="commentaire" styleClass="RedCssClass"/> <br/>

    <h:commandButton value="Calcul" action="#{calculBean.calculer}" />
  </h:form>
</f:view>
</body>
</html>
```

NB: `required='true'` permet de préciser qu'un champ doit obligatoirement être renseigné .

css/styles.css:

```
.RedCssClass { color: #EE0000; }
```

====> exemple d'affichage en cas d'erreur de saisie:

Montant:

Nb années:

Taux: "taux": La donnée n'est pas un nombre valide.

Commentaire (required): "commentaire": Une donnée est requise.

NB: si `<h:messages />` et `<h:message for="id_zoneX" />` sont utilisés conjointement dans une même page, il faut ajouter **globalOnly="true"** à `<h:messages />` pour que les messages d'erreurs liés à des saisies erronées ne soient pas affichés deux fois et pour que `<h:messages globalOnly="true".../>` n'affiche que les erreurs associées à des exceptions (transformées en FacesMessages) au niveau des managedBeans.

```
<h:messages showDetail="true" globalOnly="true" styleClass="RedCssClass"/>
```

2. Validation automatique et explicite

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

<html>
<head>
  <title>Param_calcul</title>
<LINK rel="stylesheet" href="../css/styles.css" type="text/css" />
</head>
<body>
<f:view>
  <h:form id="formXY" >
    Montant (entre 100 et 999999):
    <h:inputText id="montant"
      label="montant" value="#{calculBean.montant}" >
      <f:validateDoubleRange minimum="100" maximum="999999"/>
    </h:inputText>
    <h:message for="montant" styleClass="RedCssClass"/> <br/>

    Taux (entre 0.01 et 0.25) :
    <h:inputText id="taux" id="taux" value="#{calculBean.taux}">
      <f:validateDoubleRange minimum="0.01" maximum="0.25"/>
    </h:inputText>
    <h:message for="taux" styleClass="RedCssClass"/> <br/>

    Commentaire (required , length >= 2):
    <h:inputText id="commentaire" required='true'
      label="commentaire" value="#{calculBean.commentaire}">
      <f:validateLength minimum="2"/>
    </h:inputText>
```

```

        <h:message for="commentaire" styleClass="RedCssClass"/> <br/>
<h:commandButton value="Calcul" action="#{calculBean .calculer}"/>
    </h:form>
</f:view>
</body>
</html>

```

====> exemple d'affichage en cas d'erreur de saisie:

Montant (entre 100 et 999999):

Nb années (entre 1 et 30):

Taux (entre 0.01 et 0.25) :

"taux": La donnée n'est pas comprise entre 0,01 et 0,25.

Commentaire (required , length >= 2):
 "commentaire": La donnée a moins que les 2 caractères maximum requis.

NB:

On peut également utiliser les balises `<f:convertXxx ...>` à la place de `<f:validateXxx />` car les balises `<f:convertXxx ...>` fonctionnent aussi bien en saisie/validation qu'en affichage :

```

<h:inputText value="#{xxxBean.montant}">
    <f:convertNumber maxFractionDigits="2"/>
</h:inputText>

```


3. Messages globaux / erreurs de traitement

Au sein d'une méthode d'action, certains traitements peuvent déclencher des exceptions (Pb de connexions à la base de données, exceptions métiers : règle de gestion non vérifiée,).

Pour remonter proprement un message d'erreur à destination de l'utilisateur, on peut éventuellement procéder de la façon suivante:

```
public String doActionXY() {
    String res=null;
    try{ .....    res="ok";
    } catch(Exception ex){
        ex.printStackTrace();    res="ko";
        FacesContext.getCurrentInstance().addMessage( null /* id = null pour message global */,
            new FacesMessage("message erreur" , e.getMessage());
    }
    return res;
}
```

et coté jsp:

```
<h:messages globalOnly='true' />
```

avec ou sans **showDetail="true"** sachant que les messages sont créés par
new FacesMessage("message erreur principal" , "détail du message")

NB: D'une manière très générale au sein de JSF, pour compenser le fait que les méthodes du "ManagedBean" n'aient pas de paramètres de type "HttpServletRequest" ni "HttpServletResponse", on accède indirectement au contexte JSF/HTTP via la méthode statique

FacesContext.getCurrentInstance() ;

On peut ainsi indirectement accéder aux objets "request" , "response" , "session" , "application/ServletContext" de l'api des servlet .

Ex: Object httpParamValue= **FacesContext.getCurrentInstance().getExternalContext().getRequestParameterMap().get(paramName);**

NB: le nom d'un paramètre http correspondant à un composant JSF est "id_Form:id_Composant" (à indirectement paramétrer via id="..." dans les pages jsp) .

4. Annotations compatibles JSF2 pour la validation

En théorie depuis la version 2 de JSF et en pratique avec certaines extensions pour JSF (ex : richFaces , primeFaces) on peut directement placer dans le code JAVA d'un managedBean (ou d'un "sous bean") les annotations @NotNull , @Min , @Max , @Size , @Pattern (du package "javax.validation.constraints") .

Ces annotations (standards en java/jee6) sont à considérées comme une alternative aux sous balises <f:validateXY .../>

Exemple:

```
@Size(min = 1, message = "Please enter the Email")
@Pattern(regexp = "[a-zA-Z0-9]+@[a-zA-Z0-9]+\\.[a-zA-Z0-9]+",
          message = "Email format is invalid.")
private String email;

@Size(min = 1, message = "Please enter a Username")
private String userName;
```

NB : "hibernate-validator.jar" (à récupérer par exemple avec maven) est souvent nécessaire en tant qu'implémentation effective de l'API "javax.validation" qui interprètent les annotations @Size , @Max , @Min , ...

5. Validations avancées (en annexes)

- Personnalisation des messages d'erreurs prédéfinis (.properties)
- Programmation de validateurs spécifiques (custom validator)

V - Facelet (templates)

1. Présentation des facelets pour JSF2

La technologie des "facelets" de JSF sert à paramétrer et réutiliser des **modèles/templates de mise en page** (pour que la plupart des pages d'un même site aient la même structure : pied de page , ...). Les "facelets" sont l'équivalent JSF des "tiles" de struts

Outre l'aspect "modèle/template de pages", la technologie des "facelets" impose une ré-écriture des pages ".JSP" en pages ".XHTML" (où seul l'entête des pages ".xhtml" est radicalement différent) .

Remarques (très importantes):

- La technologie "*facelets*" était à l'origine de *JSF 1* un *complément facultatif* (à ajouter en tant que ".jar" supplémentaire)
- Depuis la version 2 de JSF , la technologie "facelet" est déjà intégrée dans l'implémentation de JSF (pas besoin d'ajouter facelet...jar) .
Certaines fonctionnalités récentes de JSF2 ne fonctionnent qu'avec les facelets .
Certaines extensions récentes pour JSF2 (ex: primeFaces , richFaces 4) ne fonctionnent également qu'avec les facelets. Autrement dit, les facelets sont quasiment imposés par JSF2
- Attention à l'écriture rigoureuse d'un template JSF2 (h:head , h:body) sachant que certains assistants "eclipse" génèrent des templates pour JSF1 (incompatibles avec JSF2).

2. Pages ".xhtml" et plus de pages ".jsp"

La technologie des « facelets » s'est beaucoup inspirée du framework « Tapestry » et en a repris le principe fondamental suivant:

- Au lieu d'insérer directement des balises jsp spécifiques à un framework ou une api (ex: Struts , Jsf , Jstl) , on écrit une page (ayant idéalement l'extension « .xhtml ») et comportant essentiellement des balises html classiques (ex: form , input ,) .
- En ajoutant au sein des balises html l'attribut spécial **jsfc="nom_balise_jsf"** , un **préprocesseur** de pages « facelets / xhtml » peut ainsi générer l'équivalent d'une page JSP classique avec des taglibs spécifiques à jsf.

Exemple:

page_xy.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
...
<form jsfc="h:form" id="helloForm">
    ${msg.prompt} <input label="Name" jsfc="h:inputText" required="true"
                      id="name" value="#{person.name}" />
    <input type="submit" jsfc="h:commandButton" id="submit"
          action="greeting" value="Say Hello" />
</form>
... </html>
```

Avantages:

- Une syntaxe très proche du « pur html » et donc bien compréhensible par des infographistes peu habitués aux spécificités java/jsp/jsf.
- La phase de « pré-traitement » des pages « .xhtml » (heureusement complètement transparente vis à vis de la structure Java_EE) permet aux « facelets » d'introduire de nouvelles fonctionnalités spécifiques telles que les templates (modèles de mise en page).

Remarques (importantes):

- contrairement aux pages ".jsp", les pages ".xhtml" ne sont pas transformées en "servlet" mais ont un comportement et des performances similaires.
- Si les pages ".xhtml" sont toujours utilisées que coté "java/eclipse" (et pas par des infographistes), on peut tout à fait placer directement des "h:form" ou "h:inputText" dans la page :

page_xy.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"    >
...
<h:form id="helloForm">
    ${msg.prompt} <h:inputText label="Name" required="true"
                        id="name" value="#{person.name}" />
    <h:commandButton type="submit" id="submit"
                    action="greeting" value="Say Hello" />
</h:form>
...
</html>
```

3. Notion de "template"

3.1. modèle commun plutôt qu' inclusions dispersées

De façon à **décomposer une grande page HTML en de multiples petites régions** (ex: "bandeau d'entête" , "pied de page" , "menu sur le coté" , "zone principale") , on peut soit :

- **inclure** directement quelques *sous éléments* (.jsp , .html , ...)
- utiliser les «**facelets**» de JSF (*qui ressemblent un peu aux "Tiles" de STRUTS*)

L'**inclusion directe de sous pages "JSP"** (via `<jsp:include page="partie1.jsp" />` ou bien `<%@include %>`) est **plus souple mais nécessite un assez grand nombre de lignes dispersées dans tout un tas de pages JSP pour encoder ces inclusions** .

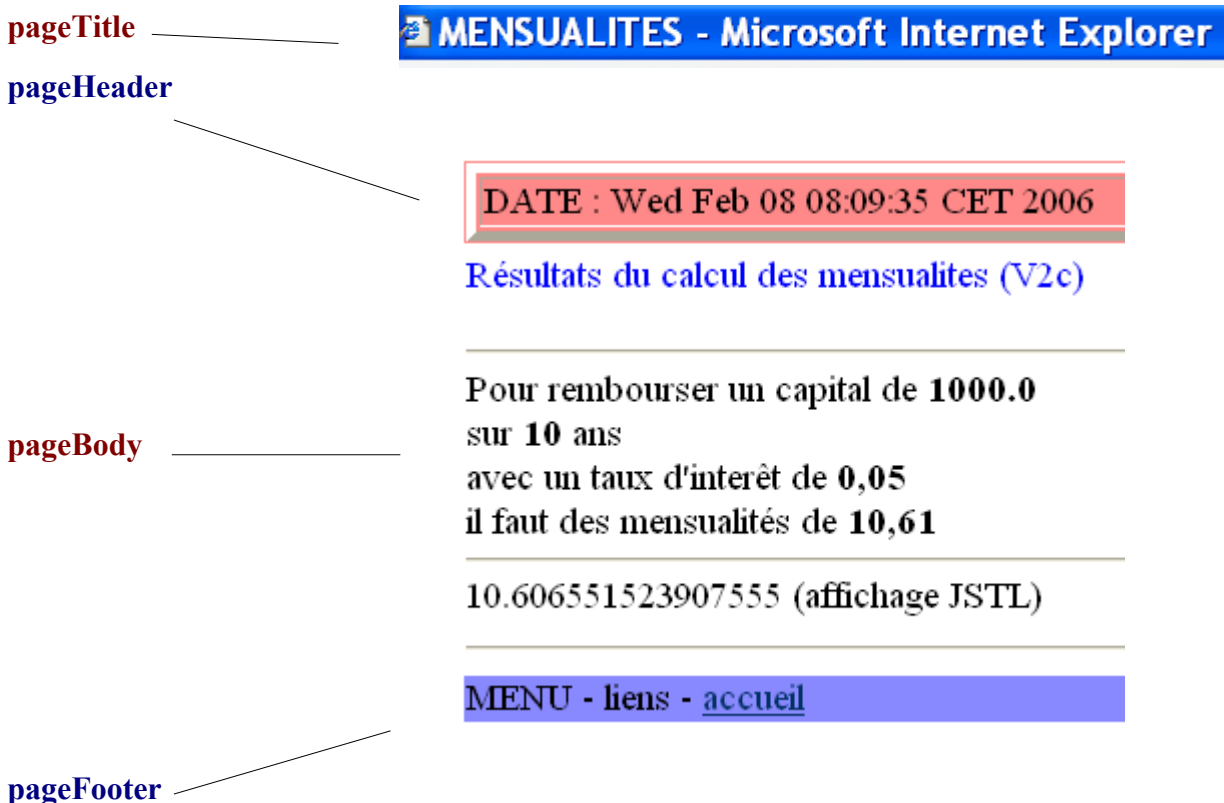
Effectuer une modification importante dans la mise en page oblige alors à retoucher un très grand nombre de pages JSP.

La mise en oeuvre des "Templates" de JSF/Facelet consiste à:

- **définir** (sous la forme d'une page JSP spéciale) **un modèle générique de mise en page** , appelé *template*, avec un *découpage en zones/régions* .
- spécifier des **définitions de pages**, basées sur ce modèle, en renseignant certaines **valeurs spécifiques** qui seront insérées dans certaines régions du modèle .

Avantage des "templates" de jsf/facelet ==> *Effectuer une modification importante de la mise en page d'un site Web ne consiste alors qu'a remettre à jour le modèle et quelques feuilles de styles.*

3.2. Vue d'ensemble schématique



4. Mise en oeuvre des "templates" de jsf/facelet

4.1. Constitution d'un template

templates/common.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<f:view>
<f:loadBundle basename="resources" var="msg" />
<h:head>
<title><ui:insert name="pageTitle">Page Title</ui:insert>
</title>
<style type="text/css">          ...</style>
</h:head>

<h:body bgcolor="#ffffff">
<table style="border:1px solid #CAD6E0" align="center" cellpadding="0"
      cellspacing="0" border="0" width="400">
<tbody>
  <tr>
    <td class="header" height="42" align="center" valign="middle"
        width="100%" bgcolor="#E4EBEB">
      <ui:insert name="pageHeader">Page Header</ui:insert>
    </td>
  </tr>
  <tr>
    <td height="1" width="100%" bgcolor="#CAD6E0"></td>
  </tr>
  <tr>
    <td width="100%" colspan="2">
      <table width="100%" style="height:150px" align="left"
            cellpadding="0" cellspacing="0" border="0">
        <tbody>
          <tr>
            <td align="center" width="100%" valign="middle">
              <ui:insert name="pageBody">Page Body
              </ui:insert>
            </td>
          </tr>
        </tbody>
      </table>
    </td>
  </tr>
  <tr>
    <td colspan="2" valign="bottom" height="1" width="100%"
        bgcolor="#CAD6E0"> ...</td>
  </tr>
</tbody>
</table>
</h:body>
</f:view>
</html>
```

NB : selon le(s) navigateur(s) plus ou moins récent(s) prévu(s) pour interpréter les pages on peut préférer un découpage sans tableau (avec des "div" accompagnées de styles CSS) ou bien encore un

découpage basé sur les nouveautés de HTML5 .

5. Définition d'une page s'appuyant sur un template

pages/xy.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <f:loadBundle basename="resources" var="msg" />

  <ui:composition template="/templates/common.xhtml">
    <ui:define name="pageTitle">Input User Name</ui:define>
    <ui:define name="pageHeader">F... Application</ui:define>
    <ui:define name="pageBody">
      <h:message showSummary="true" showDetail="false"
        style="color: red; font-weight: bold;" for="name" />
      <form jsfc="h:form" id="helloForm">
        ${msg.prompt}
        <input label="Name" jsfc="h:inputText"
          required="true" id="name" value="#{person.name}" />
        <input type="submit" jsfc="h:commandButton" id="submit"
          action="greeting" value="Say Hello" />
      </form>
    </ui:define>
  </ui:composition>
</html>
```

6. Configurations conseillées (pour facelets)

NB: configuration à adapter au cas par cas (selon versions)

WEB-INF/web.xml (*<context-param>* pour la plupart facultatifs / valeurs par défaut)

```
<?xml version="1.0"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <description>...</description>
  <display-name>....</display-name>
  <context-param>
    <param-name>facelets.REFRESH_PERIOD</param-name>
    <param-value>2</param-value>
  </context-param>
  <context-param>
    <param-name>facelets.DEVELOPMENT</param-name>
    <param-value>true</param-value>
  </context-param>

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
```



```

    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
</web-app>

```

NB : fixer à "true" la valeur du paramétrage **facelets.DEVELOPMENT** permet (en phase de développement) de tester immédiatement une nouvelle version d'une page ".xhtml" au sein d'un navigateur internet sans avoir à systématiquement arrêter/relancer tomcat pour que celle-ci soit rechargée.

index.jsp (page d'accueil):

```

<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html><head></head>
<body><a href="pages/welcome.jsf">welcome</a></body>
</html>

```

pour aller vers **pages/welcome.xhtml** (en passant par *FacesServlet*) .

NB : contrairement à l'ancien JSF1 , plus besoin d'ajouter "jsf-facelets.jar" dans WEB-INF/lib et plus besoin d'ajouter `<view-handler>com.sun.facelets.FaceletViewHandler</view-handler>` dans *faces-config.xml*.

Tout est déjà automatiquement intégré et paramétré au sein d'une implémentation de JSF2 (ex : MyFaces 2).

7. Templates évolués (avec sous templates)

layout/main.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <f:view contentType="text/html">
    <ui:insert name="pageMetaData"></ui:insert> <!-- pour mode "GET" de JSF2 -->
  </f:view>
  <h:head> ... </h:head>
  <h:body>
    <table style="border: 1px solid #CAD6E0" align="left" cellpadding="0" cellspacing="0"
    border="0" width="96%">
    <tbody>
      ...
      <!-- pied_page (pageFooter): -->
      <tr><td class="bottom" height="32" align="center" valign="middle" width="100%"
      bgcolor="#E4EBEB">
        <ui:include src="footer.xhtml">
          <ui:param name="appName" value="boutique"/>
        </ui:include>
      </td>
    </tr>
    </tbody>
  </table>
</h:body>
</f:view>
</html>
```

layout/footer.xhtml

```
<span xmlns:h="http://java.sun.com/jsf/html">
  <a href="mentions_legales.jsf">mentions legales</a> -
  ... -
  <a href="services" target="_new">services (web/cxf)</a>
  ... -
  <a href="pilotage.jsf">pilotage(admin) de #{appName}</a>
</span>
```

ou quelquefois <h:outputText value="#{appName}"/>

VI - Composants JSF standards (de base)

1. Éléments de base (zone de saisie, bouton , ...)

Les composants fondamentaux de JSF (et les tags associés) ont déjà été vus dans les chapitres précédents.

Rappels:

- `<h:inputText value="..." id="..." >` représente une **zone de saisie**
- `<h:commandButton value="..." action="..." >` représente un **bouton poussoir**
- `<h:outputText value="..." >` affiche un donnée de façon textuelle

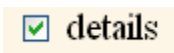
Variantes existantes pour les zones de textes:

```
<h:inputSecret value="#{monBean.password}" />
```

```
<h:inputTextArea ... />
```

2. Case à cocher simple

```
<h:selectBooleanCheckbox value="#{empruntBean.details}" /> details <br/>
```



3. Lien hypertexte

```
<h:outputLink value="http://myfaces.apache.org" >
  <h:outputText value="MyFaces Homepage"/>
</h:outputLink>
```

4. Zone de Liste (à sélection simple)

JavaBean avec liste des valeurs possibles:

```
import javax.faces.model.SelectItem;

public class ColorBean {
    ...
    private SelectItem[] listeDeCouleur = { new SelectItem("#FF0000","rouge") ,
                                             new SelectItem("#00FF00","vert") ,
                                             new SelectItem("#0000FF", "bleu") ,
                                             new SelectItem("#000000","noir") };

    // NB : la valeur de retour de getXxx() peut être de type Collection<SelectItem>
    public SelectItem[] getListeDeCouleur() { return listeDeCouleur; }
}
```

Le constructeur de *SelectItem* est surchargé :

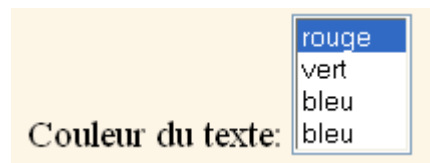
- une variante avec un seul paramètre ==> valeur = libellé affiché.
- une variante avec deux paramètres ==> valeur , libellé_différent_de_la_valeur

L'attribut **value** de `<f:selectItems />` permet de référencer une propriété d'un bean qui correspondra à l'ensemble des valeurs possibles proposées par la liste:

Couleur du texte:

```
<h:selectOneListbox value="#{myBean.foreground}">
    <f:selectItems value="#{colorBean.listeDeCouleur}" />
</h:selectOneListbox> <br/>
```

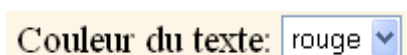
==>



avec en plus **size="1"** dans `<h:selectOneListbox ...>` on obtient une **liste déroulante** (*ComboBox*)

NB: On peut également obtenir directement une **liste déroulante** (*ComboBox*) en utilisant `<h:selectOneMenu >` à la place de `<selectOneListBox>`

==>



NB : depuis la version 2 de JSF, on peut directement utiliser `<f:selectItems .../>` à partir d'une collection d'objets quelconques via les attributs adaptateurs suivants :


```
<f:selectItems value="#{mbPlanning.clientContextList}"
               var="c" itemValue="#{c.id}" itemLabel="#{c.name}"/>
```

Autrement dit, il n'est plus toujours nécessaire de préparer une méthode `getValeursSelectionnablesXy()` retournant spécifiquement une Collection de `SelectItem` .

Indiquer directement certaines valeurs possibles fixes d'une liste:

```
Couleur du fond:
<h:selectOneMenu value="#{colorBean.background}">
  <f:selectItem itemValue="#FFFFFF" itemLabel="blanc"/>
  <f:selectItem itemValue="#AAAAAA" itemLabel="gris"/>
</h:selectOneMenu> <br/>
```

==>

Couleur du fond: 

5. Liste de bouton radios exclusifs (SelectOneRadio)

```
Couleur du fond:
<h:selectOneRadio value="#{colorBean.background}">
  <f:selectItem itemValue="#FFFFFF" itemLabel="blanc"/>
  <f:selectItem itemValue="#AAAAAA" itemLabel="gris"/>
</h:selectOneRadio> <br/>
```

==>

Couleur du fond:
☒ blanc ☐ gris

Remarque:

L'encodage jsp est quasiment le même que dans le cas d'une liste déroulante .

Le UIComponent est ici de la même famille "SelectOne".

Seuls les types de rendus varient : "Menu" ou "Radio"

6. Alignement via h:panelGrid

```
<h:panelGrid id="subPanel_list_clientContext" columns="2">
  <h:outputLabel for="ClientTitleId" value="title:" />
  <h:inputText id="ClientTitleId" value="#{mbPlanning.currClient.title}" />
  <h:....comp_l2_c1 ./> <h:....comp_l2_c2 ./>
</h:panelGrid>
```

-->même effet qu'un tableau invisible avec colonne de libellé et colonne de saisie .

7. Sélections multiples (via cases à cocher ou liste)

Sports pratiqués:

```
<h:selectManyListbox value="#{sportBean.listeSel}" >
  <f:selectItems value="#{sportBean.listeDeSport}"/>
</h:selectManyListbox> <br/>
```



Sports pratiqués:

```
<h:selectManyCheckbox value="#{sportBean.listeSel}" >
  <f:selectItems value="#{sportBean.listeDeSport}"/>
</h:selectManyCheckbox> <br/>
```

Sports pratiqués:

☐ Foot ☒ VTT ☐ Natation ☒ Ski

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

```
import javax.faces.model.SelectItem;
```

```
public class SportBean {
```

```
    private List listeSel =null; // liste des éléments sélectionnés
```

```
    private SelectItem[] listeDeSport = { new SelectItem("Foot") , new SelectItem("VTT") ,
                                           new SelectItem("Natation") , new SelectItem("Ski") };
```

```
    public List getListeSel() { return listeSel; }
```

```
    public void setListeSel(List listeSel) { this.listeSel = listeSel; }
```

```
...}
```

8. Tableaux (<h:dataTable> , <h:column>)

La combinaison <h:dataTable> <h:column> sert à construire un **tableau** en fonction d'une **collection indexée (ou tableau) d'objets**.

Attention: une collection de type java.util.Set est incompatible avec h:dataTable (car non indexée)

Chaque colonne correspondra à une des propriétés d'un objet de la collection.

Il y aura autant de lignes que d'objets dans la collection.

Syntaxe générale:

```
<h:dataTable value="#{unBean.uneCollection}"
              var="rowVar" border="1">
  <h:column>
    <h:outputText value="#{rowVar.champ1ObjCol}"/>
  </h:column>
  <h:column>
    <h:outputText value="#{rowVar.champ2ObjCol}"/>
  </h:column>
  ...
</h:dataTable>
```

Détails:

- L'attribut **value** de <h:dataTable> peut référencer une **collection** (liste , vecteur ,) , un **tableau** ou bien encore un **ResultSet** (jdbc) ou un **DataModel** (javax.faces.model).
- L'attribut **value** de <h:outputText> d'une colonne peut référencer une propriété existante d'un élément (objet/Bean) de la collection ou bien un objet complet de la collection (ex: instance de la classe Double ayant une méthode toString()).
==> value="#{rowVar.num}" , value="#{rowVar.libelle}" ou value="#{rowVar}"

Ajout d'entête (ou de pied) de colonne:

==> insérer dans <h:column> une sous balise <f:facet> avec name="header" ou "footer":

```
...
<h:column>
  <f:facet name="header">
    <f:verbatim>numero</f:verbatim>
  </f:facet>
  <h:outputText value="#{rowVar.num_mois}"/>
</h:column>
...
```

NB:

- En règle générale un sous élément de type <f:facet> de JSF rajoute des détails à l'élément parent (ici h:column).
- Le contenu HTML de <f:facet ...> peut être généré via <f:verbatim> ou bien <h:outputText ...>

Alternance des couleurs sur les lignes:

numero	reste à rembourser
n° 1	59 490,4
n° 2	58 978,69
n° 3	58 464,84
n° 4	57 948,84
n° 5	57 430,7

css/styles.css

```
.HEADING {
font-family: Arial, Helvetica, sans-serif;
font-weight: bold; font-size: 24px; color: black;
background-color: silver; text-align: center;
}

.ROW1 {
font-family: Arial, Helvetica, sans-serif;
font-size: 18px; color: black;
background-color: white; text-indent: 10px;
}

.ROW2 {
font-family: Arial, Helvetica, sans-serif;
font-size: 18px; color: white;
background-color: black; text-indent: 10px;
}
```

```
<head> ...
<LINK rel="stylesheet" href="../css/styles.css" type="text/css" />
</head>
...
<h:dataTable value="#{xxxBean.listeYyy}" var="rowVar" border="1"
             headerClass="HEADING" rowClasses="ROW1,ROW2" >
```

Eventuel ajout d'un contenu HTML en tant préfixe ou suffixe aux données :

```
...
<h:column>
    <f:verbatim> n° </f:verbatim>
    <h:outputText value="#{rowVar.num}" />
</h:column>
...
```

Remarques générales sur <h:dataTable> :

- h:dataTable avec border="0" permet de générer un tableau invisible (simple boucle)
- il est possible d'imbriquer des h:dataTable les uns dans les autres

VII - Événements JSF

1. Catégories d'événements et gestionnaires

événements JSF	Spécificités
Action Controller	<p>Pour gérer des événements servant à déclencher des traitements en arrière plan (éventuellement suivis d'affichage(s)).</p> <ul style="list-style-type: none"> • Toujours associé à un submit en HTTP • Le déclenchement de la méthode d'aiguillage a lieu après le peuplement (et la validation) du JavaBean . • la méthode déclenchée n'a pas d'objet Event en paramètre.
Event Listener	<p>Pour gérer des événements prévus pour demander à modifier l'apparence de l'interface utilisateur (Sous zone visible ou pas , grisée ou non , ...)</p> <ul style="list-style-type: none"> • pas systématiquement associé à un submit • intervient souvent avant le peuplement (et la validation) d'un JavaBean • n'affecte pas directement la navigation • la méthode déclenchée a un objet Event en paramètre.

Les "action controller" ont déjà été vus dans les chapitres précédents (**JavaBean** avec **méthode d'aiguillage** dont le nom est mentionné par l'attribut **action** de `<h:commandButton .../>`) .

NB: l'attribut **action** peut être situé sur l'un des 3 éléments suivants:

- `<h:commandButton value="..." .../>`
- `<h:commandButton image="..." .../>`
- `<h:commandLink .../>`

2. Gestionnaires d'événements (listener)

NB: A l'origine du framework , la version 1 de JSF ne prenait pas en compte ajax et tout le code qui gérait un événement était entièrement coté serveur en java et une page entière était refabriquée (avec souvent qu'une petite différence avec l'ancienne page).

Depuis la version 2 de JSF , ces mêmes listeners "java" peuvent être invoqués de manière plus optimisée via **f:ajax** (ou **p:ajax** de primefaces) et les mécanismes internes de JSF2 ne véhiculent alors vers le navigateur que la partie de la page à réactualiser.

2.1. Sous catégories de gestionnaires d'événements

événements JSF	Spécificités
ActionListener	<ul style="list-style-type: none"> • Toujours associé à un submit en HTTP ou bien à un événement ajax . • Déclenchement identique à un "Action Controller" mais attribut actionListener et non pas action .

événements JSF	Spécificités
ValueChangeListener	<ul style="list-style-type: none"> pas systématiquement associé à un submit déclenchement sur des zones de saisies, des cases à cocher, des boutons radios ou des listes déroulantes (sélections).

2.2. Action Listeners

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
...<body>
<f:view>
  <h:form>
    <h:commandButton value="Activer/désactiver choix couleurs"
      actionListener="#{colorBean.toggleColorSupport}"
      immediate="true"/><br/>
    Couleur fond : <h:inputText value="#{colorBean.background}"
      disabled="#{!colorBean.colorDetails}" /><br/>
    Couleur texte : <h:inputText value="#{colorBean.foreground}"
      disabled="#{!colorBean.colorDetails}" /><br/>
    ...
    <h:commandButton value="Submit" action="#{colorBean.suite}"/>
  </h:form>
</f:view>
</body></html>
```

NB:

- Tout type de champ JSF comporte un attribut **disabled** permettant de le désactiver (mode grisé) --> ce paramétrage (basé sur un booléen) tiendra indirectement compte du nouvel état du JavaBean après le traitement de l'événement.
- L'attribut **immediate** valué à **"true"** permet de demander un déclenchement de la méthode événementielle avant un peuplement (et la validation) du JavaBean.

```
import javax.faces.event.ActionEvent;
// JavaBean à idéalement déclarer en scope=session !!!
public class ColorBean {
  private String background;    // + public get/set
  private String foreground;    // + public get/set
  private boolean colorDetails; // + public is/set
  ...
  public ColorBean() { background="white"; foreground="black"; colorDetails=true; }
  public void toggleColorSupport(ActionEvent event) {
    colorDetails = !colorDetails;
    System.out.println("New state (after toggle / action Listener):" + colorDetails);
  }
  ... }
}
```

Passage de paramètres/attributs via ActionEvent:

```
<h:commandButton actionListener="#{listeCptBean.listenDetailsCompte}" value="details">
  <f:attribute name="idCpt" value="#{compte.numero}" />
</h:commandButton>
```

....

```

public void listenDetailsCompte(ActionEvent event){
    Object attrIdCpt= event.getComponent().getAttributes().get("idCpt");
    if(attrIdCpt!=null)
        System.out.println("attrIdCpt: " + attrIdCpt.toString());
}

```

2.3. Event Listeners

```

<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
...<body> <f:view>
    <h:form>
        <h:selectBooleanCheckbox value="#{colorBean.colorDetails}"
            immediate="true" onclick="submit()"
            valueChangeListener="#{colorBean.onIsDetailsChange}" />
        choix des couleurs. <br/>
        *** Couleur fond :
        <h:inputText value="#{colorBean.background}"
            disabled="#{!colorBean.colorDetails}" /><br/>
    </h:form>
</f:view> </body></html>

```

NB:

- Le submit http n'est pas automatiquement déclenché sur un *valueChangeListener* .
- Le code javascript *onclick="submit()"* déclenche ici explicitement un submit http (ce qui n'est utile qu'en ancienne version "non ajax")
- Pour un déclenchement via ajax , on n'ajoute pas *onclick="submit()"* mais l'on imbrique la sous balise *<f:ajax event="click or change or keyup or"*
render="id_zone_a_reactualiser" ... />

```

package my_jsf;
import javax.faces.event.ValueChangeEvent;

public class ColorBean {
    private String background;    // + public get/set
    private String foreground;    // + public get/set
    private boolean colorDetails; // + public is/set
    public ColorBean() { background="white"; foreground="black"; colorDetails=true; }
    ...
    public void onIsDetailsChange(ValueChangeEvent event)
    {
        colorDetails = ( (Boolean) event.getNewValue()).booleanValue();
        System.out.println("New state (after Value Change Listener on checkbox):" + colorDetails);
    }
}

```

NB: Un objet de type *ValueChangeEvent* comporte (entre autres) les méthodes suivantes:

- Object *getNewValue()*
- Object *getOldValue()*
- *getComponent()*

VIII - Principaux apports de JSF2

1. Gestion du mode "GET" (depuis JSF2)

ViewParameter dans <f:view> dans .xhtml

```
<f:metadata>
  <f:viewParam name="param1" value="#{bean.prop1}"/>
</f:metadata>
```

Ceci permet d'initialiser automatiquement la propriété "prop1" du managedBean "bean" via la valeur du paramètre http "param1" qui peut être véhiculé en mode "GET" via une URL du type *page1.jsf?param1=valeur1*.

NB : Le fait de pouvoir déclencher une *page JSF2 en mode GET* permet à l'internaute la création de "bookmark" / "signet" dans les favoris de son navigateur.

Il existe aussi `<f:event type="preRenderView" listener="#{bean.doSomething}"/>` que l'on peut placer dans <f:metadata> si l'on veut déclencher un traitement d'initialisation assez tôt (en mode événementiel).

Attention : ces nouveautés JSF2 ne fonctionnent qu'au sein de fichier ".xhtml" (avec xmlns) et ne fonctionnent pas dans un fichier ".jsp" (avec taglib).

Exemple :

déclenchement (depuis *listeComptes.xhtml*) :

```
...
<h:dataTable var="cpt" border="2" value="#{clientComptes.listeComptes}">
  <h:column>...</h:column>  <h:column>...</h:column>
  <h:column>
    <f:facet name="header"><f:verbatim>détails</f:verbatim></f:facet>
    <a href='operations.jsf?numCptSel=${cpt.numero}'> dernières opérations</a>
  </h:column>
</h:dataTable>
...
```

operations.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">

  <ui:composition template="/templates/common.xhtml">
    <ui:define name="pageMetaData">
      <f:metadata>
        <f:viewParam name="numCptSel" value="#{operations.numCpt}"/>
        <f:event type="preRenderView" listener="#{operations.initOperations}"/>
      </f:metadata>
    </ui:define>
    <ui:define name="pageTitle">operations</ui:define>
    <ui:define name="pageHeader">Dernieres operations</ui:define>
```

```

<ui:define name="body">
    <h3>compte <h:outputText value="#{operations.numCpt}" /> </h3>
    <hr/>
    <h:dataTable var="op" border="2" value="#{operations.listeOperations}">
        ... </h:dataTable>
</ui:define>
</ui:composition>
</html>

```

Operations.java

```

import javax.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;
...
//@ManagedBean
@Named
@RequestScoped //javax.context (CDI/JSR299/JEE6) plutot que javax.faces.bean (JSF2 only)
public class Operations {
    private Long numCpt;
    private List<Operation> listeOperations;
    ...

    /*version mode GET (JSF2 , facelet) avec méthode d'initialisation (event / preRenderView)*/
    public void initOperations(ComponentSystemEvent event){
        //System.out.println("isPostBack:"+FacesContext.getCurrentInstance().isPostBack());
        //System.out.println("isAjaxRequest:")
        //      +FacesContext.getCurrentInstance().getPartialViewContext().isAjaxRequest());
        //NB: ajaxRequest is a (partial) postBack
        try { System.out.println("initOperations (mode GET / JSF2 / facelet) , numCpt="+numCpt);
            //numCpt initialized by <f:viewParam name="numCptSel" value="#{operations.numCpt}" />
            listeOperations=getServiceGestionComptes().getOperationsOfCompte(numCpt);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Possibilité de déclencher des requêtes en mode "GET" via <h:button ...> ou <h:link ...> selon le paramètre outcome comparé à une partie d'une règle de navigation de faces-config.xml .

```

<h:link outcome="success">
    <f:param name="foo" value="bar"/>
</h:link>

```

2. Autres apports de JSF2

- Annotations @ManagedBean , @ManagedProperty, @SessionScoped ,
Attention, attention : Selon le contexte , on pourra choisir @SessionScoped, @RequestScope et @ApplicationScope en version "*javax.faces.bean*" (spécifique à *JSF2*) ou bien en version "*javax.enterprise.context*" (de *JEE6 / JSR299/ CDI*) .
 Quand c'est possible, l'utilisation de CDI (avec @Inject) est mieux que @ManagedProperty .
- navigations par défaut (return "xy" --> xy.jsp) , prise en compte d'ajax ,

2.1. Prise en compte d'ajax dans le standard JSF2

Bien que la prise en charge d'ajax soit déjà opérationnelle dans certaines extensions pour JSF1 (ex : richFaces3), la version 2 de JSF a maintenant introduit une prise en charge "standard" pour ajax.

Certaines extensions pour JSF2 (ex : richFaces4, primeFaces 3.2, ...) pourront éventuellement étendre ce standard (via quelques ajouts/améliorations).

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head> </h:head>
  <h:body>
    <h3>JSF 2.0 + Ajax Hello World Example</h3>
    <h:form>
      <h:inputText id="name" value="#{helloBean.name}"></h:inputText>
      <h:commandButton value="Welcome Me">
        <f:ajax execute="name" render="output" />
      </h:commandButton>
      <h2><h:outputText id="output"
        value="#{helloBean.sayWelcome}" /></h2>
    </h:form>
  </h:body>
</html>
```

NB :

- Ceci permet de rafraîchir la zone dont l'id est "output" via une requête ajax gérée automatiquement par le framework JSF2 (sans provoquer un rechargement complet de la page dans le navigateur). [partial processing]
- Le paramétrage execute="id1" (ou "id1 id2 id3") permet de préciser quelles sont les valeurs (potentiellement nouvellement saisies) à envoyer au serveur pour calculer la réponse à générer/récupérer/afficher.
- Si la zone à rafraîchir est au dehors du formulaire courant alors syntaxe en render=":idOutput" ou bien render=":xxx:yyy".
- Possibilité de préciser l'événement déclencheur, exemple :

```
<h:inputText value="#{bean.text}" >
  <f:ajax event="keyup" render="output" />
</h:inputText>
```

Valeurs spéciales pour attribut render ou execute (ou dans version richFaces ou primeFaces) :

@all	Toute la vue
@none	Aucun rendu ou aucun composant pris en compte si nouvelle(s) valeur(s)
@this	Composant déclenchant la requête ajax
@form	Tous les composants du formulaire courant

IX - Extensions JSF (tomahawk , richFaces, ...)

1. Principales extensions pour JSF (aperçu)

NB: les extensions suivantes sont concurrentes --> n'en choisir généralement qu'une !!!

Extensions JSF	Caractéristiques
Tomahawk (apache)	Composants graphiques évolués (Calendrier, Onglets,) sans ajax
iceFaces	Composants graphiques évolués (Calendrier, Onglets,) avec extension ajax
richFaces (jboss labs)	Composants graphiques évolués (Calendrier, Onglets,) avec extension ajax très simple à paramétrer/utiliser
primeFaces	<p><i>Extension plus récente (JSF 2 seulement) .</i></p> <p>A peu près les mêmes fonctionnalités que RichFaces 4 mais un peu plus "light" et encore plus riche (graphiquement).</p> <p><i>PrimesFaces</i> existe en versions "web ordinaire" et "mobile".</p>
autres	Site web " <i>jsfMatrix</i> "

+ extension complémentaire "facelet" maintenant intégrée dans JSF2 .

2. richFaces

Attention, attention:

richfaces 4 (pour JSF2) a beaucoup évolué par rapport à **richfaces 3 (pour JSF 1.x)** .

Principales différences (nouveau de richfaces 4):

- plus (absolument) besoin de configuration supplémentaire dans web.xml
- le fichier "templates/common.xhtml" doit comporter `<f:view> <h:head> ... </h:head> <h:body> </h:body> </f:view>` pour le bon fonctionnement de richFaces 4
- `<a4j:ajax render="...">` de richfaces 4 remplace l'ancien `<a4j:support reRender>` de la v3

2.1. Namespace/taglib de richFaces4

`xmlns:rich="http://richfaces.org/rich"`

`xmlns:a4j="http://richfaces.org/a4j"`

2.2. Eventuelle utilisation du plugin "Jboss-Tools" dans eclipse

En installant le gros plugin "Jboss-Tools" dans eclipse on bénéficie d'un paquet d'assistants évolués permettant de simplifier l'écriture des pages jsp pour jsf.

---> drag & drop de composants JSF (et richFaces)

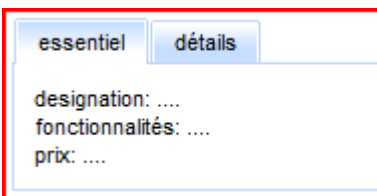
---> paramétrage des propriétés

2.3. Onglets (RF4)

Il suffit d'imbriquer des `<rich:tab>` au sein d'un `<rich:tabPanel>`

exemple:

```
<f:view>
  <h1>test onglets </h1>
  <hr/>
  <rich:tabPanel switchType="client" width="80%">
    <rich:tab header="essentiel">
      designation: ....<br/>
      fonctionnalités: ....<br/>
      prix: ....<br/>
    </rich:tab>
    <rich:tab header="détails">
      poids: ....<br/>
      options: ....<br/>
      ....<br/>
    </rich:tab>
  </rich:tabPanel>
</f:view>
```



2.4. Menus déroulants (RF4)

NB:

- il est essentiel de bien respecter l'imbrication suivante:
rich:menuItem dans rich:dropDownMenu dans rich:toolbar dans h:form .
- `<rich:menuSeparator />` et `<rich:menuGroup>` sont des éléments
- 4 types d'activations (submitMode):
 - ajax ---> `action="#{commonDataBean.doAjaxActionX}"`
 - server ---> `action="#{commonDataBean.doActionY}"`
 - none ---> pas d'action mais imbrication de `<h:outputLink >` , client → code js

Exemple (pages/test_menu.jsp) :

```
....
<f:view>
  <h1>test menu </h1>
  <h:form>
    <rich:toolbar>
      <rich:dropDownMenu>
        <f:facet name="label"> <h:outputText value="liens internes" /></f:facet>
        <rich:menuItem submitMode="none">
          <h:outputLink value=" ../index.jsp">page accueil</h:outputLink>
        </rich:menuItem>
        <rich:menuSeparator />
      </rich:dropDownMenu>
    </rich:toolbar>
  </h:form>
</f:view>
```

```

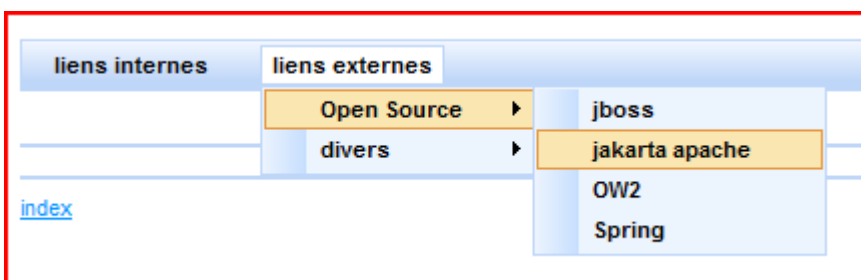
        <rich:menuItem submitMode="ajax" value="action_ajax_x"
                      render="zone_msg"
                      action="#{commonDataBean.doAjaxActionX}" />
        <rich:menuSeparator />
        <rich:menuItem submitMode="server"
                      value="action_server_y"
                      action="#{commonDataBean.doActionY}" />
    </rich:dropDownMenu>
    <rich:dropDownMenu >
        <f:facet name="label"> <h:outputText value="liens externes" /></f:facet>
        <rich:menuGroup label="Open Source">
            <rich:menuItem submitMode="none">
                <h:outputLink value="http://www.jboss.org">
                    jboss</h:outputLink>
            </rich:menuItem>
            <rich:menuItem submitMode="none">
                <h:outputLink value="http://jakarta.apache.org">
                    jakarta apache</h:outputLink>
            </rich:menuItem>
        </rich:menuGroup>
        <rich:menuGroup label="divers">
            <rich:menuItem submitMode="none">
                <h:outputLink value="http://www.google.fr/">
                    google (recherche)</h:outputLink>
            </rich:menuItem>
        </rich:menuGroup>
    </rich:dropDownMenu>
</rich:toolbar>
</h:form>
<hr/>
    <h:outputText id="zone_msg" value="#{commonDataBean.msg}" />
</f:view>

```

```

public String doAjaxActionX(){
    if(this.msg==null)
        this.msg="X";
    else this.msg = this.msg + "X";
    return null;
}

```



2.5. rich:Calendar

Aug 30, 2007 

optionalHeader Facet

<< < **August, 07** > >>

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
31	29	30	31	1	2	3	4
32	5	6	7	8	9	10	11
33	12	13	14	15	16	17	18
34	19	20	21	22	23	24	25
35	26	27	28	29	30	31	1
36	2	3	4	5	6	7	8

Aug 30, 2007 (x) Today

<rich:calendar popup="false"/>

```
<rich:calendar id="date" value="#{bean.dateTest}">
  <a4j:ajax event="dateselected" render="mainTable"/>
</rich:calendar>
```

2.6. Ajax4Jsf (A4j) pour richfaces 4 / JSF2

Categorie(s)	Produit(s) (de la catégorie sélectionnée)	Détails du produit sélectionné
livre CD informatique	CD1 CD2 CD3	CD2 , 76.00668901588568

..... categorie:

```
<h:selectOneListbox value="#{produitsJsfBean.categorie}" >
  <a4j:ajax event="click" render="list_prod"/>
  <f:selectItems value="#{produitsJsfBean.categorieSelectionnables}" />
  <!-- pour appel à getCategorieSelectionnables() -->
</h:selectOneListbox>
```

produits:

```
<h:selectOneListbox value="#{produitsJsfBean.produit}" id="list_prod">
  <f:selectItems value="#{produitsJsfBean.produitSelectionnables}" />
  <!-- pour appel à getProduitSelectionnables() -->
</h:selectOneListbox> <br/>
```

...

3. PrimeFaces

L'extension "**PrimeFaces**" est plus récente .

Elle a été mise en oeuvre spécifiquement pour la version 2 de JSF.

Primefaces est globalement :

- **très riche en composants graphiques prédéfinis** (encore mieux que richFaces)
- *d'assez bonne qualité* (certains composants sont mieux que ceux de richFaces , d'autres moins bien)
- **assez light** (assez peu de ".jar" à ajouter)
- déclinée en **deux versions** (pour ihm *web/ordinaire* et *mobile*) .

Plein d'exemples (avec code + rendu) sur le site de PrimeFaces :

<http://www.primefaces.org/showcase-labs/ui/home.jsf>

3.1. Namespace/taglib de primefaces

xmlns:p="<http://primefaces.org/ui>"

3.2. Dépendance maven principale pour primefaces

```
<dependency>
  <groupId>org.primefaces</groupId>
  <artifactId>primefaces</artifactId>
  <version>4.0</version>
</dependency>
```

3.3. Quelques exemples "primefaces"

```
<h:form >
date: <p:calendar id="date" value="#{withDateBean.date}" /> <br/>
...
pays: <p:autoComplete completeMethod="#{generalBean.completePays}"
      value="#{generalBean.pays}" /> <br/>
...
</h:form>
```

avec

```
public List<String> completePays(String p){
    List<String> suggestions = new ArrayList<String>();
    String uP = p.substring(0, 1).toUpperCase();
    if(p.length()>1) uP+=p.substring(1);
```

```

    for(String s : listePays){
        if(s.startsWith(uP))
            suggestions.add(s);
    }
    return suggestions;
}

```

June 2014						
Su	Mo	Tu	We	Th	Fr	Sa
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

suggestion (with ajax)

pays: f

texte:

onglets (primefaces) :

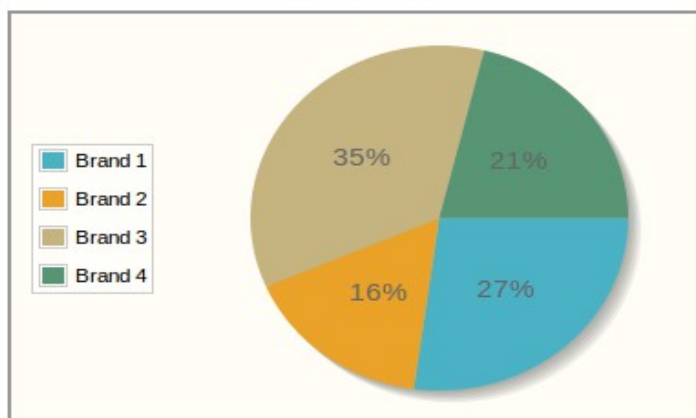
```

<p:tabView id="tabViewXy">
  <p:tab title="onglet1">
    ...
  </p:tab>
  <p:tab title="onglet2">
    ...
  </p:tab>
</p:tabView>

```

lineChart barChart pieChart

Sample Pie Chart



ANNEXES

X - Annexe – Validations avancées

Différents styles pour les validations

Type de validation	Spécificités
Manuelle/Applicative	Propriétés du JavaBean codées sous forme de String Validation à encoder dans les méthodes setXxx() Retourner null en cas d'erreur dans la méthode d'aiguillage Construire ses propres messages d'erreurs
Automatique et implicite	Utiliser des types éventuellement numériques (int, double) au sein du JavaBean Préciser required='true' si nécessaire au niveau du formulaire. Les messages sont automatiquement construits et remontés en cas d'erreur de conversion ou de manque (zone non renseignée). Il suffit d'afficher les éventuels problèmes via <h:message />
Automatique et explicite	Mode automatique basé sur le mode implicite et amélioré en explicitant certaines contraintes (coté formulaire) via les balises <f:validateLength /> , <f:validateDoubleRange .../> et <f:validateLongRange ... />
Avec extensions personnalisées	Il est possible de personnaliser certains messages. On peut également programmer son propre validateur (à enregistrer dans faces-config.xml) en implémentant l'interface Validator et en redéfinissant validate() .

1. Validation manuelle (libre mais longue)

==> cette méthode ne devrait idéalement être utilisée qu'en dernier recours (lorsque les solutions automatisées prises en charge par le framework) ne suffisent pas (cas pointus , validation consistant à comparer plusieurs propriétés (date_debut < date_fin)).

En mode "validation manuelle" , il faut (au niveau du JavaBean):

- encoder la plupart des propriétés sous forme de chaînes de caractères et *tenter d'effectuer soi même des conversions* (==> int , double , ...)
- Fabriquer une collection de messages d'erreurs qui sera retournée telle quelle ou bien sous la forme d'une grande chaîne .

2. Eventuelle personnalisation des messages JSF

```
..
javax.faces.component.UIInput.CONVERSION=Illegal format!
javax.faces.component.UIInput.REQUIRED=Missing value!
...
```

==> Messages à personnaliser au sein de **Messages(_fr).properties**

NB: le fichier *Messages_fr.properties* est généralement placé au sein du .jar de l'implémentation de JSF (ex: *myfaces-impl-....jar* de MyFaces avec *javax.faces.Messages.properties*)

On peut effectuer une copie de ce(s) fichier(s) dans le code de l'application web (ex: `src/jsf_resources/Messages[_fr].properties`) pour ensuite peaufiner un peu les messages d'erreurs (meilleurs traductions,).

Pour que JSF tienne compte du nouveau fichier de ressources il faut le spécifier de la façon suivante dans *faces-config.xml* :

```
<application>
...
  <message-bundle>package_xxx.ResourceBaseName</message-bundle>
</application>
```


3. Valdateur spécifique (custom validator)

- 1) programmer une classe implémentant l'interface `Validator` et sa méthode `validate()`
`public class MyDoubleValidator implements Validator {`

```

    public void validate(FacesContext context, UIComponent component, Object value)
        throws ValidatorException {
        try {
            Double.parseDouble((String)value);
        } catch (Exception e) {
            throw new ValidatorException(new FacesMessage("Should be a number"));
        }
    }
}

```

- 2) préparer éventuellement quelques ressources (messages d'erreurs , constantes pour effectuer des comparaisons , ...) dans un fichierproperties
 3) il faut dire à JSF qu'on a un nouveau valdateur et comment on veut s'en servir ... on ouvre notre fichier `faces-config.xml` et hop :

```

<validator>
    <validator-id>mustBeDouble</validator-id>
    <validator-class>
        com.jsf.validators.MyDoubleValidator
    </validator-class>
</validator>

```

- 4) associer ce valdateur à un composant de l'arbre
 via le tag `<f:validator type="package_xxx.ClasseDuValdateur">`
 ou bien idéalement `<f:validator validatorId="idDuValdateur">` à imbriquer dans le tag `<h:xxx>` du composant . Exemple:

```

<html:inputText required="true" id="amount" value="#{payementOrder.amount}">
    <f:validator validatorId="mustBeDouble" />

```

[NB: on peut aussi inventer un nouveau tag pour ce valdateur ; ce qui peut être pratique pour le paramétrer]

3.1. interface `javax.faces.validator.Validator`

```

public void validate(javax.faces.context.FacesContext context,
                    javax.faces.component.UIComponent component,
                    java.lang.Object value)
        throws ValidatorException

public ValidatorException(javax.faces.application.FacesMessage message);

```

```

public FacesMessage(FacesMessage.Severity severity,
                    java.lang.String summary,
                    java.lang.String detail)

```

XI - Annexe - Properties & internationalisation

1. Fichier de propriétés (une version par langue)

prop/myJsfpProp.properties

```
hello=hello
display_mens=monthly => {0} {1}
....
```

prop/myJsfpProp_fr.properties

```
hello=bonjour
display_mens=il faut des mensualites de {0} {1}
...
```

NB: Eviter si possible mettre des "." dans le nom des propriétés
Car syntaxe en "#{bundleName.propriété}" pour les récupérer .

2. Prise en compte au sein des pages JSP

```
<%@ taglib uri="http://java.sun.com/jsp/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsp/html" prefix="h" %>

<f:loadBundle basename="prop.myJsfpProp" var="myJsfpProperties"/>

<html> <head> <title>resultats</title> </head>
<body>
<f:view>
    <h:outputText value="#{myJsfpProperties.hello}"/>
    <hr/>
    <h:outputFormat value="#{myJsfpProperties.display_mens}">
        <f:param value="#{calculBean.mensualite}"/>
        <f:param value="Euros"/>
    </h:outputFormat>
</f:view>
</body>
</html>
```

NB:

- L'attribut **basename** de <f:loadBundle /> doit correspondre au nom de base du fichier de propriétés (sans _fr ni .properties) .
- L'attribut **var** de <f:loadBundle /> est un **nom de ResourceBundle chargé en mémoire** . Ce nom est par la suite mentionné en tant que préfixe au sein de l'attribut **value** de <h:outputText /> ou <h:outputFormat .../>
- La partie finale de la syntaxe **value="#{bundleName.propriété}"** correspond à un nom de propriété récupérée dans un fichier ".properties" .
- Si le nom d'une propriété est xxx.yyy alors syntaxe **value="#{bundleName.['xxx.yyy']}"**
- La n^{ème} sous balise <f:param /> de <f:outputFormat /> permet d'indiquer la valeur de la n^{ème} partie variable ({0} , {1} , ...) d'une propriété paramétrable.

3. Ressource globale (à déclarer dans faces-config.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
  version="2.0">
  <application>
    <resource-bundle>
      <base-name>prop.myJsfpProp</base-name>
      <var>myJsfpProperties</var>
    </resource-bundle>
  </application>
</faces-config>
```

Cet équivalent de <f:loadBundle /> a une portée plus globale
 ---> toutes les pages d'une application jsf .

Eventuel accès par programmation :

```
public class MessageProvider {
    private ResourceBundle bundle;

    public ResourceBundle getBundle() {
        if (bundle == null) {
            FacesContext context = FacesContext.getCurrentInstance();
            bundle = context.getApplication().getResourceBundle(context, "myJsfpProperties");
        }
        return bundle;
    }

    public String getValue(String key) {
        String result = null;
        try {
            result = getBundle().getString(key);
        } catch (MissingResourceException e) {
            result = "???" + key + "??? not found";
        }
        return result;
    }
}
```

XII - Annexe – astuces diverses , évolutions JSF

1. Apports de EL 2.2 (depuis Tomcat 7 / JEE6)

Les technologies JSP et JSF utilisent en interne **EL** (*Expression Language*) .

La version 2.2 de EL (*disponible depuis Tomcat7 et JEE6*) apporte (entre autres) la principale nouvelle possibilité suivante :

- **passage de paramètres au sein des méthodes appelées :**
exemple : `#{trader.buy("JAVA")}`

Il est donc maintenant possible de passer des paramètres à une méthode d'action :

```
<h:commandButton action="#{myBean.myAction(valueOfMyParameter)}"/>
```

```
@ManagedBean
@SessionScoped
public class MyBean {
    ...
    public String myAction(Integer myParameter) {
        // do something
        return null;
    }
    ...
}
```

Attention à la portabilité à court terme: (ok tomcat7 , pas ok tomcat6) !

2. DataModel (JSF)

L'**interface** `javax.faces.model.DataModel` est habituellement utilisée par les mécanismes internes de JSF (1 et 2).

ListDataModel est une des implémentations possibles de **DataModel**.

Dans la plupart des cas, on alimente un composant de type **h:dataTable** via une liste de valeurs de types quelconques (ex : `value="#{myBean.listeProduits}"`) et les mécanismes internes de JSF sont alors capables de construire automatiquement un `DataModel` (de type `ListDataModel`) de façon à alimenter les valeurs de chaque ligne du tableau .

Il est cependant possible de gérer explicitement (par programmation) un "dataModel" qui sera utilisé pour remplir un tableau .

Ceci permet (entre autres) de pouvoir facilement gérer la ligne sélectionnée comme le montre l'exemple suivant:

```

@ManagedBean
@SessionScoped
public class MyBean {

    private DataModel dataModel = new ListDataModel();
    ...

    public void init() {
        ...
        dataModel.setWrappedData(uneListeQuelquonque);
    }

    public String mySelectionAction() {
        Object o = dataModel.getRowData(); //récupérer l'élément (ligne) sélectionné .
        //+ casting + ....
        return null; //or ...
    }
    ...
}

```

```

<h:dataTable value="#{monBean.dataModel}" var="row" >
...
<h:column>
    ... <h:commandButton value="sel" action="#{myBean.mySelectionAction}" />
</h:column>
</h:dataTable>

```

NB: bien que la solution "**f.setPropertyActionListener**" présentée ci après soit en générale plus simple (plus directe) que la solution "**DataModel**" pour récupérer l'élément sélectionné dans un tableau , la solution "**DataModel**" offre l'intérêt d'être assez bien intégrée dans le framework "**spring-web-flow**".

3. Astuces diverses pour JSF

3.1. Accéder à un (autre) managedBean (en session) :

par programmation :

```
Bean bean = (Bean)FacesContext.getCurrentInstance().getCurrentInstance()
    .getExternalContext().getSessionMap().get("bean");
```

3.2. Récupérer l'élément sélectionné dans un "dataTable":

En plus de la solution "dataModel" et de la solution "f:param et backingBean" , il est possible d'accéder simplement à l'élément sélectionné dans un "dataTable" via la sous balise "<f:setPropertyActionListener .../>" :

```
@ManagedBean
@SessionScoped
public class MyBean {
    private Produit selectedProd; //+ get/Set
    ...
}
```

```
...
<h:dataTable value="#{myBean.listeProduits}" var="p" >
    <h:column>
        <h:commandButton value="supprimer" action="#{myBean.delete}">
            <f:setPropertyActionListener value="#{p}"
                target="#{myBean.selectedProd}" />
        </h:commandButton>
    </h:column>
</h:dataTable>
...
```

4. Redirection automatique en cas de session terminée/expirée

Les mécanismes internes de JSF placent tout un tas de champs cachés (<input type='hidden' .../>) au sein des pages HTML construites .

```
<input type="hidden" name="javax.faces.ViewState"
id="javax.faces.ViewState" value="cdx43wedsfdg654ed" />
```

L'un de ces champs cachés s'appelle javax.faces.**ViewState** et correspond à une sorte de compteur sophistiqué qui est stocké à la fois en tant que champ caché dans un formulaire html (avec aller/retour serveur/navigateur) et à la fois en session coté serveur.

Lorsqu'un formulaire JSF est renvoyé (via un submit http) , la page JSF/HTML se rappelle elle même (post-back) et la valeur du champ caché "viewState" est comparée à celle qui est en session http.

→ Si les deux valeurs coïncident alors tout va bien , les traitements coté serveurs sont bien exécutés normalement.

→ Si les deux valeurs ne coïncident pas , c'est qu'il y a un problème (double submit involontaire , problème de sécurité , session expirée , ...) et une exception est généralement remontée.

Le problème le plus courant correspond à une **session qui à expirée** :

- suite à un assez long timeout en production.
- suite à un rechargement du code (page + mbean) en développement.

Pour paramétrer une redirection automatique vers la page d'accueil en cas de session terminée/expirée , il suffit d'ajouter la configuration suivante au sein de WEB-INF/web.xml:

```
<error-page>
  <exception-type>javax.faces.application.ViewExpiredException</exception-type>
  <location>/pages/welcome.jsf</location>
  <!-- ou vers /welcome.jsf ou ... -->
</error-page>
```

Ce qui est très utile, voir fondamental !!!!

XIII - Annexe - paramètres jsf & backing Bean

1. Backing Bean

On appelle "Managed Bean" tout composant java pris en charge par le framework JSF (et ordinairement déclaré dans le fichier faces-config.xml) .

Un "Managed Bean" ordinaire comporte des propriétés "métiers" qui ne sont pas directement associées aux composants JSF (ex: age de type Integer , nom de type String).

Suite à l'interprétation de `value=#{xxxBean.nom}` dans un composant JSF d'une page JSP , le framework JSF établit alors automatiquement une liaison (binding) entre le composant JSF et la propriété métier "nom" du Bean "xxxBean" par le moyen un petit objet de liaison intermédiaire .

Il est éventuellement possible de paramétrer une liaison (binding) plus directe en:

- utilisant l'attribut **binding="#{xxxBean.comp}"** d'un composant JSF d'une page JSP
- préparant une propriété "comp" de type dérivé de **UIComponent** au sein d'un managed bean utilisé en arrière plan .

==> La propriété "comp" (codée via private + get/set) permettra de directement manipuler le composant JSF depuis le "managed bean" en arrière plan (comme si on tirait sur les fils d'une marionnette).

Ainsi, depuis la référence "comp" on peut récupérer la valeur choisie via **comp.getValue()** ou bien agir sur le composant avant la phase de rendu via **comp.setEnabled(false)**;

Cette façon de procéder est généralement appelée "**backing bean**".

Autrement dit, un "back bean" n'est rien d'autre qu'un "managed bean" qui comporte au moins une propriété de type dérivé de *UIComponent* et associée à un composant JSF via l'attribut *binding* d'une page JSP.

L'exemple ci-après montrera une utilisation concrète et utile du procédé "backing bean" .

2. Passage de paramètre(s)

De façon à obtenir un tableau d'entités avec pour chaque ligne un "commandLink" ou "commandButton" redirigeant vers une page de détails , on a besoin de paramétrer l'action "versDétails" avec l'identifiant de l'entité à détailler.

L'exemple ci dessous correspond à un tableau d'entité "compte bancaire" avec numéro et solde .

La colonne détails permet de déclencher l'action "listerDernieresOperations" devant être paramétrée par le numéro du compte à détailler:

listeComptes.jsp

```

<f:view><h:form>
  <h:dataTable value="#{listeComptesBean.liste_comptes}" var="compte" border="1">
    ....
    <h:column>
      <f:facet name="header"><f:verbatim>détails</f:verbatim></f:facet>
      <h:commandLink action="#{listeComptesBean.listerDernieresOperations}"
        value="dernières opérations" >
        <f:param name="paramNumCptSel"
          value="#{compte.numero}"
          binding="#{listeComptesBean.paramNumCptSel}" />
      </h:commandLink>
    </h:dataTable>
  </h:form></f:view>

```

ListeComptesBean.java

```

public class ListeComptesBean {
  ...
  private UIParameter paramNumCptSel;

  public UIParameter getParamNumCptSel() {
    return paramNumCptSel;
  }

  public void setParamNumCptSel(UIParameter paramNumCptSel) {
    this.paramNumCptSel = paramNumCptSel;
  }

  public String listerDernieresOperations() {
    String res="dernieres_operations";
    try {
      this.numCptSel=(Long) paramNumCptSel.getValue();
      ...
    } catch (Exception e) {
      e.printStackTrace();
      res="erreur";
    }
    return res;
  }
}

```

XIV - Annexe - Intégration IOC (jsf , jee , spring)

1. Injection IOC via JSF : plusieurs beans

Au lieu d'avoir un seul "JavaBean" en arrière plan des pages JSP/JSF , il est quelquefois préférable d'avoir un "JavaBean" de portée/scope "request" ou "session" comportant à son tour des références vers des sous objets de portée/scope plus large ("session" ou "application") .

1.1. Injection de dépendance prise en charge par JSF

Dans un environnement classique (sans assistants) , ceci s'effectue en paramétrant des *injections de dépendances* au niveau des "ManagedBean" de `faces-config.xml` :

```
class MyJsFBean {
    private ReferentielApp referentiel = null;
    // méthode d'injection de dépendances:
    public setReferentiel( ReferentielApp refObj){
        this. referentiel =refObj;
    }
    public String doXxxAction(){ // traitements délégués sur referentiel
        this.referentiel..... ; return null; }
    }
}
```

WEB-INF/faces-config.xml

```
<managed-bean>
    <managed-bean-name>myReferentiel</managed-bean-name>
    <managed-bean-class>myjsf.referentielApp</managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
</managed-bean>
...
<managed-bean>
    <managed-bean-name>myJsFBean</managed-bean-name>
    <managed-bean-class>myjsf.MyJsFBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
        <property-name>referentiel</property-name>
        <value>#{myReferentiel}</value>
    </managed-property>
</managed-bean>
```

Et si en plus un sous object injecté est accessible via une méthode publique de type "getSubObj()" il est alors possible de l'utiliser directement depuis une page jsp avec la syntaxe suivante:

```
<h:inputText value="#{myJsFBean.subObj.montant}" />
<h:outputText value="#{myJsFBean.subObj.resultat}" />
```

1.2. Structure IOC conseillée de la partie "Application/Web"

NB: en général les Beans de scope=request s'appuient fréquemment sur des beans plus stables (de scope=session ou application).

En scope=session, on trouve généralement des « *Coordinateurs* » associés aux *Uses Cases*.

En scope=application on trouve généralement des « *référentiel/façade* » ou des « *services* »

Référentiel applicatif / référentiel de services:

Le référentiel est unique à toute l'application et est commun à tous les utilisateurs (scope=application). Jouant essentiellement le rôle de façade vers les services métiers de l'application, un référentiel peut éventuellement servir de base pour des mécanismes de "cache" .

====> utile pour ne pas invoquer des milliers de fois des services qui renvoient quelquefois des valeurs toujours identiques (stables).

Coordinateur (scope="session") associé à un "Use Case" UML :

En plus du référentiel , il est assez souvent utile de mettre en oeuvre des objets de scope=session pour mémoriser l'état d'avancement d'un processus applicatif (données spécifiques à une session utilisateur et devant être accessibles depuis plusieurs pages/vues) .

Ces objets sont assez souvent appelés "coordinateurs" et dérivent de la modélisation des cas d'utilisations (UML).

2. Liens entre JSF et Spring

2.1. ContextLoaderListener (Spring)

A intégrer au sein de WEB-INF/web.xml

```
....
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/classes/context.xml</param-value>
</context-param>
....
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
....
```

Ceci permet de charger automatiquement en mémoire la configuration "Spring" (ici tous les fichiers ".xml" du répertoire /WEB-INF/spring) dès le démarrage de l'application WEB.

NB1: le paramètre *contextConfigLocation* peut éventuellement comporter une liste de chemin (vers plusieurs fichiers) séparés par des virgules .

Exemple: "/WEB-INF/classes/spring/*.xml" ou encore

"/WEB-INF/classes/contextSpring.xml,/WEB-INF/classes/context2.xml"

NB2: les fichiers de configurations "xxx.xml" placé (en mode source) dans "src" (ou bien dans les ressources de maven) se retrouvent normalement dans /WEB-INF/classes en fin de "build" .

NB3: via le **préfixe** "*classpath*:/*" on peut préciser des chemins qui seront recherchés dans tous les éléments du classpath (c'est à dire dans tous les ".jar" du projet : par exemple tous les ".jar" présents dans WEB-INF/lib) *exemple:*

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath*:/serviceSpringConf.xml,classpath*:/dataSourceForTestSpringConf.xml
    </param-value>
</context-param>
```

Au sein d'un éventuel petit servlet de test on peut instancier des Beans via Spring :

```
servletContext = .... getServletContext(); // vue comme objet application au sein d'une page JSP
WebApplicationContext ctx =
    WebApplicationContextUtils.getWebApplicationContext(servletContext);
IXxx bean = (IXxx) ctx.getBean(...);
....
request.setAttribute("nomBean",bean); // on stocke le bean au sein d'un scope (session,request,...)
....
rd.forward(request,response); // redirection vers page JSP
....
```

2.2. Injection "Spring" au sein du framework JSF

En plus de la configuration évoquée plus haut au niveau de *WEB-INF/web.xml* , il faut :

Modifier le fichier *WEB-INF/faces-config.xml* en y ajoutant le bloc "<application> ...</application>" précisant l'utilisation de *SpringBeanFacesELResolver* .

```
<faces-config>
  <application>
    <!-- <variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver> pour jsf 1.1 -->

    <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>
    <!-- a partir de jsf 1.2 --> ...
    ...
  </application>
  ...
</faces-config>
```

Ceci permettra d'injecter des "beans Spring" (ex: services métiers) au sein des "managed-bean" de JSF de la façon suivante:

WEB-INF/faces-config.xml

```
....
<managed-bean>
  <managed-bean-name>myJsBean</managed-bean-name>
  <managed-bean-class>myjsf.MyJsBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>myService</property-name>
    <value>#{mySpringService}</value>
  </managed-property>
</managed-bean>
....
```

Effets:

Les noms #{xxx} utilisés par JSF seront résolus:

- par les mécanismes standards de JSF (en premier)
- par le *SpringBeanFacesELResolver* de Spring puisant à son tour des "beans" instanciés via une fabrique de Spring selon la configuration de *WEB-INF/classes/context.xml* ou autre (dans un second temps).

La résolution s'effectue sur les valeurs des ID ou Noms des composants "Spring".

En d'autres termes, les mécanismes JSF, déjà en partie basés sur des principes IOC, peuvent ainsi être ajustés pour injecter des composants Spring au sein des "Managed Bean" (ici *setMyService()* de la classe *myjsf.MyJsBean*).

Détail important:

JSF utilise une syntaxe de type *#{nomBeanJsf.nomPropriété}* .

Ceci implique que le nom (l'id) d'un composant JSF ne doit pas comporter de caractère "." (ni de "-") mais le "_" est autorisé .

3. Eventuelles injections par annotations

Il existent plusieurs solutions (à choisir selon le contexte de l'architecture globale) :

- via annotations/injections "Spring"
- via annotations/injections "JSF2" et référentiel/façade masquant le "back-office".
- Via annotations/injections "CDI/JEE6" et EJB3.1 (ou futur Spring?)
-

(sachant que chacune de ces solutions comporte plusieurs variantes!!!)

3.1. Via annotations Spring (avec ou sans inject)

Possibilité technique :

Etant donné que les 2 frameworks "Spring" et "JSF" sont tous les deux capables de prendre en charge des injections de dépendances entre composants, on peut éventuellement s'appuyer sur Spring pour structurer les dépendances entre composants de la couche IHM/WEB.

Ceci permet d'utiliser une syntaxe concise et claire pour les habitués à Spring (en utilisant les annotations @Component , @Scope et @Autowired de Spring sur les composants JSF) .

```
@Component("referentiel") // nom logique (id Spring) = "referentiel" ici
@Scope("singleton") // "singleton" = plus proche équivalent Spring de "application"
public class ReferentielApp {
    private GestionComptes serviceGestionComptes =null;
    private Service2 service2 =null;

    // redirection vers le service "gestionComptes"
    public GestionComptes getServiceGestionComptes() {
        return serviceGestionComptes;
    }

    // injection ioc du service métier "gestionComptes"
    // @Autowired (by type)
    @Autowired // ou bien @Inject
    public void setServiceGestionComptes(GestionComptes serviceGestionComptes) {
        this.serviceGestionComptes = serviceGestionComptes;
    }

    public Service2 getService2() {
        return service2;
    }
}
```

SessionClient.java

```
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component("sessionClient")
@Scope("session")
public class SessionClient {

    private long num_client;    // + get/set
}
```

ListeComptesBean.java

```

....
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component("listeComptesBean")
@Scope("request")
public class ListeComptesBean {
    ...
    private SessionClient sessionClient = null; //ref vers sous objet "SessionClient"
    private ReferentielApp referentiel; // ref sur référentiel des services, ...

    @Autowired // ou bien @Inject
    public void setReferentiel(ReferentielApp refApp){ this.referentiel = refApp;
    }

    @Autowired // ou bien @Inject
    public void setSessionClient(SessionClient sessionClient) {
        this.sessionClient = sessionClient;
    }

    public String listerDernieresOperations(){
        String res="dernieres_operations";
        try {
            ....
            Compte cpt =
                referentiel.getServiceGestionComptes().getCompteByNum(this.numCptSel);
            ....
        } catch (Exception e) { e.printStackTrace(); res="erreur";
        }
        return res;
    }
    ....
}

```

NB : Sur un petit projet, on peut éventuellement court-circuiter la façade ou le référentiel et l'on peut directement injecter les services Spring dans les managedBeans de JSF via @Inject (ou @Autowired)

Défaut important de cette solution : si on utilise Spring de bout en bout , il faudra par la suite éventuellement remanier le code en profondeur si l'on décide de remplacer Spring par autre chose.

3.2. Via Annotation/injection JSF et façade/référentiel

- Laisser Spring ou Ejb3 s'occuper des objets "back-office"
- Masquer la technologie "back-office" derrière une façade et/ou un référentiel.
- Expliciter au sein des composants JSF des injections d'éléments "back_office" avec l'annotation `@ManagedProperty(value="#{facade_ou_referentiel}")` de **JSF2**.

***NB1:** cette annotation peut être placée au dessus des "private" mais il faut absolument générer les "getter/setter" pour assurer un bon fonctionnement .*

```
import javax.faces.bean.ApplicationScoped;
import javax.faces.bean.ManagedBean;
```

@ManagedBean

@ApplicationScoped

```
public class Referentiel {
    private FacadeMiniBank facadeMiniBank;

    public Referentiel() {
        ServletContext sc = (ServletContext)
            FacesContext.getCurrentInstance().getExternalContext().getContext();
        //facadeMiniBank = FacadeMiniBankFactory.getInstance();
        facadeMiniBank = FacadeMiniBankFactory.getInstanceFromContext(sc);
    }

    public FacadeMiniBank getFacadeMiniBank() {
        return facadeMiniBank;
    }
}
```

....

@ManagedBean

@SessionScoped

```
public class ClientComptes {

    private Long numClient;
    private List<Compte> listeComptes;

    public void actualiserListeComptes() throws MyServiceException {
        this.listeComptes=getServiceGestionComptes().getComptesOfClient(numClient);
    }

    @ManagedProperty(value="#{referentiel}")
    private Referentiel referentiel;

    public Referentiel getReferentiel() {
        return referentiel;
    }
}
```



```

public void setReferentiel(Referentiel referentiel) {
    this.referentiel = referentiel;
}

private GestionComptes getServiceGestionComptes(){
    return referentiel.getFacadeMiniBank().getServiceGestionComptes();
}

...
}

```

NB2: L'annotation `@ManagedProperty` est pour l'instant un peu moins pratique que `@Autowired` ou `@Inject` car elle nécessite de préciser l'id du composant à injecter .

`@Inject` est quelquefois utilisable dans un contexte JEE6/CDI (à tester !!!).

3.3. Via annotations/injections de JEE6/CDI

Dans le cadre d'un projet JEE6/CDI (ex : avec EJB3.1 et Weld pour Jboss 7.1), on peut utiliser les annotations (d'inspiration "seam" et maintenant normalisées) `@Inject` , `@SessionSscope` , `@Produces` au niveau des composants "managed Bean" de JSF2 .

```

import javax.enterprise.context.SessionScoped; //plutôt que javax.faces.bean
import javax.inject.Inject;
import javax.inject.Named;

@Named //plutôt que @ManagedBean
@SessionScoped //javax.enterprise.context (CDI/JEE6) plutôt que javax.faces.bean (JSF2 only)
public class ClientComptes implements Serializable {
    ...

    @Inject
    private GestionClients serviceGestionClients; //service ou bien facade ou bien referentiel

    @Inject
    private GestionComptes serviceGestionComptes; //service ou facade ou referentiel

    public void actualiserListeComptes() throws MyServiceException{
        this.listeComptes=serviceGestionComptes.getComptesOfClient(numClient);
    }

    ...
}

```

Les annotations CDI/JEE6 sont à priori celles qui vont devenir les plus standards et les plus universellement reconnus (JSF2 , EJB3.1 ,).

Pour l'instant Spring 3 ne reconnaît de `@Inject` et `@Named` (et pas encore `SessionScoped` et les autres annotations du package `javax.enterprise.context`).

Heureusement , Spring3 a prévu un mécanisme de spécialisation au niveau de l'interprétation des annotations et l'on peut ainsi demander à Spring 3 d'interpréter certaines annotations CDI (ex : `javax.enterprise.context.SessionScoped`) comme des équivalents des annotations de Spring (ex : `@Scope("session")`) :

Il suffit pour cela de préciser un scope-resolver dans la configuration Spring3 :

```
...
<context:component-scan base-package="tp.myapp.web.mbean"
    scope-resolver="net.wessendorf.spring.scopes.CdiScopeMetadataResolver"/>
....
```

Avec la classe suivante (de monsieur Wessendorf) :

```
/*
 * Copyright (C) 2010 Matthias Weßendorf.
 * Licensed under the Apache License, Version 2.0 (the "License");
 */
package net.wessendorf.spring.scopes;
import java.util.Set;
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.context.RequestScoped;
import javax.enterprise.context.SessionScoped;
import org.springframework.beans.factory.annotation.AnnotatedBeanDefinition;
import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.context.annotation.AnnotationScopeMetadataResolver;
import org.springframework.context.annotation.ScopeMetadata;
import org.springframework.web.context.WebApplicationContext;

public class CdiScopeMetadataResolver extends AnnotationScopeMetadataResolver {

    public ScopeMetadata resolveScopeMetadata(BeanDefinition definition)
    {
        ScopeMetadata metadata = new ScopeMetadata();
        if (definition instanceof AnnotatedBeanDefinition) {
            AnnotatedBeanDefinition annDef = (AnnotatedBeanDefinition) definition;
            Set<String> annotationTypes = annDef.getMetadata().getAnnotationTypes();

            if (annotationTypes.contains(RequestScoped.class.getName())) {
                metadata.setScopeName(WebApplicationContext.SCOPE_REQUEST);
            }
            else if (annotationTypes.contains(SessionScoped.class.getName())) {
                metadata.setScopeName(WebApplicationContext.SCOPE_SESSION);
            }
            else if (annotationTypes.contains(ApplicationScoped.class.getName())) {
                metadata.setScopeName(WebApplicationContext.SCOPE_APPLICATION);
            }
            else { // do the regular Spring stuff..
                return super.resolveScopeMetadata(definition);
            }
        }
        return metadata;
    }
}
```

librairies nécessaires :

cdi-api-1.0.jar et **javax.inject-1.jar**

==> résultat :

Ce code suivant est bien interprété avec Spring3/MyFaces2 au sein de Tomcat :

```
import javax.enterprise.context.SessionScoped;
import javax.inject.Inject;
import javax.inject.Named;
...

@Named //plutôt que @ManagedBean
@SessionScoped //javax.enterprise.context (CDI/JEE6)
                //plutôt que javax.faces.bean (JSF2 only)
public class ClientComptes {
    ...

    @Inject //plutôt que @ManagedProperty(value="#{referentiel}")
    private Referentiel referentiel;
    ...
}
```

==> Le bénéfice de cette solution se situe essentiellement au niveau de **@Inject** (qui est bien plus souple que **@ManagedProperty**) .

XV - Annexe - composants composites

1. Composants composites (depuis JSF2)

1.1. Principe

Depuis la version 2 de JSF, il est maintenant possible de créer rapidement des nouveaux composants JSF en s'appuyant sur deux éléments complémentaires :

- une classe java ou groovy (optionnelle) pour la logique fonctionnelle
- un fichier .xhtml pour le rendu graphique (encodé via l'extension composite) et s'appuyant sur des sous composants JSF habituels (ex : h:dataTable ou h:outputText)

1.2. Mise en œuvre

Exemple de fichier ".xhtml" pour le rendu d'un composant composite :

src/main/webapp/resources/mycomponents/xxx.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:cc="http://java.sun.com/jsf/composite"
    xmlns:f="http://java.sun.com/jsf/core">
<h:body>
  <cc:interface>
    <cc:attribute name="nom" required="true"/>
  </cc:interface>
  <cc:implementation>
    <b>xxx de #{cc.attrs.nom} : </b><br/>
    <h:dataTable id="listXxx" binding="#{cc.data}" var="x">
      <h:column>
        <h:outputText value="#{x.p1} : #{x.p2}"/>
      </h:column>
    </h:dataTable>
  </cc:implementation>
</h:body>
</html>
```

Remarques:

- la partie **cc:interface** permet de préciser l'interface (non visible) du composant avec sa liste d'attribut(s) .
- la partie **cc:implementation** permet de préciser l'implémentation du rendu du composant .
- la syntaxe **#{cc.attrs.nomAttribut}** permet d'accéder à la valeur d'un attribut .
- la syntaxe **#{cc.data}** permet d'accéder à la valeur principale du composant . Cette valeur (de type *UIData*) sera codée dans la classe java du composant .

Exemple de classe java pour un composant composite JSF2 :

src/main/java/mycomponents/Xxx.java

```
public class Xxx extends UIComponentBase implements NamingContainer {

    //Méthode devant absolument être surchargée par le composant :
    public String getFamily() {return "javax.faces.NamingContainer";}

    private HtmlOutputText pseudo;

    //valeur principale associée(bind) au composant :
    private UIData data;

    @Override
    public void encodeBegin(FacesContext context) throws IOException {
        DataModel dataModel = new ListDataModel();
        try {
            Object attrNomValue = getAttributes().get("nom") ;
            //...donnees...from....attrNom...
            dataModel.setWrappedData(...donnees....);
            data.setValue(dataModel);
        } catch (Exception e) {
            ....
        }
        super.encodeBegin(context);
    }
    ...
}
```

Remarques :

- Au sein de la classe java du composant, on récupère la valeur d'un attribut (ex : "nom") via la méthode **getAttributes()** retournant une map.
- ...

1.3. Packaging de composant(s) composite(s)

Projet java (de packaging "jar") avec comme parties de l'arborescence :

META-INF/resources/mycomponents/Xxx.xhtml, Yyy.xhtml ,

META-INF/faces-config.xml (souvent vide mais avec entête JSF2)

mycomponents/Xxx.java

1.4. Utilisation d'un composant composite

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:mycomp="http://java.sun.com/jsf/composite/mycomponents">
  <h:head>

  </h:head>
  <h:body>
    <f:view>
      <mycomp:Xxx nom="Didier" />
    </f:view>
  </h:body>
</html>
```

Remarques :

- mycomponents est ici le nom (de notre librairie) mais surtout le nom du répertoire (sous resources) qui comporte le fichier Xxx.xhtml
- le ".jar" correspondant à notre librairie doit simplement être présent dans WEB-INF/lib
- ...