Recherche dans DEJ avec Google Rechercher

g+1 10

Développons en Java v 2.00
Copyright (C) 1999-2014 Jean-Michel DOUDOUX.

69. JSF (Java Server Faces)

Niveau : Supérieur

69.1. La présentation de JSF

Les technologies permettant de développer des applications web avec Java ne cessent d'évoluer :

1. Servlets
2. JSP
3. MVC Model 1 : servlets + JSP
4. MVC Model 2 : un seule servlet + JSP
5. Java Server Faces

Java Server Faces (JSF) est une technologie dont le but est de proposer un framework qui facilite et standardise le développement d'applications web avec Java. Son développement a tenu compte des différentes expériences acquises lors de l'utilisation des technologies standard pour le développement d'applications web (servlet, JSP, JSTL) et de différents frameworks (Struts, ...).

Le grand intérêt de JSF est de proposer un framework qui puisse être mis en oeuvre par des outils pour permettre un développement de type RAD pour les applications web et ainsi faciliter le développement des applications de ce type. Ce type de développement était déjà courant pour des applications standalone ou clients/serveurs lourds avec des outils tels que Delphi de Borland, Visual Basic de Microsoft ou Swing avec Java.

Ce concept n'est pourtant pas nouveau dans les applications web puisqu'il est déjà mis en oeuvre par WebObject d'Apple et plus récemment par ASP.Net de Microsoft mais cette mise en oeuvre à grande échelle fût relativement tardive. L'adoption du RAD pour le développement web trouve notamment sa justification dans le coût élevé de développement de l'IHM à la « main » et souvent par copier/coller d'un mixe de plusieurs technologies (HTML, JavaScript, ...), rendant fastidieux et peu fiable le développement de ces applications.

Plusieurs outils commerciaux intègrent déjà l'utilisation de JSF notamment Studio Creator de Sun, WSAD d'IBM, JBuilder de Borland, JDevelopper d'Oracle, ...

Même si JSF peut être utilisé par codage à la main, l'utilisation d'un outil est fortement recommandée pour pouvoir mettre en oeuvre rapidement toute la puissance de JSF.

Ainsi de par sa complexité et sa puissance, JSF s'adapte parfaitement au développement d'applications web complexes en facilitant leur écriture.

Les pages officielles de cette technologie sont à l'URL :
http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html.

La version 1.0 de Java Server Faces, développée sous la JSR-127, a été validée en mars 2004.

JSF est une technologie utilisée côté serveur dont le but est de faciliter le développement de l'interface utilisateur en séparant clairement la partie « interface » de la partie « métier » d'autant que la partie interface n'est souvent pas la plus compliquée mais la plus fastidieuse à réaliser.

Cette séparation avait déjà été initiée avec la technologie JSP et particulièrement les bibliothèques de tags personnalisés. Mais JSF va encore plus loin en reposant sur le modèle MVC et en proposant de mettre en oeuvre :

- l'assemblage de composants serveurs qui génèrent le code de leur rendu avec la possibilité d'associer certains composants à une source de données encapsulée dans un bean
- l'utilisation d'un modèle de développement standardisé reposant sur l'utilisation d'événements et de listeners
- la conversion et la validation des données avant leur utilisation dans les traitements

- la gestion de l'état des composants de l'interface graphique
- la possibilité d'étendre les différents modèles et de créer ses propres composants
- la configuration de la navigation entre les pages
- le support de l'internationalisation
- le support pour l'utilisation par des outils graphiques du framework afin de faciliter sa mise en oeuvre

JSF se compose :

- d'une spécification qui définit le mode de fonctionnement du framework et une API : l'ensemble des classes de l'API est contenu dans les packages javax.faces.
- d'une implémentation de référence
- de bibliothèques de tags personnalisés fournies par l'implémentation pour utiliser les composants dans les JSP, gérer les événements, valider les données saisies, ...

Le rendu des composants ne se limite pas à une seule technologie même si l'implémentation de référence ne propose qu'un rendu des composants en HTML.

Le traitement d'une requête traitée par une application utilisant JSF utilise un cycle de vie particulier constitué de plusieurs étapes :

- Création de l'arbre de composants
- Extraction des données des différents composants de la page
- Conversion et validation des données
- Extraction des données validées et mise à jour du modèle de données (javabean)
- Traitements des événements liés à la page
- Génération du rendu de la réponse

Ces différentes étapes sont transparentes lors d'une utilisation standard de JSF.

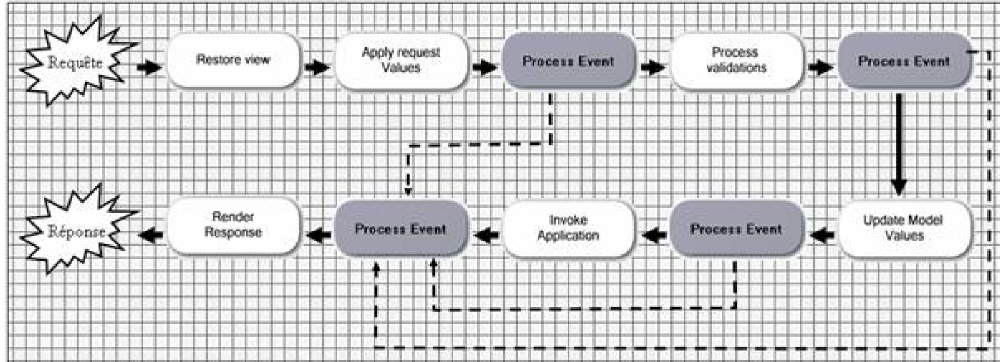
Ce chapitre contient plusieurs sections :

- La présentation de JSF
- Le cycle de vie d'une requête
- Les implémentations
- Le contenu d'une application
- La configuration de l'application
- Les beans
- Les composants pour les interfaces graphiques
- La bibliothèque de tags Core
- La bibliothèque de tags Html
- La gestion et le stockage des données
- La conversion des données
- La validation des données
- La sauvegarde et la restauration de l'état
- Le système de navigation
- La gestion des événements
- Le déploiement d'une application
- Un exemple d'application simple
- L'internationalisation
- Les points faibles de JSF

69.2. Le cycle de vie d'une requête

JSF utilise la notion de vue (view) qui est composée d'une arborescence ordonnée de composants inclus dans la page.

Les requêtes sont prises en charge et gérées par le contrôleur d'une application JSF (en général une servlet). Celle-ci va assurer la mise en oeuvre d'un cycle de vie des traitements permettant de traiter la requête en vue d'envoyer une réponse au client.



JSF propose pour chaque page un cycle de vie pour traiter la requête HTTP et générer la réponse. Ce cycle de vie est composé de plusieurs étapes :

1. Restore view ou Reconstruct Component Tree : cette première phase permet au serveur de recréer l'arborescence des composants qui composent la page. Cette arborescence est stockée dans un objet de type FacesContext et sera utilisée tout au long du traitement de la requête.
2. Apply Request Value : dans cette étape, les valeurs des données sont extraites de la requête HTTP pour chaque composant et sont stockées dans

leur composant respectif dans le FaceContext. Durant cette phase des opérations de conversions sont réalisées pour permettre de transformer les valeurs stockées sous forme de chaînes de caractères dans la requête http en un type utilisé pour le stockage des données.
3. Perform validations : une fois les données extraites et converties, il est possible de procéder à leur validation en appliquant les validators enregistrés auprès de chaque composant. Les éventuelles erreurs de conversions sont stockées dans le FaceContext. Dans ce cas, l'étape suivante est directement « Render Response » pour permettre de réafficher la page avec les valeurs saisies et afficher les erreurs
4. Synchronize Model ou update model values : cette étape permet de stocker dans les composants du FaceContext leurs valeurs locales validées respectives. Les éventuelles erreurs de conversions sont stockées dans le FaceContext. Dans ce cas, l'étape suivante est directement « Render Response » pour permettre de réafficher la page avec les valeurs saisies et d' afficher les erreurs
5. Invoke Application Logic : dans cette étape, le ou les événements émis dans la page sont traités. Cette phase doit permettre de déterminer la page résultat qui sera renvoyée dans la réponse en utilisant les règles de navigation définie dans l'application. L'arborescence des composants de cette page est créée.
6. Render Response : cette étape se charge de créer le rendue de la page de la réponse.

69.3. Les implémentations

Java Server Faces est une spécification : il est donc nécessaire d'obtenir une implémentation de la part d'un tiers.

Plusieurs implémentations commerciales ou libres sont disponibles, notamment l'implémentation de référence de Sun et MyFaces qui est devenu un projet du groupe Apache.

69.3.1. L'implémentation de référence

Comme pour toute JSR validée, Sun propose une implémentation de référence des spécifications de la JSR, qui est la plus complète possible.

Plusieurs versions de l'implémentation de référence de Sun sont proposées :

Version	Date de diffusion
1.0	Mars 2004
1.1	Mai 2004
1.1_01	Septembre 2004

La solution la plus simple pour utiliser l'implémentation de référence est d'installer le JWSDK 1.3 qui est fourni en standard avec l'implémentation de référence de JSF. La version de JSF fournie avec le JWSDK 1.3 est la 1.0.

Pour utiliser la version 1.1, il faut supprimer le répertoire jsf dans le répertoire d'installation de JWSDK, télécharger l'implémentation de référence, décompresser son contenu dans le répertoire d'installation de JWSDK et renommer le répertoire jsf-1_1_01 en jsf.

Il est aussi possible de télécharger l'implémentation de référence sur le site de Sun et de l'installer « manuellement » dans un conteneur web tel que Tomcat. Cette procédure sera détaillée dans une des sections suivantes.

Pour cela, il faut télécharger le fichier jsf-1_1_01.zip et le décompresser dans un répertoire du système. L'archive contient les bibliothèques de l'implémentation, la documentation des API et des exemples.

Les exemples de ce chapitre vont utiliser cette version 1.1 de l'implémentation de référence des JSF.

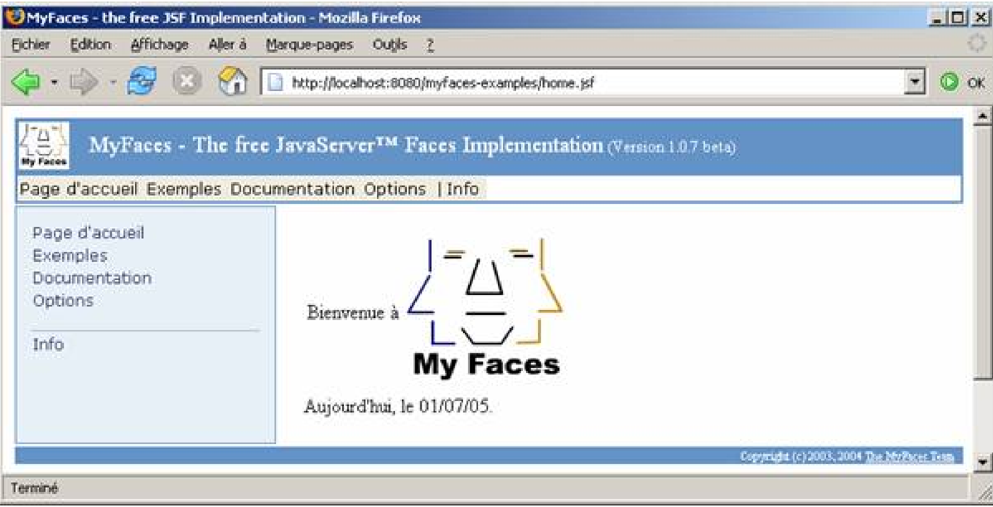
69.3.2. MyFaces

MyFaces est une implémentation libre des Java Server Faces qui est devenue un projet du groupe Apache.

Elle propose plusieurs composants spécifiques en plus de ceux imposés par les spécifications JSF.

Le site de MyFaces est à l'URL : http://myfaces.apache.org/

Il faut télécharger le fichier et le décompresser dans un répertoire du système. Il suffit alors de copier le fichier myfaces-examples.war dans le répertoire webapps de Tomcat, de relancer le serveur puis saisir l'URL http://localhost:8080/myfaces-examples



Pour utiliser MyFaces dans ses propres applications, il faut réaliser plusieurs opérations.

Il faut copier les fichiers *.jar du répertoire lib de MyFaces et myfaces-jsf-api.jar dans le répertoire WEB-INF/lib de la webapp.

Dans chaque page qui va utiliser les composants de MyFaces, il faut déclarer la bibliothèque de tags dédiés.

Exemple :

```
1. <%@ taglib uri="http://myfaces.sourceforge.net/tld/myfaces_ext_0_9.tld" prefix="x"%>
```

69.4. Le contenu d'une application

Les applications utilisant JSF sont des applications web qui doivent respecter les spécifications de J2EE.

En tant que telles, elles doivent avoir la structure définie par J2EE pour toutes les applications web :

/
/WEB-INF
/WEB-INF/web.xml
/WEB-INF/lib
/WEB-INF/classes

Le fichier web.xml doit contenir au minimum certaines informations notamment, la servlet faisant office de contrôleur, le mapping des URL pour cette servlet et des paramètres.

Exemple :

```
01. <!DOCTYPE web-app PUBLIC
02. "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
03. "http://java.sun.com/dtd/web-app_2_3.dtd">
04. <web-app>
05.   <display-name>Test JSF</display-name>
06.   <description>Application de tests avec JSF</description>
07.   <context-param>
08.     <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
09.     <param-value>client</param-value>
10.   </context-param>
11.   <!-- Faces Servlet -->
12.   <servlet>
13.     <servlet-name>Faces Servlet</servlet-name>
14.     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
15.     <load-on-startup> 1 </load-on-startup>
16.   </servlet>
17.   <!-- Faces Servlet Mapping -->
18.   <servlet-mapping>
19.     <servlet-name>Faces Servlet</servlet-name>
20.     <url-pattern>*.jsf</url-pattern>
21.   </servlet-mapping>
22. </web-app>
```

Chaque implémentation nécessite un certain nombre de bibliothèques tierces pour son bon fonctionnement.

Par exemple, pour l'implémentation de référence, les bibliothèques suivantes sont nécessaires :

jsf-api.jar
jsf-ri.jar

jstl.jar
standard.jar
common-beanutils.jar
commons-digester.jar
commons-collections.jar
commons-logging.jar

Remarque : avec l'implémentation de référence, il n'y a aucun fichier .tld à copier car ils sont intégrés dans le fichier jsf-impl.jar.

Les fichiers nécessaires dépendent de l'implémentation utilisée.

Ces bibliothèques peuvent être mises à disposition de l'application selon plusieurs modes :

- incorporées dans le package de l'application dans le répertoire /WEB-INF/lib
- incluses dans le répertoire des bibliothèques partagées par les applications web des conteneurs web s'ils proposent une telle fonctionnalité. Par exemple avec Tomcat, il est possible de copier ces bibliothèques dans le répertoire shared/lib.

L'avantage de la première solution est de faciliter la portabilité de l'application sur différents conteneur web mais elle duplique ces fichiers si plusieurs applications utilisent JSF.

Les avantages et inconvénients de la première solution sont exactement à l'opposé de ceux de la seconde solution. Le choix de l'une ou l'autre est donc à faire en fonction du contexte de déploiement.

69.5. La configuration de l'application

Toute application utilisant JSF doit posséder au moins deux fichiers de configuration qui vont contenir les informations nécessaires à la bonne exécution de l'application.

Le premier fichier est le descripteur de toute application web J2EE : le fichier web.xml contenu dans le répertoire WEB-INF.

Le second fichier est un fichier de configuration au format XML, particulier au paramétrage de JSF et nommé faces-config.xml.

69.5.1. Le fichier web.xml

Le fichier web.xml doit contenir au minimum certaines informations notamment, la servlet faisant office de contrôleur, le mapping des URLs pour cette servlet et des paramètres pour configurer JSF.

Exemple :

```
01. <!DOCTYPE web-app PUBLIC
02. "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
03. "http://java.sun.com/dtd/web-app_2_3.dtd">
04. <web-app>
05.   <display-name>Test JSF</display-name>
06.   <description>Application de tests avec JSF</description>
07.   <context-param>
08.     <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
09.     <param-value>client</param-value>
10.   </context-param>
11.
12.   <!-- Servlet faisant office de controleur-->
13.   <servlet>
14.     <servlet-name>Faces Servlet</servlet-name>
15.     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
16.     <load-on-startup> 1 </load-on-startup>
17.   </servlet>
18.
19.   <!-Le mapping de la servlet -->
20.   <servlet-mapping>
21.     <servlet-name>Faces Servlet</servlet-name>
22.     <url-pattern>*.faces</url-pattern>
23.   </servlet-mapping>
24. </web-app>
```

Le tag <servlet> permet de définir une servlet et plus particulièrement dans ce cas de préciser la servlet qui sera utilisée comme contrôleur dans l'application. Le plus simple est d'utiliser la servlet fournie avec l'implémentation de référence javax.faces.webapp.FacesServlet. Le tag <load-on-startup> avec comme valeur 1 permet de demander le chargement de cette servlet au lancement de l'application.

Le tag <servlet-mapping> permet de préciser le mapping des URLs qui seront traitées par la servlet. Ce mapping peut prendre deux formes :

- mapping par rapport à une extension : exemple <url-pattern>*.faces</url-pattern>.
- mapping par rapport à un préfixe : exemple <url-pattern>/faces/*</url-pattern>.

Les URL utilisées pour des pages mettant en oeuvre JSF doivent obligatoirement passer par cette servlet. Ces URLs peuvent être de deux formes selon le mapping défini.

Exemple :

- http://localhost:8080/nom_webapp/index.faces
- http://localhost:8080/nom_webapp/faces/index.jsp

Dans les deux cas, c'est la servlet utilisée comme contrôleur qui va déterminer le nom de la page JSP à utiliser.

Le paramètre de contexte javax.faces.STATE_SAVING_METHOD permet de préciser le mode d'échange de l'état de l'arbre des composants de la page. Deux valeurs sont possibles :

- client :
- server :

Il est possible d'utiliser l'extension .jsf pour les fichiers JSP utilisant JSF à condition de correctement configurer le fichier web.xml dans ce sens. Pour cela deux choses sont à faire :

- il faut demander le mapping des URL terminant par .jsf par la servlet

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

- il faut préciser à la servlet le suffix par défaut à utiliser

```
<context-param>
<param-name>javax.faces.DEFAULT_SUFFIX</param-name>
<param-value>.jsf</param-value>
</context-param>
```

Le démarrage d'une application directement avec une page par défaut utilisant JSF ne fonctionne pas correctement. Il est préférable d'utiliser une page HTML qui va effectuer une redirection vers la page d'accueil de l'application

Exemple :

```
01. <html>
02.   <head>
03.     <meta http-equiv="Refresh" content= "0; URL=index.faces"/>
04.     <title>Demarrage de l'application</title>
05.   </head>
06.   <body>
07.     <p>D&eacute;marrage de l'application ...</p>
08.   </body>
09. </html>
```

Il suffit alors de préciser dans le fichier web.xml que cette page est la page par défaut de l'application.

Exemple :

```
1. ...
2. <welcome-file-list>
3.   <welcome-file>index.htm</welcome-file>
4. </welcome-file-list>
5. ...
```

69.5.2. Le fichier faces-config.xml

Le plus simple est de placer ce fichier dans le répertoire WEB-INF de l'application Web.

Il est aussi possible de préciser son emplacement dans un paramètre de contexte nommé javax.faces.application.CONFIG_FILES dans le fichier web.xml. Il est possible par ce biais de découper le fichier de configuration en plusieurs morceaux. Ceci est particulièrement intéressant pour de grosses applications car un seul fichier de configuration peut dans ce cas devenir très gros. Il suffit de préciser chacun des fichiers séparés par une virgule dans le tag <param-value>.

Exemple :

```
1. ...
2. <context-param>
3.   <param-name>javax.faces.application.CONFIG_FILES</param-name>
4.   <param-value>
5. /WEB-INF/ma-faces-config.xml, /WEB-INF/navigation-faces.xml, /WEB-INF/beans-faces.xml
6.   </param-value>
7. </context-param>
8. ...
```

Ce fichier au format XML permet de définir et de fournir des valeurs d'initialisation pour des ressources nécessaires à l'application utilisant JSF.

Ce fichier doit impérativement respecter la DTD proposée par les spécifications de JSF :

http://java.sun.com/dtd/web-facesconfig_1_0.dtd

Le tag racine du document XML est le tag <face-config>. Ce tag peut avoir plusieurs tags fils :

Tag	Rôle
application	permet de préciser ou de remplacer des éléments de l'application
factory	permet de remplacer des fabriques par des fabriques personnalisées de certaines ressources (FacesContextFactory, LifeCycleFactory, RenderKitFactory, ...)
component	définit un composant graphique personnalisé
converter	définit un convertisseur pour encoder/décoder les valeurs des composants graphiques (conversion de String en Object et vice versa)
managed-bean	définit un objet utilisé par un composant qui est automatiquement créé, initialisé et stocké dans une portée précisée
navigation-rule	définit les règles qui permettent de déterminer l'enchaînement des traitements de l'application
referenced-bean	
render-kit	définit un kit pour le rendu des composants graphiques
lifecycle	
validator	définit un validateur personnalisé de données saisies dans un composant graphique

Ces tags fils peuvent être utilisés 0 ou plusieurs fois dans le tag <face-config>.

Le tag <application> permet de préciser des informations sur les entités utilisées par l'internationalisation et/ou de remplacer des éléments de l'application.

Les éléments à remplacer peuvent être : ActionListener, NavigationHandler, ViewHandler, PropertyResolver, VariableResolver. Ceci n'est utile que si la version fournie dans l'implémentation ne correspond pas aux besoins et doit être personnalisée par l'écriture d'une classe dédiée.

Le tag fils <message-bundle> permet de préciser le nom de base des fichiers de ressources utiles à l'internationalisation.

Le tag <locale-config> permet de préciser les locales qui sont supportées par l'application. Il faut utiliser autant de tags fils <supported-locale> que de locales supportées. Le tag fil <default-locale> permet de préciser la locale par défaut.

Exemple :

```
01. ...
02. <application>
03.   <message-bundle>com.jmdoudoux.test.jsf.monapp.bundles.Messages</message-bundle>
04.   <locale-config>
05.     <default-locale>fr</default-locale>
06.     <supported-locale>en</supported-locale>
07.   </locale-config>
08. </application>
09. ...
```

69.6. Les beans

Les beans sont largement utilisés dans une application JSF notamment pour permettre l'échange de données entre les différentes entités et le traitement des événements.

Les beans sont des classes qui respectent une spécification particulière notamment la présence :

- de getters et de setters qui respectent une convention de nommage particulière pour les attributs
- un constructeur par défaut sans arguments

69.6.1. Les beans managés (managed bean)

Les beans managés sont des javabeans dont le cycle de vie va être contrôlé par le framework JSF en fonction des besoins et du paramétrage fourni dans le fichier de configuration.

Dans le fichier de configuration, chacun de ces beans doit être déclaré avec un tag <managed-bean>. Ce tag possède trois tags fils obligatoires :

- <managed-bean-name> : le nom attribué au bean (celui qui sera utilisé lors de son utilisation)
- <managed-bean-class> : le type pleinement qualifié de la classe du bean
- <managed-bean-scope> : précise la portée dans laquelle le bean sera stocké et donc utilisable

La portée peut prendre les valeurs suivantes :

- request : cette portée est limitée entre l'émission de la requête et l'envoi de la réponse. Les données stockées dans cette portée sont utilisables lors d'un transfert vers une autre page (forward). Elles sont perdues lors d'une redirection (redirect).
- session : cette portée permet la circulation de données entre plusieurs échanges avec un même client
- application : cette portée permet l'accès à des données pour toutes les pages d'une même application quelque soit l'utilisateur

Exemple :

```
1. ...
2. <managed-bean>
3.   <managed-bean-name>login</managed-bean-name>
4.   <managed-bean-class>com.jmd.test.jsf.LoginBean</managed-bean-class>
5.   <managed-bean-scope>session</managed-bean-scope>
6. </managed-bean>
7. ...
```

Il est possible de fournir des valeurs par défaut aux propriétés en utilisant le tag <managed-property>. Ce tag possède deux tags fils :

- <property-name> : nom de la propriété du bean
- <value> : valeur à associer à la propriété

Exemple :

```
01. ...
02. <managed-bean>
03.   <managed-bean-name>login</managed-bean-name>
04.   <managed-bean-class>com.jmd.test.jsf.LoginBean</managed-bean-class>
05.   <managed-bean-scope>session</managed-bean-scope>
06.   <managed-property>
07.     <property-name>nom</property-name>
08.     <value>test</value>
09.   </managed-property>
10. </managed-bean>
11. ...
```

Lorsque que le bean sera instancié, JSF appellera automatiquement les setters des propriétés identifiées dans des tags <managed-property> avec leurs valeurs <value> respectives.

Pour initialiser la propriété à null, il faut utiliser le tag <null-value>

Exemple :

```
1. ...
2. <managed-property>
3.   <property-name>nom</property-name>
4.   <null-value>
5. </managed-property>
6. ...
```

Ces informations seront utilisées par JSF pour automatiser la création ou la récupération d'un bean lorsque celui-ci sera utilisé dans l'application.

Le grand intérêt de ce mécanisme est de ne pas avoir à se soucier de l'instanciation du bean ou de sa recherche dans la portée puisque c'est le framework qui va s'en occuper de façon transparente.

69.6.2. Les expressions de liaison de données d'un bean

Il est toujours nécessaire dans la partie présentation d'obtenir la valeur d'une donnée d'un bean pour par exemple l'afficher.

JSF propose une syntaxe basée sur des expressions qui facilitent l'utilisation des valeurs d'un bean. Ces expressions doivent être délimitées par #{ et }.

Basiquement une expression est composée du nom du bean suivi du nom de la propriété désirée séparés par un point.

Exemple :

```
1. <h:inputText value="#{login.nom}"/>
```

Cet exemple affecte la valeur de l'attribut nom du bean login au composant de type saisie de texte. Dans ce cas précis, c'est aussi cet attribut de ce bean qui recevra la valeur saisie lorsque la page sera envoyée au serveur.

En fonction du contexte le résultat de l'évaluation peut conduire à l'utilisation du getter (par exemple pour afficher la valeur) ou du setter (pour affecter la valeur après un envoi de la page). C'est JSF qui le détermine en fonction du contexte.

La notation par point peut être remplacée par l'utilisation de crochets. Dans ce cas, le nom de la propriété doit être mis entre simples ou doubles quotes dans les crochets.

Exemple :

```
1. login.nom
2. login["nom"]
3. login['nom']
```

Ces trois expressions sont rigoureusement identiques. Cette syntaxe peut être plus pratique lors de la manipulation de collections mais elle est obligatoire lorsque la propriété contient un point.

Exemple :

```
1. msg["login.titre"]
```

L'utilisation des quotes simples ou doubles est équivalente car il faut les imbriquer par exemple lors de leur utilisation comme valeur de l'attribut d'un composant.

Exemple :

```
1. <h:inputText value='#{login["nom"]}'/>
2. <h:inputText value="#{login['nom']}"/>
```

Attention, la syntaxe utilisée par JSF est proche mais différente de celle proposée par JSTL : JSF utilise le délimiteur #{ ... } et JSTL utilise le délimiteur ${ ... } .

JSF définit un ensemble de variables prédéfinies et utilisables dans les expressions de liaison de données :

Variables	Rôle
header	une collection de type Map encapsulant les éléments définis dans les paramètres de l'en-tête de la requête http (seule la première valeur est renvoyée)
header-values	une collection de type Map encapsulant les éléments définis dans les paramètres de l'en-tête de la requête http (toutes les valeurs sont renvoyées sous la forme d'un tableau)
param	une collection de type Map encapsulant les éléments définis dans les paramètres de la requête http (seule la première valeur est renvoyée)
param-values	une collection de type Map encapsulant les éléments définis dans les paramètres de la requête http (toutes les valeurs sont renvoyées sous la forme d'un tableau)
cookies	une collection de type Map encapsulant les éléments définis dans les cookies
initParam	une collection de type Map encapsulant les éléments définis dans les paramètres d'initialisation de l'application
requestScope	une collection de type Map encapsulant les éléments définis dans la portée request
sessionScope	une collection de type Map encapsulant les éléments définis dans la portée session
applicationScope	une collection de type Map encapsulant les éléments définis dans la portée application
facesContext	une instance de la classe FacesContext
View	une instance de la classe UIViewRoot qui encapsule la vue

Lorsque qu'une variable est utilisée dans une expression, JSF recherche dans la liste des variables prédéfinies, puis recherche une instance dans la portée request, puis dans la portée session et enfin dans la portée application. Si aucune instance n'est trouvée, alors JSF crée une nouvelle instance en tenant compte des informations du fichier de configuration. Cette instanciation est réalisée par un objet de type VariableResolver de l'application.

La syntaxe des expressions possède aussi quelques opérateurs :

Opérateurs	Rôle	Exemple
+ - * / % div mod	opérateurs arithmétiques	
< <= > >= == !=		

lt le gt ge eq ne	opérateurs de comparaisons	
&& || ! and or not	opérateurs logiques	<h:inputText rendered="#{!monBean.affichable}" />
Empty	opérateur vide : un objet null, une chaîne vide, un tableau ou une collection sans élément	
? :	opérateur ternaire de test	

Il est possible de concaténer les résultats des évaluations de plusieurs expressions simplement en les plaçant les uns à la suite des autres.

Exemple :

```
1. <h:outputText value="#{messages.salutation}, #{utilisateur.nom}!"/>
```

Il est parfois nécessaire d'évaluer une expression dans le code des objets métiers pour obtenir sa valeur. Comme tous les composants sont stockés dans le FaceContext, il est possible d'accéder à cet objet pour obtenir les informations désirées. Il est d'abord nécessaire d'obtenir l'instance courante de l'objet FaceContext en utilisant la méthode statique getCurrentInstance().

Exemple :

```
1. FacesContext context = FacesContext.getCurrentInstance();
2. ValueBinding binding = context.getApplication().createValueBinding("#{login.nom}");
3. String nom = (String) binding.getValue(context);
```

69.6.3. Les Backing beans

Les beans de type backing bean sont spécialement utilisés avec JSF pour encapsuler tout ou partie des composants d'une page et ainsi faciliter leur accès notamment lors des traitements.

Ces beans sont particulièrement utiles durant des traitements réalisés lors de validations ou de gestion d'événements car ils permettent un accès aux composants dont ils possèdent une référence.

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. import javax.faces.component.UIInput;
04.
05. public class LoginBean {
06.
07.   private UIInput composantNom;
08.   private String nom;
09.   private String mdp;
10.
11.   public UIInput getComposantNom() {
12.     return composantNom;
13.   }
14.
15.   public void setComposantNom(UIInput input) {
16.     composantNom = input;
17.   }
18.
19.   public String getNom() {
20.     return nom;
21.   }
22. ...
23. }
```

Dans la vue, il est nécessaire de lier un composant avec son attribut correspondant dans le backing bean. L'attribut binding d'un composant permet de réaliser cette liaison.

Exemple :

```
1. <h:inputText value="#{login.nom}" binding="#{login.composantNom}" />
```

69.7. Les composants pour les interfaces graphiques

JSF propose un ensemble de composants serveurs pour faciliter le développement d'interfaces graphiques utilisateur.

Pour les composants, JSF propose :

- un ensemble de classes qui gèrent le comportement et l'état d'un composant
- un modèle pour assurer le rendu du composant pour un type d'application (par exemple HTML)
- un modèle de gestion des événements émis par le composant reposant sur le modèle des listeners
- la possibilité d'associer à un composant un composant de conversion de données ou de validation des données

Tous ces composants héritent de la classe abstraite UIComponentBase.

JSF propose 12 composants de base :

UICommand	Composant qui permet de réaliser une action émettant un événement
UIForm	Composant qui regroupe d'autres composants dont l'état sera renvoyé au serveur
UIGraphic	Composant qui représente une image
UIInput	Composant qui permet de saisir des données
UIOutput	Composant qui permet d'afficher des données
UIPanel	Composant qui regroupe d'autres composants à afficher sous la forme d'un tableau
UIParameter	
UISelectItem	Composant qui représente un élément sélectionné dans un ensemble
UISelectItems	Composant qui représente un ensemble d'éléments
UISelectBoolean	Composant qui permet de sélectionner parmi deux états
UISelectMany	Composant qui permet de sélectionner plusieurs éléments dans ensemble
UISelectOne	Composant qui permet de sélectionner un seul élément dans ensemble

Ces classes sont des javabeans qui définissent les fonctionnalités de base des composants permettant la saisie et la sélection de données.

Chacun de ces composants possède un type, un identifiant, une ou plusieurs valeurs locales et des attributs. Ils sont extensibles et il est même possible de créer ses propres composants.

Le comportement de ces composants repose sur le traitement d'événements respectant le modèle de gestion des événements de JSF.

Ces classes ne sont pas utilisées directement : elles sont utilisées par la bibliothèque de tags personnalisés qui se charge de les instancier et de leur associer le modèle de rendu adéquat.

Ces classes ne prennent pas en charge le rendu du composant. Par exemple, un objet de type UICommand peut être rendu en HTML sous la forme d'un lien hypertexte ou d'un bouton de formulaire.

69.7.1. Le modèle de rendu des composants

Pour chaque composant, il est possible de définir un ou plusieurs modèles qui se chargent du rendu de ce composant dans un contexte client particulier (par exemple HTML).

L'association entre un composant et son modèle de rendu est réalisée dans un RenderKit : il précise pour chaque composant le ou les modèles de rendu à utiliser. Par exemple, un objet de type UISelectOne peut être rendu sous la forme d'un ensemble de boutons radio, d'une liste ou d'une liste déroulante. Chacun de ces rendus est défini par un objet de type Renderer.

L'implémentation de référence propose un seul modèle de rendu pour les composants qui génèrent de l'HTML.

Ce modèle favorise la séparation entre l'état, le comportement d'un composant et sa représentation finale.

Le modèle de rendu permet de définir la représentation visuelle des composants. Chaque composant peut être rendu de plusieurs façons avec plusieurs modèles de rendu. Par exemple, un composant de type UICommand peut être rendu sous la forme d'un bouton ou d'un lien hypertexte. Dans cet exemple, le rendu est HTML mais il est possible d'utiliser un autre système de rendu comme XML ou WML.

Le modèle de rendu met un oeuvre un ou plusieurs kits de rendus.

69.7.2. L'utilisation de JSF dans une JSP

Pour une utilisation dans une JSP, l'implémentation de référence propose deux bibliothèques de tags personnalisés :

- core : cette bibliothèque contient des fonctionnalités de bases ne générant aucun rendu. L'utilisation de cette bibliothèque est obligatoire car elle contient notamment l'élément view
- html : cette bibliothèque se charge des composants avec un rendu en HTML

Pour utiliser ces deux bibliothèques, il est nécessaire d'utiliser une directive taglib pour chacune d'elle au début de la page JSP.

Exemple :

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

Le préfixe est libre mais par convention ce sont ceux fournis dans l'exemple qui sont utilisés.

Le tag <view> est obligatoire dans toutes pages utilisant JSF. Cet élément va contenir l'état de l'arborescence des composants de la page si l'application est configurée pour stocker l'état sur le client.

Le tag <form> génère un tag HTML form qui définit un formulaire.

Exemple :

```
01. <html>
02. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
03. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
04. <f:view>
05. <head>
06.   <title>Application de tests avec JSF</title>
07. </head>
08. <body>
09.   <h:form>
10.   ...
11.   </h:form>
12. </body>
13. </f:view>
14. </html>
```

69.8. La bibliothèque de tags Core

Cette bibliothèque est composée de 18 tags.

Tag	Rôle
actionListener	ajouter un listener pour une action sur un composant
attribute	ajouter un attribut à un composant
convertDateTime	ajouter un convertisseur de type DateTime à un composant
convertNumber	ajouter un convertisseur de type numérique à un composant
facet	définit un élément particulier d'un composant
loadBundle	charger un fichier contenant les chaînes de caractères d'une locale dans une collection de type Map
param	ajouter un paramètre à un composant
selectitem	définir l'élément sélectionné dans un composant permettant de faire un choix
selectitems	définir les éléments sélectionnés dans un composant permettant de faire un choix
subview	définir une sous-vue
verbatim	ajouter un texte brut à la vue
view	définir une vue
validator	ajouter un validateur à un composant
validateDoubleRange	ajouter un validateur de type « plage de valeurs réelles » à un composant
validateLength	ajouter un validateur de type « taille de la valeur » à un composant
validateLongRange	ajouter un validateur de type « plage de valeurs entières » à un composant
valueChangeListener	ajouter un listener pour un changement de valeur sur un composant

La plupart de ces tags permettent d'ajouter des objets à un composant. Leur utilisation sera détaillée tout au long de ce chapitre.

69.8.1. Le tag <selectItem>

Ce tag représente un élément dans un composant qui peut en contenir plusieurs.

Les attributs de base sont les suivants :

Attribut	Rôle
itemValue	contient la valeur de l'élément
itemLabel	contient le libellé de l'élément
itemDescription	contient une description de l'élément (utilisé uniquement par les outils de développement)
itemDisabled	contient l'état de l'élément
binding	contient le nom d'une méthode qui renvoie un objet de type javax.faces.model.SelectItem
id	contient l'identifiant du composant
value	contient une expression qui désigne un objet de type javax.faces.model.SelectItem

Exemple :

```
1. <f:selectItem value="#{test.elementSelectionne}"/>
```

L'attribut value attend en paramètre une expression désignant une méthode qui renvoie un objet de type SelectItem. Cet objet encapsule l'objet de la liste qui sera sélectionné.

Exemple :

```
1. ...
2. public SelectItem getElementSelectionne() {
3.   return new SelectItem("Element 1");
4. }
5. ...
```

La classe SelectItem possède quatre constructeurs qui permettent de définir les différentes propriétés qui composent l'élément.

69.8.2. Le tag <selectItems>

Ce tag représente une collection d'éléments dans un composant qui peut en contenir plusieurs.

Ce tag est particulièrement utile car il évite d'utiliser autant de tags selectItem que d'éléments à définir.

Exemple :

```
1. ...
2. <h:selectOneRadio>
3.   <f:selectItems value="#{test.listeElements}"/>
4. </h:selectOneRadio>
5. ...
```

La collection d'objets de type SelectItem peut être soit une collection soit un tableau.

Exemple : avec un tableau d'objects de type SelectItem

```
01. package com.jmd.test.jsf;
02.
03. import javax.faces.model.SelectItem;
04.
05. public class TestBean {
06.
07.   private SelectItem[] elements = {
08.     new SelectItem(new Integer(1), "Element 1"),
09.     new SelectItem(new Integer(2), "Element 2"),
10.     new SelectItem(new Integer(3), "Element 3"),
11.     new SelectItem(new Integer(4), "Element 4"),
12.   };
13.
14.   public SelectItem[] getListeElements() {
15.     return elements;
```

```
16.   }
17.
18. ...
19. }
```

La collection peut être de type Map : dans ce cas le framework associe la clé de chaque occurrence à la propriété itemValue et la valeur à la propriété itemLabel

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. import java.util.HashMap;
04. import java.util.Map;
05.
06. import javax.faces.model.SelectItem;
07.
08. public class TestBean {
09.
10.   private Map elements = null;
11.
12.   public Map getListeElements() {
13.     if (elements == null) {
14.       elements = new HashMap();
15.       elements.put("Element 1", new Integer(1));
16.       elements.put("Element 2", new Integer(2));
17.       elements.put("Element 3", new Integer(3));
18.       elements.put("Element 4", new Integer(4));
19.     }
20.     return elements;
21.   }
22.
23.   public SelectItem getElementSelectionne() {
24.     return new SelectItem("Element 1");
25.   }
26.
27.   ...
28. }
```

69.8.3. Le tag <verbatim>

Ce tag permet d'insérer du texte dans la vue.

Son utilisation est obligatoire dans le corps des tags JSF pour insérer autre chose qu'un tag JSF. Par exemple, pour insérer un tag HTML dans le corps d'un tag JSF, il est obligatoire d'utiliser le tag <verbatim>.

Les tags suivants peuvent avoir un corps : commandLink, outputLink, panelGroup, panelGrid et dataTable.

Exemple :

```
1. <h:outputLink value="http://java.sun.com" title="Java">
2.   <f:verbatim>
3.     Site Java de Sun
4.   </f:verbatim>
5. </h:outputLink>
```

Il est possible d'utiliser le tag <outputText> à la place du tag <verbatim>.

69.8.4. Le tag <attribute>

Ce tag permet de fournir un attribut quelconque à un composant puisque chaque composant peut stocker des attributs arbitraires.

Ce tag possède deux attributs :

Attribut	Rôle
Name	nom de l'attribut
Value	valeur de l'attribut

Dans le code d'un composant, il est possible d'utiliser la méthode getAttributes() pour obtenir une collection de type Map des attributs du composant.

Ceci offre un mécanisme souple pour fournir des paramètres sans être obligé de créer un nouveau composant ou de modifier un composant existant en lui ajoutant un ou plusieurs attributs.

69.8.5. Le tag <facet>

Ce tag permet de définir des éléments particuliers d'un composant.

Il est par exemple utilisé pour définir les lignes d'en-tête et de pied de page des tableaux.

Ce tag possède plusieurs attributs :

Attribut	Rôle
Name	Permet de préciser le type de l'élément généré par le tag Les valeurs possibles sont header et footer

Exemple :

```
<h:column>
  <f:facet name="header">
    <h:outputText value="Nom" />
  </f:facet>
  <h:outputText value="#{personne.nom}"/>
</h:column>
```

69.9. La bibliothèque de tags Html

Cette bibliothèque est composée de 25 tags qui permettent la réalisation de l'interface graphique de l'application.

Tag	Rôle
form	le tag <form> HTML
commandButton	un bouton
commandLink	un lien qui agit comme un bouton
graphicImage	une image
inputHidden	une valeur non affichée
inputSecret	une zone de saisie de texte monoligne dont la valeur dont la valeur n'est pas lisible
inputText	une zone de saisie de texte monoligne
inputTextarea	une zone de saisie de texte multiligne
outputLink	un lien
outputFormat	du texte affiché avec des valeurs fournies en paramètre
outputText	du texte affiché
panelGrid	un tableau
panelGroup	un panneau permettant de regrouper plusieurs composants
selectBooleanCheckbox	une case à cocher
selectManyCheckbox	un ensemble de cases à cocher
selectManyListbox	une liste déroulante où plusieurs éléments sont sélectionnables
selectManyMenu	un menu où plusieurs éléments sont sélectionnables
selectOneListbox	une liste déroulante où un seul élément est sélectionnable
selectOneMenu	un menu où un seul élément est sélectionnable
selectOneRadio	un ensemble de boutons radio
dataTable	une grille proposant des fonctionnalités avancées
column	une colonne d'une grille
message	le message d'erreur lié à un composant
messages	les messages d'erreur liés à tous les composants

69.9.1. Les attributs communs

Ces tags possèdent des attributs communs pouvant être regroupés en trois catégories :

- les attributs de base
- les attributs liés à HTML
- les attributs liés à JavaScript

Chaque tag utilise ou non chacun de ces attributs.

Les attributs de base sont les suivants :

Attribut	Rôle
id	contient l'identifiant du composant
binding	permet l'association avec un backing bean
rendered	contient un booléen qui indique si le composant doit être affiché
styleClass	contient le nom d'une classe CSS à appliquer au composant
value	contient la valeur du composant
valueChangeListener	permet l'association à une méthode qui va traiter les changements de valeurs
converter	contient une classe de conversion des données de chaîne de caractères en objet et vice versa
validator	contient une classe de validation des données
required	contient un booléen qui indique si une valeur doit obligatoirement être saisie

L'attribut id est très important car il permet d'avoir accès :

- au tag dans le code de la vue par d'autres tags
  <h:inputText id="nom" required="true"/>

<h:message for="nom"/>

- au tag dans le code JavaScript de la vue
- dans le code Java des objets métiers.
  UIComponent component = event.getComponent().findComponent("nomComposant");

L'attribut binding permet d'associer le composant avec un champ d'une classe de type bean. Un tel bean est nommé backing bean dans une application JSF.

Exemple :

```
01. ...
02. <h:inputText value="#{login.nom}" id="nom" required="true" binding="#{login.inputTextNom}"/>
03. ..
04.
05. ...
06. import javax.faces.component.UIComponent;
07.
08. public class LoginBean {
09.   private String nom;
10.
11.   private UIComponent inputTextNom;
12.
13.   public UIComponent getInputTextNom() {
14.     return inputTextNom;
15.   }
16.
17.   public void setInputTextNom(UIComponent inputTextNom) {
18.     this.inputTextNom = inputTextNom;
19.   }
20. ...
```

L'attribut value permet de préciser la valeur d'un tag. Cette valeur peut être fournie sous deux formes :

- en dur dans le code :
  <h:outputText value="Bonjour"/>
- en utilisant une expression de liaison de données :
  <h:inputText value="#{login.nom}"/>

L'attribut converter permet de préciser une classe qui va convertir la valeur d'un objet en chaîne de caractères et vice versa. L'utilisation de cet attribut est détaillée dans une des sections suivantes.

L'attribut validator permet de préciser une classe qui va réaliser des contrôles de validation sur la valeur saisie. L'utilisation de cet attribut est

détaillée dans une des sections suivantes.

L'attribut styleClass permet de préciser le nom d'un style défini dans une feuille de style CSS qui sera appliqué au composant.

Exemple : le fichier monstyle.css

```
.titre {
color:red;
}
```

Dans la vue, il faut inclure la feuille de style dans la partie en-tête de la page HTML.

Exemple :

```
...
<link href="monstyle.css" rel="stylesheet" type="text/css"/>
...
<h:outputText value="#{msg.login_titre}" styleClass="titre"/>
...
```

L'attribut renderer permet de préciser si le composant sera affiché ou non dans la vue. La valeur de l'attribut peut être obtenue dynamiquement par l'utilisation du langage d'expression.

Exemple :

```
<h:panelGrid rendered='#{listepersonnes.nbOccurrences gt 0}'/>
```

Les principaux attributs liés à HTML sont les suivants :

Attribut	Rôle
accesskey	contient le raccourci clavier pour donner le focus au composant
alt	contient le texte alternatif pour les composants non textuels
border	contient la taille de la bordure en pixel
disabled	permet de désactiver le composant
maxlength	contient le nombre maximum de caractères saisis
readonly	permet de rendre une zone de saisie en lecture seule
rows	contient le nombre de lignes visibles pour une zone de saisie multiligne
shape	contient la définition d'une région
size	contient la taille de la zone de saisie
style	contient le style CSS à utiliser
target	contient le nom de la frame cible pour l'affichage de la page
title	contient le titre du composant généralement transformé en une bulle d'aide
width	contient la largeur du composant

Le rôle de la plupart de ces tags est identique à leurs homologues définis dans HTML 4.0.

L'attribut style permet de définir un style CSS qui sera appliqué au composant. Cet attribut contient directement la définition du style à la différence de l'attribut styleClass qui contient le nom d'une classe CSS définie dans une feuille de style. Il est préférable d'utiliser l'attribut styleClass plutôt que l'attribut style afin de faciliter la maintenance de la charte graphique.

Exemple :

```
<h:outputText value="#{login.nom}" style="color:red;"/>
```

Les attributs liés à JavaScript sont :

Attribut	Rôle
onblur	perte du focus
onchange	changement de la valeur
onclick	clic du bouton de la souris sur le composant

ondblclick	double-clic du bouton de la souris sur le composant
onfocus	réception du focus
onkeydown	une touche est enfoncée
onkeypress	appui sur une touche
onkeyup	une touche est relachée
onmousedown	
onmousemove	déplacement du curseur de la souris sur le composant
onmouseout	déplacement du cuseur de la souris hors du composant
onmouseover	passage de la souris au-dessus du composant
onmouseup	le bouton de la souris est relaché
onreset	réinitialisation du formulaire
onselect	sélection du texte dans une zone de saisie
onsubmit	soumission du formulaire

69.9.2. Le tag <form>

Ce tag représente un formulaire HTML.

Il possède les attributs suivants :

Attributes	Rôle
binding, id, rendered, styleClass	attributs communs de base
accept, acceptcharset, dir, enctype, lang, style, target, title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onreset, onsubmit	attributs communs liés aux événements JavaScript

Il est préférable de définir explicitement l'attribut id pour permettre son exploitation notamment dans le code JavaScript, sinon un id est généré automatiquement.

Ceci est d'autant plus important que les id des composants intégrés dans le formulaire sont préfixés par l'id du formulaire suivi du caractère deux-points. Il faut tenir compte de ce point lors de l'utilisation de code JavaScript faisant référence à un composant.

69.9.3. Les tags <inputText>, <inputTextarea>, <inputSecret>

Ces trois tags permettent de générer des composants pour la saisie de données.

Les attributs de ces tags sont les suivants :

Attributs	Rôle
cols	définir le nombre de colonnes (pour le composant inputTextarea uniquement)
immediate	permettre de demander d'ignorer les étapes de validation des données
redisplay	permettre de réafficher le contenu lors du réaffichage de la page (pour le composant inputSecret uniquement)
required	rendre obligatoire la saisie d'une valeur
rows	définir le nombre de lignes affichées (pour le composant inputTextarea uniquement)
valueChangeListener	préciser une classe de type listener lors du changement de la valeur
binding, converter, id, rendered, required, styleClass, value, validator	attributs communs de base
accesskey, alt, dir, disabled,lang, maxlength, readonly, size, style, tabindex, title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onselect	attributs communs liés aux événements JavaScript

Exemple :

```
01. <html>
02. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
03. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
04. <f:view>
05. <head>
06.   <title>Saisie des données</title>
07. </head>
08. <body>
09.   <h:form>
10.     <h3>Saisie des données</h3>
11.     <p><h:inputText size="20" /></p>
12.     <p><h:inputTextarea rows="3" cols="20" /></p>
13.     <p><h:inputSecret size="20" /></p>
14.   </h:form>
15. </body>
16. </f:view>
17. </html>
```

Résultat

texte

ligne1
ligne2
ligne3

******

69.9.4. Le tag <ouputText> et <outputFormat>

Ces deux tags permettent d'insérer dans la vue une valeur sous la forme d'une chaîne de caractères. Par défaut, ils ne génèrent pas de tags HTML mais insèrent simplement la valeur dans la vue sauf si un style CSS est précisé avec l'attribut style ou styleClass. Dans ce cas, la valeur est contenue dans un tag HTML <span>.

Les attributs de ces deux tags sont :

Attributs	Rôle
escape	booléen qui précise si certains caractères de la valeur seront encodés ou non. La valeur par défaut est false.
binding, converter, id, rendered, styleClass, value	attributs communs de base
style, title	attributs communs liés à HTML

L'attribut escape est particulièrement utile pour encoder certains caractères spéciaux avec leur code HTML correspondant.

Exemple :

```
1. <h:outputText escape="true" value="Nombre d'occurrences > 200"/>
```

Le tag outputText peut être utilisé pour générer du code HTML en valorisant l'attribut escape à false.

Exemple :

```
1. <p><h:outputText escape="false" value="<H2>Saisie des données</H2>"/></p>
2. 
3. <p><h:outputText escape="true" value="<H2>Saisie des données</H2>"/></p>
```

Résultat :

Saisie des données

<H2>Saisie des données</H2>

Le tag outputFormat permet de formater une chaîne de caractères avec des valeurs fournies en paramètres.

Exemple :

```
<p>
  <h:outputFormat value="La valeur doit être entre {0} et {1}.">
    <f:param value="1"/>
    <f:param value="9"/>
  </h:outputFormat>
</p>
```

Résultat

La valeur doit être entre 1 et 9.

Ce composant utilise la classe java.text.MessageFormat pour formater le message. L'attribut value doit donc contenir une chaîne de caractères utilisable par cette classe.

Le tag <param> permet de fournir la valeur de chacun des paramètres.

69.9.5. Le tag <graphicImage>

Ce composant représente une image : il génère un tag HTML <img>.

Les attributs de ce tag sont les suivants :

Attributs	Rôle
binding, id, rendered, styleClass, value	Attributs communs de base
alt, dir, height, ismap, lang, longdesc, style, title, url, usemap, width	Attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	Attributs communs liés aux événements JavaScript

Les attributs value et URL peuvent préciser l'URL de l'image.

Exemple :

```
<p><h:graphicImage value="/images/erreur.jpg" /></p>
<p><h:graphicImage url="/images/warning.jpg" /></p>
```

Résultat

69.9.6. Le tag <inputHidden>

Ce composant représente un champ caché dans un formulaire.

Les attributs sont les suivants :

Attributs	Rôle
binding, converter, id, immediate, required, validator, value, valueChangeListener	attributs communs de base

Exemple :

```
1. <h:inputHidden value="#{login.nom}" />
```

Résultat dans le code HTML:

```
1. ...
2.      <input type="hidden" name="_id0:_id12" value="test" />
3. ...
```

69.9.7. Le tag <commandButton> et <commandLink>

Ces composants représentent respectivement un bouton de formulaire et un lien qui déclenche une action. L'action demandée sera traitée par le framework JSF.

Les attributs sont les suivants :

Attributs	Rôle
action	peut être une chaîne de caractères ou une méthode qui renvoie une chaîne de caractères qui sera traitée par le navigation handler.
actionListener	précise une méthode possédant une signature void nomMethode(ActionEvent) qui sera exécutée lors d'un clic
image	URL tenant compte du contexte de l'application pour l'image qui sera utilisée à la place du bouton (uniquement pour le tag commandButton)
type	type de bouton généré : button, submit, reset (uniquement pour le tag commandButton)
value	le texte affiché par le bouton ou le lien
accesskey, alt, binding, id, lang, rendered, styleClass	attributs communs de base
coords (uniquement pour le tag commandLink), dir, disabled, hreflang (uniquement pour le tag commandLink), lang, readonly, rel (uniquement pour le tag commandLink), rev (uniquement pour le tag commandLink), shape (uniquement pour le tag commandLink), style, tabindex, target (uniquement pour le tag commandLink), title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs liés aux événements JavaScript

Il est possible d'insérer dans le corps du tag <commandLink> d'autres composants qui feront partie intégrante du lien comme par exemple du texte ou une image.

Exemple :

```
01. <p>
02.   <h:commandLink>
03.     <h:outputText value="Valider"/>
04.   </h:commandLink>
05. </p>
06. <p>
07.   <h:commandLink>
08.     <h:graphicImage value="/images/oeil.jpg"/>
09.   </h:commandLink>
10. </p>
```

Résultat

Valider

Il est aussi possible de fournir un ou plusieurs paramètres qui seront envoyés dans la requête en utilisant le tag <param> dans le corps du tag

Exemple :

```
1. <h:commandLink>
2.   <h:outputText value="Selectionner"/>
3.   <f:param name="id" value="1"/>
4. </h:commandLink>
```

Résultat dans le page HTML générée :

```
1. <a href="#" onclick="document.forms['_id0']['_id0:_idcl'].value='_id0:_id15';
2.       document.forms['_id0'].submit(); return false;">
3.     mg src="/test_JSF/images/oeil.jpg" alt="" /></a>
```

Le tag <commandLink> génère dans la vue du code JavaScript pour soumettre le formulaire lors d'un clic.

69.9.8. Le tag <ouputLink>

Ce composant représente un lien direct vers une ressource dont la demande ne sera pas traitée par le framework JSF.

Les attributs sont les suivants :

Attributs	Rôle
accesskey, binding, converter, id, lang, rendered, styleClass, value	attributs communs de base
charset, coords, dir, hreflang, lang, rel, rev, shape, style, tabindex, target, title, type	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	attributs communs liés aux événements JavaScript

L'attribut value doit contenir l'URL qui sera utilisée dans l'attribut href du lien HTML. Si le premier caractère est un # (dièse) alors le lien pointe vers une ancre définie dans la même page.

Il est possible d'insérer dans le corps du tag ouputLink d'autres composants qui feront partie intégrante du lien.

Exemple :

```
1. <p>
2.   <h:outputLink value="http://java.sun.com">
3.     <h:graphicImage value="/images/java.jpg"/>
4.   </h:outputLink>
5. </p>
```

Résultat

Le code HTML généré dans la page est le suivant :

Attention, pour mettre du texte dans le corps du tag, il est nécessaire d'utiliser un tag verbatim ou outputText.

Exemple :

```
1. <p>
2.   <h:outputLink value="http://java.sun.com" title="Java">
3.     <f:verbatim>
4.        Site Java de Sun
5.     </f:verbatim>
6.   </h:outputLink>
7. </p>
```

69.9.9. Les tags <selectBooleanCheckbox> et <selectManyCheckbox>

Ces composants représentent respectivement une case à cocher et un ensemble de cases à cocher.

Les attributs sont les suivants :

Attributs	Rôle
disabledClass	classe CSS pour les éléments non sélectionnés (pour le tag selectManyCheckbox uniquement)
enabledClass	classe CSS pour les éléments sélectionnés (pour le tag selectManyCheckbox uniquement)
layout	préciser la disposition des éléments (pour le tag selectManyCheckbox uniquement)
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	attributs communs de base
accesskey, border, dir, disabled, lang, readonly, style, tabindex, title	attributs communs liés à HTML (border pour le tag selectManyCheckbox uniquement)
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs liés aux événements JavaScript

L'attribut layout permet de préciser la disposition des cases à cocher : lineDirection pour une disposition horizontale (c'est la valeur par défaut) et pageDirection pour une disposition verticale.

Le tag <selectBooleanCheckbox> dont la valeur peut être associée à une propriété booléenne d'un bean représente une case à cocher simple.

Exemple :

```
1. <h:selectBooleanCheckbox value="#{saisieOptions.recevoirLettre}">
2. </h:selectBooleanCheckbox> Recevoir la lettre d'information
```

Résultat :

☐ Recevoir la lettre d'information

Pour gérer l'état du composant, il faut utiliser l'attribut value en lui fournissant la valeur d'une propriété booléen d'un backing bean.

Exemple :

```
01. public class SaisieOptions {
02.
03.   private boolean recevoirLettre;
04.
05.   public void setRecevoirLettre(boolean valeur) {
06.     recevoirLettre = valeur;
07.   }
08.
09.   public boolean getRecevoirLettre() {
10.     return recevoirLettre;
11.   }
12. ...
```

Le tag <selectManyCheckbox> représente un ensemble de cases à cocher. Dans cet ensemble, il est possible d'en sélectionner une ou plusieurs.

Chaque case à cocher est définie par un tag selectItem dans le corps du tag selectManyCheckbox.

Exemple :

```
1. <h:selectManyCheckbox layout="pageDirection">
2.   <f:selectItem itemValue="petit" itemLabel="Petit" />
3.   <f:selectItem itemValue="moyen" itemLabel="Moyen" />
4.   <f:selectItem itemValue="grand" itemLabel="Grand" />
5.   <f:selectItem itemValue="tresgrand" itemLabel="Tres grand" />
6. </h:selectManyCheckbox>
```

Résultat :

- ☐ Petit
- ☐ Moyen
- ☐ Grand
- ☐ Tres grand

Le rendu du composant est un tableau HTML dont chaque cellule contient une case à cocher encapsulée dans un tag HTML <label> :

Exemple :

```
01. <table>
02.  <tr>
03.   <td>
04.    <label><input name="_id0:_id1" value="petit" type="checkbox"> Petit</input></label></td>
05.  </tr>
06.  <tr>
07.   <td>
08.    <label><input name="_id0:_id1" value="moyen" type="checkbox"> Moyen</input></label></td>
09.  </tr>
10.  <tr>
11.   <td>
12.    <label><input name="_id0:_id1" value="grand" type="checkbox"> Grand</input></label></td>
13.  </tr>
14.  <tr>
15.   <td>
16.     <label><input name="_id0:_id1" value="tresgrand" type="checkbox"> Tres grand</input>
17.     </label></td>
18.  </tr>
19. </table>
```

69.9.10. Le tag <selectOneRadio>

Ce composant représente un ensemble de boutons radio dont un seul peut être sélectionné.

Les attributs sont les suivants :

Attributs	Rôle
disabledClass	classe CSS pour les éléments non sélectionnés
enabledClass	classe CSS pour les éléments sélectionnés
layout	préciser la disposition des éléments
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	Attributs communs de base
accesskey, border, dir, disabled, lang, readonly, style, tabindex, title	Attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	Attributs communs liés aux événements JavaScript

Les éléments peuvent être précisés un par un avec le tag <selectItem>.

Exemple :

```
1. <h:selectOneRadio layout="pageDirection">
2.   <f:selectItem itemValue="petit" itemLabel="Petit" />
3.   <f:selectItem itemValue="moyen" itemLabel="Moyen" />
4.   <f:selectItem itemValue="grand" itemLabel="Grand" />
5.   <f:selectItem itemValue="tresgrand" itemLabel="Tres grand" />
6. </h:selectOneRadio>
```

Résultat :

- ○ Petit
- ○ Moyen
- ○ Grand
- ○ Tres grand

Le rendu du composant est un tableau HTML dont chaque cellule contient un bouton radio encapsulé dans un tag HTML <label> :

Exemple :

```
01. <table>
02.   <tr>
03.     <td>
04.       <label><input type="radio" name="_id0:_id1" value="petit"> Petit</input></label></td>
05.   </tr>
06.   <tr>
07.     <td>
08.       <label><input type="radio" name="_id0:_id1" value="moyen"> Moyen</input></label></td>
09.   </tr>
10.   <tr>
11.     <td>
12.       <label><input type="radio" name="_id0:_id1" value="grand"> Grand</input></label></td>
13.   </tr>
14.   <tr>
15.     <td>
16.       <label><input type="radio" name="_id0:_id1" value="tresgrand"> Tres grand</input>
17.       </label></td>
18.   </tr>
19. </table>
```

Les éléments peuvent être précisés sous la forme d'un tableau de type SelecItem avec le tag <selectItems>.

Exemple :

```
1. <h:selectOneRadio value="#{saisieOptions.taille}" layout="pageDirection" id="taille">
2. <f:selectItems value="#{saisieOptions.tailleItems}"/>
3. </h:selectOneRadio>
```

Dans ce cas, le bean doit contenir au moins deux méthodes : getTaille() pour renvoyer la valeur de l'élément sélectionner et getTailleItems() qui renvoie un tableau d'objets de type SelectItems contenant les éléments.

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. import javax.faces.model.*;
04.
05. public class SaisieOptions {
06.
07.   private Integer taille = null;
08.
09.   private SelectItem[] tailleItems = {
10.     new SelectItem(new Integer(1), "Petit"),
11.     new SelectItem(new Integer(2), "Moyen"),
12.     new SelectItem(new Integer(3), "Grand"),
13.     new SelectItem(new Integer(4), "Très grand") };
14.
15.   public SaisieOptions() {
16.     taille = new Integer(2);
17.   }
18.
19.   public Integer getTaille() {
20.     return taille;
21.   }
22.
23.   public void setTaille(Integer newValue) {
24.     taille = newValue;
25.   }
26.
27.   public SelectItem[] getTailleItems() {
28.     return tailleItems;
29.   }
30. }
```

Le bean doit être déclaré dans le fichier faces-config.xml

Exemple :

```
1. <managed-bean>
2.   <managed-bean-name>saisieOptions</managed-bean-name>
3.   <managed-bean-class>com.jmd.test.jsf.SaisieOptions</managed-bean-class>
4.   <managed-bean-scope>session</managed-bean-scope>
5. </managed-bean>
```

Résultat :

○ Petit
◉ Moyen
○ Grand
○ Très grand

69.9.11. Le tag <selectOneListbox>

Ce composant représente une liste d'éléments dont un seul peut être sélectionné

Les attributs sont les suivants :

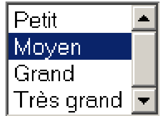
Attributs	Rôle
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	attributs communs de base
accesskey, dir, disabled, lang, readonly, style, size, tabindex, title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs liés aux événements JavaScript

L'attribut size permet de préciser le nombre d'éléments de la liste affichée.

Exemple :

```
1. <h:selectOneListbox value="#{saisieOptions.taille}">
2.   <f:selectItems value="#{saisieOptions.tailleItems}"/>
3. </h:selectOneListbox>
```

Résultat :



69.9.12. Le tag <selectManyListbox>

Ce composant représente une liste d'éléments dont plusieurs peuvent être sélectionnés.

Les attributs sont les suivants :

Attributs	Rôle
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	attributs communs de base
accesskey, dir, disabled, lang, readonly, style, size, tabindex, title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs liés aux événements JavaScript

Exemple :

```
1. <h:selectManyListbox value="#{saisieOptions.legumes}">
2.   <f:selectItems value="#{saisieOptions.legumesItems}"/>
3. </h:selectManyListbox>
```

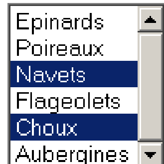
La liste des éléments sélectionnés doit pouvoir contenir zéro ou plusieurs valeurs sous la forme d'un tableau ou d'une liste.

Exemple :

```
01. package com.jmd.test.jsf;
02. 
03. import javax.faces.model.*;
04. 
```

```
05. public class SaisieOptions {
06.
07.   private String[] legumes = {
08.     "navets", "choux" };
09.   private SelectItem[] legumesItems = {
10.     new SelectItem("epinards", "Epinards"),
11.     new SelectItem("poireaux", "Poireaux"),
12.     new SelectItem("navets", "Navets"),
13.     new SelectItem("flageolets", "Flageolets"),
14.     new SelectItem("choux", "Choux"),
15.     new SelectItem("aubergines", "Aubergines") };
16.
17.   public SaisieOptions() {
18.   }
19.
20.   public String[] getLegumes() {
21.     return legumes;
22.   }
23.
24.   public SelectItem[] getLegumesItems() {
25.     return legumesItems;
26.   }
27. }
```

Résultat :



Il est possible d'utiliser un objet de type List à la place des tableaux.

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. import javax.faces.model.*;
04. import java.util.*;
05.
06. public class SaisieOptions {
07.
08.   private List legumes = null;
09.
10.   private List legumesItems = null;
11.
12.   public List getLegumesItems() {
13.     if (legumesItems == null) {
14.       legumesItems = new ArrayList();
15.       legumesItems.add(new SelectItem("epinards", "Epinards"));
16.       legumesItems.add(new SelectItem("poireaux", "Poireaux"));
17.       legumesItems.add(new SelectItem("navets", "Navets"));
18.       legumesItems.add(new SelectItem("flageolets", "Flageolets"));
19.       legumesItems.add(new SelectItem("choux", "Choux"));
20.       legumesItems.add(new SelectItem("aubergines", "Aubergines"));
21.     }
22.     return legumesItems;
23.   }
24.
25.   public List getLegumes() {
26.     return legumes;
27.   }
28.
29.   public void setLegumes(List newValue) {
30.     legumes = newValue;
31.   }
32.
33.   public SaisieOptions() {
34.     legumes = new ArrayList();
35.     legumes.add("navets");
36.     legumes.add("choux");
37.   }
38. }
```

69.9.13. Le tag <selectOneMenu>

Ce composant représente une liste déroulante dont un seul élément peut être sélectionné.

Les attributs sont les suivants :

Attributs	Rôle
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	attributs communs de base
accesskey, dir, disabled, lang, readonly, style, tabindex, title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs liés aux événements JavaScript

Exemple :

```
1. <h:selectOneMenu value="#{saisieOptions.taille}">
2. <f:selectItems value="#{saisieOptions.tailleItems}"/>
3. </h:selectOneMenu>
```

Résultat :

Epinards

Exemple : le code HTML généré

```
1. <select name="_id0:_id6" size="1"> <option value="1">Petit</option>
2.   <option value="2" selected="selected">Moyen</option>
3.   <option value="3">Grand</option>
4.   <option value="4">Tr&egrave;s grand</option>
5. </select>
```

69.9.14. Le tag <selectManyMenu>

Ce composant représente une liste d'éléments dont le rendu HTML est un tag select avec une seule option visible.

Les attributs sont les suivants :

Attributs	Rôle
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	Attributs communs de base
accesskey, dir, disabled, lang, readonly, style, tabindex, title	Attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	Attributs communs liés aux événements JavaScript

Exemple :

```
1. <h:selectManyMenu value="#{saisieOptions.legumes}">
2. <f:selectItems value="#{saisieOptions.legumesItems}"/>
3. </h:selectManyMenu>
```

Résultat :

69.9.15. Les tags <message> et <messages>

Des messages peuvent être émis lors de traitements. Ils sont stockés dans le contexte de l'application JSF pour être restitués dans la vue. Ils permettent notamment de fournir des messages d'erreur aux utilisateurs.

JSF définit quatre types de messages :

- Information
- Warning

- Error
- Fatal

Chaque message possède un résumé et un descriptif.

Le tag <messages> permet d'afficher tous les messages stockés dans le contexte de l'application JSF.

Le tag message permet d'afficher un seul message, le dernier ajouté, pour un composant donné.

Les attributs sont les suivants :

Attributs	Rôle
errorClass	nom d'une classe CSS pour un message de type error
errorStyle	style CSS pour un message de type error
fatalClass	nom d'une classe CSS pour un message de type fatal
fatalStyle	style CSS pour un message de type fatal
globalOnly	booléen qui permet de n'afficher que les messages qui ne sont pas associés à un composant. Par défaut, False (uniquement pour le tag messages)
infoClass	nom d'une classe CSS pour un message de type Information
infoStyle	style CSS pour un message de type Information
Layout	format de la liste de messages : list ou table (uniquement pour le composant messages)
showDetail	booléen qui précise si la description des messages est affichée ou non. Par défaut, false pour le tag message et true pour le tag messages
showSummary	booléen qui précise si le résumé des messages est affichée ou non. Par défaut, true pour le tag message et false pour le tag messages
Tooltip	booléen qui précise si la description est affiché sous la forme d'une bulle d'aide
warnClass	nom d'une classe CSS pour un message de type Warning
warnStyle	Style CSS pour un message de type Warning
For	l'identifiant du composant pour lequel le message doit être affiché
binding, id, rendered, styleClass	attributs communs de base
style, title	attributs communs liés à HTML

69.9.16. Le tag <panelGroup>

Ce composant permet de regrouper plusieurs composants.

Les attributs sont les suivants :

Attributs	Rôle
binding, id, rendered, styleClass	attributs communs de base
Style	attributs communs liés à HTML

Exemple :

```
1. <td bgcolor='#DDDDDD'>
2.   <h:panelGroup>
3.     <h:inputText value="#{login.nom}" id="nom" required="true"/>
4.     <h:message for="nom"/>
5.   </h:panelGroup>
6. </td>
```

Résultat :

Erreur de validation: Valeur requise.

69.9.17. Le tag <panelGrid>

Ce composant représente un tableau HTML.

Les attributs sont les suivants :

Attributs	Rôle
Bgcolor	couleur de fond du tableau
Border	taille de la bordure du tableau
Cellpadding	espacement intérieur de chaque cellule
Cellspacing	espacement extérieur de chaque cellule
columnClasses	nom de classes CSS pour les colonnes. Il est possible de préciser plusieurs noms de classes qui seront utilisées sur chaque colonne
Columns	nombre de colonnes du tableau
footerClass	nom de la classe CSS pour le pied du tableau
Frame	précise les règles pour le contour du tableau. Les valeurs possibles sont : none, above, below, hsides, vsides, lhs, rhs, box, border
headerClass	nom de la classe CSS pour l'en-tête du tableau
rowClasses	nom de classes CSS pour les lignes. Il est possible de préciser deux noms de classes séparés par une virgule qui seront utilisés alternativement sur chaque ligne
Rules	précise les règles de dessin des lignes entre les cellules. Les valeurs possibles sont : groups, rows, columns, all
Summary	résumé du tableau
binding, id, rendered, styleClass, value	attributs communs de base
dir, lang, style, title, width	attributs communs liés à HTML
onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	attributs communs liées aux événements JavaScript

Par défaut les composants sont insérés les uns à la suite des autres dans les cellules en partant de la gauche vers la droite et en passant à ligne suivante si nécessaire.

Il est possible de ne mettre qu'un seul composant par cellule. Ainsi pour placer plusieurs composants dans une cellule, il faut les regrouper dans un tag panelGroup.

Exemple :

```
01. <h:panelGrid columns="2">
02.   <h:outputText value="Nom : " />
03.   <h:panelGroup>
04.     <h:inputText value="#{login.nom}" id="nom" required="true"/>
05.     <h:message for="nom"/>
06.   </h:panelGroup>
07.   <h:outputText value="Mot de passe :" />
08.   <h:inputSecret value="#{login.mdp}"/>
09.   <h:commandButton value="Login" action="login"/>
10. </h:panelGrid>
```

Résultat :

Nom :
Mot de passe :
Login

Le code HTML généré est le suivant :

Exemple : le code HTML généré

```
01. ...
02. <table>
03.
04.   <tbody>
05.     <tr>
06.       <td>Nom : </td>
07.       <td><input id="_id0:nom" type="text" name="_id0:nom" /></td>
08.     </tr>
09.     <tr>
10.       <td>Mot de passe :</td>
11.       <td><input type="password" name="_id0:_id6" value="" /></td>
```

```
12.     </tr>
13.     <tr>
14.       <td><input type="submit" name="_id0:_id7" value="Login" /></td>
15.     </tr>
16.   </tbody>
17. </table>
18. ...
```

69.9.18. Le tag <dataTable>

Ce composant représente un tableau HTML dans lequel des données vont pouvoir être automatiquement présentées. Ce composant est sûrement le plus riche en fonctionnalité et donc le plus complexe des composants fournis en standard.

Les attributs sont les suivants :

Attributs	Rôle
Bgcolor	couleur de fond du tableau
Border	taille de la bordure du tableau
Cellpadding	espacement intérieur de chaque cellule
Cellspacing	espacement extérieur de chaque cellule
columnClasses	nom de classes CSS pour les colonnes. Il est possible de préciser plusieurs noms de classes qui seront utilisées sur chaque colonne
First	index de la première occurrence des données qui sera affichée dans le tableau
footerClass	nom de la classe CSS pour le pied du tableau
Frame	précise les règles pour le contour du tableau. Les valeurs possibles sont : none, above, below, hsides, vsides, lhs, rhs, box, border
headerClass	nom de la classe CSS pour l'en-tête du tableau
rowClasses	nom de classes CSS pour les lignes. Il est possible de préciser deux noms de classes qui seront utilisées alternativement sur chaque ligne
Rules	précise les règles de dessin des lignes entre les cellules. Les valeurs possibles sont : groups, rows, columns, all
Summary	résumé du tableau
Var	nom de la variable qui va contenir l'occurrence en cours de traitement lors du parcours des données
binding, id, rendered, styleClass, value	attributs communs de base
dir, lang, style, title, width	attributs communs liés à HTML
onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	attributs communs liés aux événements JavaScript

Le tag <dataTable> parcourt les données et pour chaque occurrence, il crée une ligne dans le tableau.

L'attribut value représente une expression qui précise les données à utiliser. Ces données peuvent être sous la forme :

- d'un tableau
- d'un objet de type java.util.List
- d'un objet de type java.sql.ResultSet
- d'un objet de type javax.servlet.jsp.jstl.sql.Result
- d'un objet de type javax.faces.model.DataModel

Pour chaque élément encapsulé dans les données, le tag dataTable crée une nouvelle ligne.

Quelque soit le type qui encapsule les données, le composant dataTable va les mapper dans un objet de type DataModel. C'est cet objet que le composant va utiliser comme source de données. JSF définit 5 classes qui héritent de la classe DataModel : ArrayDataModel, ListDataModel, ResultDataModel, ResultSetDataModel et ScalarDataModel.

La méthode getWrappedObject() permet d'obtenir la source de données fournie en paramètre de l'attribut value.

L'attribut item permet de préciser le nom d'une variable qui va contenir les données d'une occurrence.

Chaque colonne est définie grâce à un tag <column>.

Exemple :

```
01. <h:dataTable value="#{listePersonnes.personneItems}" var="personne" cellspacing="4">
02.   <h:column>
```

```
03.     <f:facet name="header">
04.       <h:outputText value="Nom" />
05.     </f:facet>
06.     <h:outputText value="#{personne.nom}"/>
07.   </h:column>
08.
09.   <h:column>
10.     <f:facet name="header">
11.       <h:outputText value="Prenom" />
12.     </f:facet>
13.     <h:outputText value="#{personne.prenom}" />
14.   </h:column>
15.
16.   <h:column>
17.     <f:facet name="header">
18.       <h:outputText value="Date de naissance" />
19.     </f:facet>
20.     <h:outputText value="#{personne.datenaiss}"/>
21.   </h:column>
22.
23.   <h:column>
24.     <f:facet name="header">
25.       <h:outputText value="Poids" />
26.     </f:facet>
27.     <h:outputText value="#{personne.poids}"/>
28.   </h:column>
29.
30.   <h:column>
31.     <f:facet name="header">
32.       <h:outputText value="Taille" />
33.     </f:facet>
34.     <h:outputText value="#{personne.taille}"/>
35.   </h:column>
36.
37. </h:dataTable>
```

L'en-tête et le pied du tableau sont précisés avec un tag <facet> pour chacun dans chaque tag <column>

Dans l'exemple précédent l'instance listePersonnes est une classe dont le code est le suivant :

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. import java.util.ArrayList;
04. import java.util.Calendar;
05. import java.util.GregorianCalendar;
06. import java.util.List;
07.
08. public class PersonnesBean {
09.
10.   private List PersonneItems = null;
11.
12.   public List getPersonneItems() {
13.     if (PersonneItems == null) {
14.       PersonneItems = new ArrayList();
15.       PersonneItems.add(new Personne("Nom1", "Prenom1",
16.         new GregorianCalendar(1967, Calendar.OCTOBER, 22).getTime(),10,1.10f));
17.       PersonneItems.add(new Personne("Nom2", "Prenom2",
18.         new GregorianCalendar(1972, Calendar.MARCH, 10).getTime(),20,1.20f));
19.       PersonneItems.add(new Personne("Nom3", "Prenom3",
20.         new GregorianCalendar(1944, Calendar.NOVEMBER, 4).getTime(),30,1.30f));
21.       PersonneItems.add(new Personne("Nom4", "Prenom4",
22.         new GregorianCalendar(1958, Calendar.JULY, 19).getTime(),40,1.40f));
23.       PersonneItems.add(new Personne("Nom5", "Prenom5",
24.         new GregorianCalendar(1934, Calendar.JANUARY, 6).getTime(),50,1.50f));
25.       PersonneItems.add(new Personne("Nom6", "Prenom6",
26.         new GregorianCalendar(1989, Calendar.DECEMBER, 12).getTime(),60,1.60f));
27.     }
28.     return PersonneItems;
29.   }
30. }
```

La méthode getPersonneItems() renvoie une collection d'objets de type Personne.

La classe Personne encapsule simplement les données d'une personne.

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. import java.util.Date;
04.
05. public class Personne {
06.   private String nom;
07.   private String prenom;
08.   private Date datenaiss;
09.   private int poids;
```

```
10.   private float taille;
11.   private boolean supprime;
12.
13.   public Personne(String nom, String prenom, Date datenaiss, int poids,
14.     float taille) {
15.     super();
16.     this.nom = nom;
17.     this.prenom = prenom;
18.     this.datenaiss = datenaiss;
19.     this.poids = poids;
20.     this.taille = taille;
21.     this.supprime = false;
22.   }
23.
24.   public boolean isSupprime() {
25.     return supprime;
26.   }
27.
28.   public void setSupprime(boolean supprimer) {
29.     supprime = supprimer;
30.   }
31.
32.   public Date getDatenaiss() {
33.     return datenaiss;
34.   }
35.
36.   public void setDatenaiss(Date datenaiss) {
37.     this.datenaiss = datenaiss;
38.   }
39.
40.   public String getNom() {
41.     return nom;
42.   }
43.
44.   public void setNom(String nom) {
45.     this.nom = nom;
46.   }
47.
48.   public int getPoids() {
49.     return poids;
50.   }
51.
52.   public void setPoids(int poids) {
53.     this.poids = poids;
54.   }
55.
56.   public String getPrenom() {
57.     return prenom;
58.   }
59.
60.   public void setPrenom(String prenom) {
61.     this.prenom = prenom;
62.   }
63.
64.   public float getTaille() {
65.     return taille;
66.   }
67.
68.   public void setTaille(float taille) {
69.     this.taille = taille;
70.   }
71. }
```

Il est très facile de préciser un style particulier pour des lignes paires et impaires.

Il suffit de définir les deux styles désirés.

Exemple dans la partie en-tête de la JSP :

```
01. <STYLE type="text/css">
02. <!--
03. .titre {
04. background-color:#000000;
05. color:#FFFFFF;
06. }
07. .paire {
08. background-color:#EFEFEF;
09. }
10. .impaire {
11. background-color:#CECECE;
12. }
13. -->
14. </STYLE>
```

Il suffit d'utiliser les attributs headerClass, footerClass, rowClasses ou columnClasses. Avec ces deux derniers attributs, il est possible de préciser plusieurs styles séparés par une virgule pour définir l'apparence de chacune des lignes de façon répétitive.

Exemple :

```
1. <h:dataTable value="#{listePersonnes.personneItems}" var="personne"
2.   cellspacing="4" width="60%" rowClasses="paire,impaire" headerClass="titre">
```

Résultat :

Nom	Prenom	Date de naissance	Poids	Taille
Nom1	Prenom1	22/10/1967	10	1.1
Nom2	Prenom2	10/03/1972	20	1.2
Nom3	Prenom3	04/11/1944	30	1.3
Nom4	Prenom4	19/07/1958	40	1.4
Nom5	Prenom5	06/01/1934	50	1.5
Nom6	Prenom6	12/12/1989	60	1.6

Les éléments du tableau peuvent par exemple être sélectionnés grâce à une case à cocher pour permettre de réaliser des traitements sur les éléments marqués.

Il suffit de rajouter dans l'exemple précédent une colonne contenant une case à cocher et, sous le tableau, un bouton qui va réaliser les traitements sur les éléments cochés.

Exemple dans la JSP :

```
01. ...
02. <h:form>
03.   <h1>Test</H1>
04.   <div align="center">
05.     <h:dataTable value="#{listePersonnes.personneItems}" var="personne"
06.       cellspacing="4" width="60%" rowClasses="paire,impaire" headerClass="titre">
07. ...
08.     <h:column>
09.       <f:facet name="header">
10.         <h:outputText value="Sélection"/>
11.       </f:facet>
12.       <h:selectBooleanCheckbox value="#{personne.supprime}" />
13.     </h:column>
14.
15.     </h:dataTable>
16.
17.     <p>
18.       <h:commandButton value="Supprimer les sélectionnés"
19.         action="#{listePersonnes.supprimer}"/>
20.     </p>
21.
22.   </div>
23. </h:form>
24. ...
```

Il reste alors à ajouter les traitements dans la méthode supprimer() de la classe PersonnesBean qui sera appelée lors d'un clic sur le bouton « Supprimer les sélectionnés ».

Exemple :

```
01. public class PersonnesBean {
02.
03. ...
04.
05.   public String supprimer() {
06.     Iterator iterator = personneItems.iterator();
07.     Personne pers=null;
08.     while (iterator.hasNext()) {
09.       pers = (Personne) iterator.next();
10.       System.out.println("nom="+pers.getNom()+" "+pers.isSupprime());
11.       // ajouter les traitements utiles
12.     }
13.     return null;
14.   }
15. }
```

Nom	Prenom	Date de naissance	Poids	Taille	Sélection
Nom1	Prenom1	22/10/1967	10	1.1	☐
Nom2	Prenom2	10/03/1972	20	1.2	☑
Nom3	Prenom3	04/11/1944	30	1.3	☐
Nom4	Prenom4	19/07/1958	40	1.4	☑
Nom5	Prenom5	06/01/1934	50	1.5	☐
Nom6	Prenom6	12/12/1989	60	1.6	☑

Supprimer les sélectionnés

Un clic sur le bouton « Supprimer les sélectionnés » affiche dans la console, la liste des éléments avec l'état de la case à cocher.

Exemple :

```
1. nom=Nom1 false
2. nom=Nom2 true
3. nom=Nom3 false
4. nom=Nom4 true
5. nom=Nom5 false
6. nom=Nom6 true
```

69.10. La gestion et le stockage des données

Les données sont stockées dans un ou plusieurs JavaBeans qui encapsulent les différentes données des composants.

Ces données possèdent deux représentations :

- une contenue en interne par le modèle
- une pour leur présentation dans l'interface graphique (pour la saisie ou l'affichage)

Chaque objet de type Renderer possède une représentation par défaut des données. La transformation d'une représentation en une autre est assurée par des objets de type Converter. JSF fournit en standard plusieurs objets de type Converter mais il est aussi possible de développer ses propres objets.

69.11. La conversion des données

JSF propose en standard un mécanisme de conversion des données. Celui-ci repose sur un ensemble de classes dont certaines sont fournies en standard pour des conversions de base. Il est possible de définir ses propres classes de conversion pour répondre à des besoins spécifiques.

Ces conversions sont nécessaires car toutes les données transmises et affichées le sont sous la forme de chaînes de caractères. Cependant, leur exploitation dans les traitements nécessite souvent qu'elles soient stockées dans un autre format pour être exploitées : un exemple flagrant est une donnée de type date.

Toutes les données saisies par l'utilisateur sont envoyées dans la requête http sous la forme de chaînes de caractères. Chacune de ces valeurs est désignée par « request value » dans les spécifications de JSF.

Ces valeurs sont stockées dans leurs composants respectifs dans des champs désignés par « submitted value » dans les spécifications.

Ces valeurs sont éventuellement converties implicitement ou explicitement et sont stockées dans leur composant respectif dans des champs désignés par « local value ». ensuite, ces données sont éventuellement validées si besoin.

L'intérêt d'un tel procédé est de s'assurer que les données seront valides avant de pouvoir les utiliser dans les traitements. Si la conversion ou la validation échoue, les traitements du cyle de vie de la page sont arrêtés et la page est réaffichéeen montrant les messages d'erreurs. Sinon la phase de mise à jour des données (« Update model values ») du modèle est exécutée.

Les spécifications JSF imposent l'implémentation des convertisseurs suivants : javax.faces.DateTime, javax.faces.Number, javax.faces.Boolean, javax.faces.Byte, javax.faces.Character, javax.faces.Double, javax.faces.Float, javax.faces.Integer, javax.faces.Long, javax.faces.Short, javax.faces.BigDecimal et javax.faces.BigInteger.

JSF effectue une conversion implicite des données lorsque celles-ci correspondent à un type primitif, à BigDecimal ou BigInteger en utilisant les convertisseurs appropriés.

Deux convertisseurs sont proposés en standard pour mettre en oeuvre des conversions qui ne correspondent pas à des types primitifs :

- le tag convertNumber : utilise le convertisseur javax.faces.Number
- le tag convertDateTime : utilise le convertisseur javax.faces.DateTime

69.11.1. Le tag <convertNumber>

Ce tag permet d'ajouter à un composant un convertisseur de valeur numérique.

Ce tag possède les attributs suivants :

Attributs	Rôle
type	type de valeur. Les valeurs possibles sont number (par défaut), currency et percent
pattern	motif de formatage qui sera utilisé par une instance de java.text.DecimalFormat
maxFractionDigits	nombre maximum de chiffres composant la partie décimale
minFractionDigits	nombre minimum de chiffres composant la partie décimale
maxIntegerDigits	nombre maximum de chiffres composant la partie entière
minIntegerDigits	nombre minimum de chiffres composant la partie entière
integerOnly	booléen qui précise si uniquement la partie entière est prise en compte (false par défaut)
groupingUsed	booléen qui précise si le séparateur de groupe d'unité est utilisé (true par défaut)
locale	objet de type java.util.Locale permettant de définir la Locale à utiliser pour les conversions
currencyCode	code de la monnaie utilisée pour la conversion
currencySymbol	symbole de la monnaie utilisée pour la conversion

Exemple :

```
01. <p>valeur1 = <h:outputText value="#{convert.prix}">
02. <f:convertNumber type="currency"/>
03. </h:outputText>
04. </p>
05.
06. <p>valeur2 = <h:outputText value="#{convert.poids}">
07. <f:convertNumber type="number"/>
08. </h:outputText>
09. </p>
10.
11. <p>valeur3 = <h:outputText value="#{convert.ratio}">
12. <f:convertNumber type="percent"/>
13. </h:outputText>
14. </p>
15.
16. <p>valeur4 = <h:outputText value="#{convert.prix}">
17. <f:convertNumber integerOnly="true" maxIntegerDigits="2"/>
18. </h:outputText>
19. </p>
20.
21. <p>valeur5 = <h:outputText value="#{convert.prix}">
22. <f:convertNumber pattern="#.##"/>
23. </h:outputText>
24. </p>
```

Le code du bean utilisé dans cet exemple est le suivant :

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. public class Convert {
04.   private int poids;
05.   private float prix;
06.   private float ratio;
07.
08.   public Convert() {
09.     super();
10.     this.poids = 12345;
11.     this.prix = 1234.56f ;
12.     this.ratio = 0.12f ;
13.   }
14.
15.
16.   public int getPoids() {
17.     return poids;
18.   }
19.
20.   public void setPoids(int poids) {
21.     this.poids = poids;
22.   }
23.
```

```
24.   public float getRatio() {
25.     return ratio;
26.   }
27.
28.   public void setRatio(float ratio) {
29.     this.ratio = ratio;
30.   }
31.
32.   public float getPrix() {
33.     return prix;
34.   }
35.
36.   public void setPrix(float prix) {
37.     this.prix = prix;
38.   }
39. }
```

valeur1 = 1 234,56 €

valeur2 = 12 345

valeur3 = 12%

valeur4 = 34,56

valeur5 = 1234,56

69.11.2. Le tag <convertDateTime>

Ce tag permet d'ajouter à un composant un convertisseur de valeurs temporelles.

Ce tag possède les attributs suivants :

Attributs	Rôle
Type	type de valeur. Les valeurs possibles sont date (par défaut), time et both
dateStyle	style prédéfini de la date. Les valeurs possibles sont short, medium, long, full ou default
timeStyle	style prédéfini de l'heure. Les valeurs possibles sont short, medium, long, full ou default
Pattern	motif de formatage qui sera utilisé par une instance de java.text.SimpleDateFormat
Locale	objet de type java.util.Locale permettant de définir la locale à utiliser pour les conversions
timeZone	objet de type java.util.TimeZone utilisé lors des conversions

Exemple :

```
01. <p>Date1 = <h:outputText value="#{convertDate.dateNaiss}">
02. <f:convertDateTime pattern="MM/yyyy"/>
03. </h:outputText>
04. </p>
05.
06. <p>Date2 = <h:outputText value="#{convertDate.dateNaiss}">
07. <f:convertDateTime pattern="EEE, dd MMM yyyy"/>
08. </h:outputText>
09. </p>
10.
11. <p>Date3 = <h:outputText value="#{convertDate.dateNaiss}">
12. <f:convertDateTime pattern="dd/MM/yyyy"/>
13. </h:outputText>
14. </p>
15.
16. <p>Date4 = <h:outputText value="#{convertDate.dateNaiss}">
17. <f:convertDateTime dateStyle="full"/>
18. </h:outputText>
19. </p>
```

Le code du bean utilisé comme source dans cet exemple est le suivant :

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. import java.util.Date;
```

```
04.
05. public class ConvertDate {
06.   private Date dateNaiss;
07.
08.   public ConvertDate() {
09.     super();
10.     this.dateNaiss = new Date();
11.   }
12.
13.   public Date getDateNaiss() {
14.     return dateNaiss;
15.   }
16.
17.   public void setDateNaiss(Date dateNaiss) {
18.     this.dateNaiss = dateNaiss;
19.   }
20. }
```

Résultat :

Date1 = 06/2005

Date2 = mer., 15 juin 2005

Date3 = 15/06/2005

Date4 = mercredi 15 juin 2005

69.11.3. L'affichage des erreurs de conversions

Les messages d'erreurs issus de ces conversions peuvent être affichés en utilisant les tag <message> ou <messages>.

Par défaut, ils contiennent une description : « Conversion error occured ».

Pour modifier ce message par défaut ou l'internationaliser, il faut définir une clé javax.faces.component.UIInput.CONVERSION dans le fichier properties de définition des chaînes de caractères.

Exemple :

```
1. javax.faces.component.UIInput.CONVERSION=La valeur saisie n'est pas correctement formatée.
```

69.11.4. L'écriture de convertisseurs personnalisés

JSF fournit en standard des convertisseurs pour les types primitifs et quelques objets de base. Il peut être nécessaire de développer son propre convertisseur pour des besoins spécifiques.

Pour écrire son propre convertisseur, il faut définir une classe qui implémente l'interface Converter. Cette interface définit deux méthodes :

- Object getAsObject(FacesContext context, UIComponent component, String newValue) : cette méthode permet de convertir une chaîne de caractères en objet
- String getAsString(FacesContext context, UIComponent component, Object value) : cette méthode permet de convertir un objet en chaîne de caractères

La méthode getAsObject() doit lever une exception de type ConverterException si une erreur de conversion est détectée dans les traitements.

La suite de ce chapitre sera développée dans une version future de ce document

69.12. La validation des données

JSF propose en standard un mécanisme de validation des données. Celui-ci repose sur un ensemble de classes qui permettent de faire des vérifications standards. Il est possible de définir ses propres classes de validation pour répondre à des besoins spécifiques.

La validation peut se faire de deux façons : au niveau de certains composants ou avec des classes spécialement développées pour des besoins spécifiques. Ces classes sont attachables à un composant et sont réutilisables. Ces validations sont effectuées côté serveur.

Les validators sont enregistrés sur des composants. Ce sont des classes qui utilisent des données pour effectuer des opérations de validation de la valeur des données : contrôle de présence, de type de données, de plage de valeurs, de format, ...

69.12.1. Les classes de validation standard

Toutes ces classes implémentent l'interface javax.faces.validator.Validator. JSF propose en standard plusieurs classes pour la validation :

- deux classes de validation sur une plage de données : LongRangeValidator et DoubleRangeValidator
- une classe de validation de la taille d'une chaîne de caractères : LengthValidator

Pour faciliter l'utilisation de ces classes, la bibliothèque de tags personnalisés Core propose des tags dédiés à la mise en oeuvre de ces classes :

- validateDoubleRange : utilise la classe DoubleRangeValidator
- validateLongRange : utilise la classe LongRangeValidator
- validateLength : utilise la classe LengthValidator

Ces trois tags possèdent deux attributs nommés minimum et maximum qui permettent de préciser respectivement la valeur de début et de fin selon le Validator utilisé. L'un, l'autre ou les deux attributs peuvent être utilisés.

L'ajout d'une validation sur un contrôle peut se faire de plusieurs manières :

- ajout d'une ou plusieurs validations directement dans la JSP
- ajout par programmation d'une validation en utilisant la méthode addValidator().
- certaines implémentations de composants peuvent contenir des validations implicites.

Pour ajouter une validation à un composant dans la JSP , il suffit d'insérer le tag de validation dans le corps du tag du composant.

Exemple :

```
1. <h:inputText id="nombre" converter="#{Integer}" required="true"
2.   value="#{saisieDonnees.nombre}">
3.   <f:validate_longrange minimum="1" maximum="9" />
4. </h:inputText>
```

Certaines implémentations de composants peuvent contenir des validations implicites en fonction du contexte. C'est par exemple le cas du composant <inputText> qui, lorsque son attribut required est à true, effectue un contrôle de présence de données saisies.

Exemple :

```
1. <h:inputText id="nombre" converter="#{Integer}" required="true"
2.   value="#{saisieDonnees.nombre}"/>
```

Toutes les validations sont faites côté serveur dans la version courante de JSF.

Les messages d'erreurs issus de ces conversions peuvent être affichés en utilisant les tags <message> ou <messages>.

Ils contiennent une description par défaut selon le validator utilisé commençant par « Validation error : ».

Pour modifier ce message par défaut ou l'internationaliser, il faut définir une clé dédiée dans le fichier properties de définition des chaînes de caractères. Les clés définies sont les suivantes :

- javax.faces.component.UIInput.REQUIRED
- javax.faces.validator.NOT_IN_RANGE
- javax.faces.validator.DoubleRangeValidator.MAXIMUM
- javax.faces.validator.DoubleRangeValidator.TYPE
- javax.faces.validator.DoubleRangeValidator.MINIMUM
- javax.faces.validator.LongRangeValidator.MAXIMUM
- javax.faces.validator.LongRangeValidator.MINIMUM
- javax.faces.validator.LongRangeValidator.TYPE
- javax.faces.validator.LengthValidator.MAXIMUM
- javax.faces.validator.LengthValidator.MINIMUM

69.12.2. Contourner la validation

Dans certains cas, il est nécessaire d'empêcher la validation. Par exemple, dans une page de saisie d'informations disposant d'un bouton « Valider » et « Annuler ». La validation doit être opérée lors d'un clic sur le bouton « Valider » mais ne doit pas l'être lors d'un clic sur le bouton « Annuler ».

Pour chaque composant dont l'action doit être exécutée sans validation, il faut mettre l'attribut immediate du composant à true.

Exemple :

```
1. <h:commandButton value="Annuler" action="annuler" immediate="true"/>
```

69.12.3. L'écriture de classes de validation personnalisées

JSF fournit en standard des classes de validation de base. Il peut être nécessaire de développer ses propres classes de validation pour des besoins spécifiques.

Pour écrire sa propre classe de validation, il faut définir une classe qui implémente l'interface javax.faces.validator.Validator. Cette interface définit une seule méthode :

- public void validate(FacesContext context, UIComponent component, Object toValidate) : cette méthode permet de réaliser les traitements de validation

Elle attend en paramètre :

- un objet de type FacesContext qui permet d'accéder au contexte de l'application jsf
- un objet de type UICOmponent qui contient une référence sur le composant dont la donnée est à valider
- un objet de type Object qui encapsule la valeur de la données à valider.

La méthode validate() doit lever une exception de type ValidatorException si une erreur dans les traitements de validation est détectée.

Exemple :

```
01. import java.util.regex.Matcher;
02. import java.util.regex.Pattern;
03.
04. import javax.faces.application.FacesMessage;
05. import javax.faces.component.UIComponent;
06. import javax.faces.component.UIInput;
07. import javax.faces.context.FacesContext;
08. import javax.faces.validator.Validator;
09. import javax.faces.validator.ValidatorException;
10.
11. import com.sun.faces.util.MessageFactory;
12.
13. public class NumeroDeSerieValidator implements Validator {
14.
15.   public static final String CLE_MESSAGE_VALIDATION_IMPOSSIBLE =
16.     "message.validation.impossible";
17.
18.   public void validate(FacesContext contexte, UIComponent composant,
19.     Object objet) throws ValidatorException {
20.     String valeur = null;
21.     boolean estValide = false;
22.
23.     if ((contexte == null) || (composant == null)) {
24.       throw new NullPointerException();
25.     }
26.     if (!(composant instanceof UIInput)) {
27.       return;
28.     }
29.
30.     valeur = objet.toString();
31.
32.     Pattern p = Pattern.compile("[0-9][0-9]-[0-9][0-9][0-9]",Pattern.MULTILINE);
33.     Matcher m = p.matcher(valeur);
34.     estValide = m.matches();
35.
36.     if (!estValide) {
37.       FacesMessage errMsg = MessageFactory.getMessage(contexte,
38.       CLE_MESSAGE_VALIDATION_IMPOSSIBLE);
39.       throw new ValidatorException(errMsg);
40.     }
41.
42.   }
43. }
```

Dans l'exemple précédent, la valeur à valider doit respecter une expression régulière de la forme deux chiffres, un tiret et trois chiffres.

Si la validation échoue alors il sera nécessaire d'informer l'utilisateur de la raison de l'échec grâce à un message stocké dans le resourceBundle de l'application.

Exemple :

```
1. message.validation.impossible=Le format du numéro de série est erroné
```

La valeur du message dans le resourceBundle peut être obtenue en utilisant la méthode getMessage() de la classe MessageFactory. Cette méthode attend en paramètres le contexte JSF de l'application et la clé du ressourceBundle à extraire. Elle renvoie un objet de type FacesMessages. Il suffit de fournir cet objet à la nouvelle instance de la classe ValidatorException.

Pour pouvoir utiliser une classe de validation, il faut la déclarer dans le fichier de configuration.

Exemple :

```
1. <validator>
2.   <validator-id>com.jmd.test.jsf.NumeroDeSerie</validator-id>
3.   <validator-class>com.jmd.test.jsf.NumeroDeSerieValidator</validator-class>
4. </validator>
```

Le tag <validator-id> permet de définir un identifiant pour la classe de validation. Le tag <validator-class> permet de préciser la classe pleinement qualifiée.

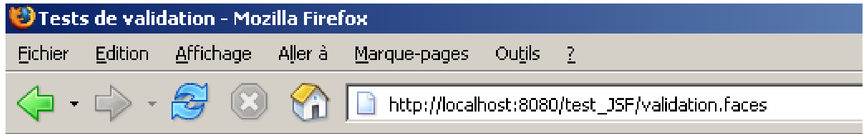
Pour utiliser la classe de validation dans une page, il faut utiliser le tag <validator> en fournissant à l'attribut validatorId la valeur donnée au tag <validator-id> dans le fichier de configuration :

Exemple :

```
01. <h:panelGrid columns="2">
02.   <h:outputText value="Numéro de série : " />
03.   <h:panelGroup>
04.     <h:inputText value="#{validation.numeroSerie}" id="numeroSerie" required="true">
05.       <f:validator validatorId="com.jmd.test.jsf.NumeroDeSerie"/>
06.     </h:inputText>
07.     <h:message for="numeroSerie"/>
08.   </h:panelGroup>
09. </h:panelGrid>
```

La saisie d'un numéro répondant à l'expression régulière et l'appui sur la touche entrée n'affiche aucun message d'erreur :

La saisie d'un numéro ne répondant pas à l'expression régulière affiche le message d'erreur :



69.12.4. La validation à l'aide de bean

Il est possible de définir une méthode dans un bean qui va offrir les services de validation. Cette méthode doit avoir une signature similaire à celle de la méthode validate() de l'interface Validator.

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. import java.util.regex.Matcher;
04. import java.util.regex.Pattern;
05.
06. import javax.faces.application.FacesMessage;
```

```
07. import javax.faces.component.UIComponent;
08. import javax.faces.component.UIInput;
09. import javax.faces.context.FacesContext;
10. import javax.faces.validator.ValidatorException;
11.
12. import com.sun.faces.util.MessageFactory;
13.
14. public class Validation {
15.   public static final String CLE_MESSAGE_VALIDATION_IMPOSSIBLE =
16.     "message.validation.impossible";
17.
18.   private String numeroSerie;
19.
20.   public String getNumeroSerie() {
21.     return numeroSerie;
22.   }
23.
24.   public void setNumeroSerie(String numeroSerie) {
25.     this.numeroSerie = numeroSerie;
26.   }
27.
28.   public void valider(FacesContext contexte, UIComponent composant, Object objet) {
29.     String valeur = null;
30.     boolean estValide = false;
31.
32.     if ((contexte == null) || (composant == null)) {
33.       throw new NullPointerException();
34.     }
35.     if (!(composant instanceof UIInput)) {
36.       return;
37.     }
38.
39.     valeur = objet.toString();
40.
41.     Pattern p = Pattern.compile("[0-9][0-9]-[0-9][0-9][0-9]",Pattern.MULTILINE);
42.     Matcher m = p.matcher(valeur);
43.     estValide = m.matches();
44.
45.     if (!estValide) {
46.       FacesMessage errMsg = MessageFactory.getMessage(contexte,
47.       CLE_MESSAGE_VALIDATION_IMPOSSIBLE);
48.       throw new ValidatorException(errMsg);
49.     }
50.   }
51. }
```

Pour utiliser cette méthode, il faut utiliser l'attribut validator et lui fournir en paramètre une expression qui désigne la méthode d'une instance du bean

Exemple :

```
1. <h:panelGrid columns="2">
2.   <h:outputText value="Numéro de série : " />
3.   <h:panelGroup>
4.     <h:inputText value="#{validation.numeroSerie}" id="numeroSerie"
5.       required="true" validator="#{validation.valider}" />
6.     <h:message for="numeroSerie"/>
7.   </h:panelGroup>
8. </h:panelGrid>
```

Cette approche est particulièrement utile pour des besoins spécifiques à une application car sa mise en oeuvre est difficilement portable d'une application à une autre.

69.12.5. La validation entre plusieurs composants

De base, le modèle de validation des données proposé par JSF repose sur une validation unitaire de chaque composant. Il est cependant fréquent d'avoir besoin de faire une validation en fonction des données d'un ou plusieurs autres composants.

Pour réaliser ce genre de tâche, il faut créer un backing bean qui aura accès à chacun des composants nécessaires aux traitements et définir dans ce bean une méthode qui va réaliser les traitements de validation.

Exemple :

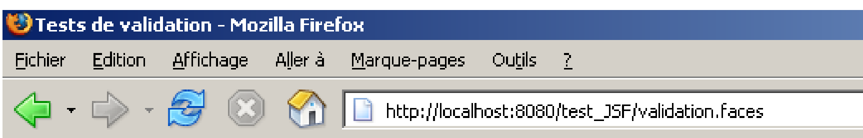
```
001. package com.jmd.test.jsf;
002.
003. import java.util.regex.Matcher;
004. import java.util.regex.Pattern;
005.
006. import javax.faces.application.FacesMessage;
007. import javax.faces.component.UIComponent;
008. import javax.faces.component.UIInput;
009. import javax.faces.context.FacesContext;
```

```
010. import javax.faces.validator.ValidatorException;
011.
012. import com.sun.faces.util.MessageFactory;
013.
014. public class Validation {
015.   public static final String CLE_MESSAGE_VALIDATION_IMPOSSIBLE =
016.     "message.validation.impossible";
017.
018.   private String numeroSerie;
019.   private String cle;
020.   private UIInput cleInput;
021.   private UIInput numeroSerieInput;
022.
023.   public String getNumeroSerie() {
024.     return numeroSerie;
025.   }
026.
027.   public void setNumeroSerie(String numeroSerie) {
028.     this.numeroSerie = numeroSerie;
029.   }
030.   public void valider(FacesContext contexte, UIComponent composant, Object objet) {
031.     String valeur = null;
032.     boolean estValide = false;
033.
034.     if ((contexte == null) || (composant == null)) {
035.       throw new NullPointerException();
036.     }
037.     if (!(composant instanceof UIInput)) {
038.       return;
039.     }
040.
041.     valeur = objet.toString();
042.
043.     Pattern p = Pattern.compile("[0-9][0-9]-[0-9][0-9][0-9]",Pattern.MULTILINE);
044.     Matcher m = p.matcher(valeur);
045.     estValide = m.matches();
046.
047.     if (!estValide) {
048.       FacesMessage errMsg = MessageFactory.getMessage(contexte,
049.       CLE_MESSAGE_VALIDATION_IMPOSSIBLE);
050.       throw new ValidatorException(errMsg);
051.     }
052.   }
053.
054.   public void validerCle(FacesContext contexte, UIComponent composant, Object objet) {
055.     System.out.println("validerCle");
056.
057.     String valeurNumero = numeroSerieInput.getLocalValue().toString();
058.     String valeurCle = cleInput.getLocalValue().toString();
059.     boolean estValide = false;
060.     if (contexte == null) {
061.       throw new NullPointerException();
062.     }
063.
064.     Pattern p = Pattern.compile("[0-9][0-9]-[0-9][0-9][0-9]",Pattern.MULTILINE);
065.     Matcher m = p.matcher(valeurNumero);
066.     estValide = m.matches() && valeurCle.equals("789");
067.
068.     System.out.println("estValide="+estValide);
069.     if (!estValide) {
070.       FacesMessage errMsg = MessageFactory.getMessage(contexte,
071.       CLE_MESSAGE_VALIDATION_IMPOSSIBLE);
072.       throw new ValidatorException(errMsg);
073.     }
074.   }
075.
076.   public String getCle() {
077.     return cle;
078.   }
079.
080.   public void setCle(String cle) {
081.     this.cle = cle;
082.   }
083.
084.   public UIInput getCleInput() {
085.     return cleInput;
086.   }
087.
088.   public void setCleInput(UIInput cleInput) {
089.     this.cleInput = cleInput;
090.   }
091.
092.   public UIInput getNumeroSerieInput() {
093.     return numeroSerieInput;
094.   }
095.
096.   public void setNumeroSerieInput(UIInput numeroSerieInput) {
097.     this.numeroSerieInput = numeroSerieInput;
098.   }
099. }
```

Il suffit alors d'ajouter un champ caché dans la vue sur lequel la classe de validation sera appliquée.

Exemple :

```
01. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
02. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
03. <%@ page language="java" %>
04. <!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
05. <html>
06. <f:view>
07. <head>
08.   <title>Tests de validation</title>
09. </head>
10. <body bgcolor="#FFFFFF">
11.   <h:form>
12.   <h2>Tests de validation</h2>
13.
14.   <h:panelGrid columns="2">
15.     <h:outputText value="Numéro de série : " />
16.     <h:panelGroup>
17.       <h:inputText value="#{validation.numeroSerie}" id="numeroSerie"
18.         required="true" binding="#{validation.numeroSerieInput}" />
19.       <h:message for="numeroSerie"/>
20.     </h:panelGroup>
21.     <h:outputText value="clé : " />
22.     <h:panelGroup>
23.       <h:inputText value="#{validation.cle}" id="cle" binding="#{validation.cleInput}"
24.         required="true" />
25.       <h:message for="validationCle"/>
26.     </h:panelGroup>
27.   </h:panelGrid>
28.
29.   <h:inputHidden id="validationCle" validator="#{validation.validerCle}" value="nul"/>
30.
31.   <h:commandButton value="Valider" action="submit"/>
32.
33.   </h:form>
34. </body>
35. </f:view>
36. </html>
```



Tests de validation

Numéro de série : 12-456

clé : 123 Le format du numéro de série est erroné

Valider

69.12.6. L'écriture de tags pour un convertisseur ou un validateur de données

L'écriture d'un tag personnalisé facilite l'utilisation d'un convertisseur ou d'un validateur et permet de lui fournir des paramètres.

Il faut définir une classe nommée handler qui va contenir les traitements du tag. Cette classe doit hériter d'une sous-classe dédiée selon le type d'élément que va représenter le tag :

- ConverterTag : si le tag concerne un convertisseur
- ValidatorTag : si le tag concerne un validateur
- UIComponentTag et UIComponentBodyTag : si le tag concerne un composant

Le handler est un bean dont les propriétés doivent correspondre à chaque attribut défini dans le tag.

Pour pouvoir utiliser un tag personnalisé, il faut définir un fichier .tld

Ce fichier au format XML défini dans les spécifications des JSP permet de fournir des informations sur la bibliothèque de tags personnalisés notamment la version des spécifications utilisées et des informations sur chaque tag.

Enfin, il est nécessaire de déclarer l'utilisation de la bibliothèque de tags personnalisés dans la JSP.

69.12.6.1. L'écriture d'un tag personnalisé pour un convertisseur

Il faut définir un handler pour le tag qui est un bean héritant de la classe ConverterTag.

Il est important dans le constructeur du handler de faire un appel à la méthode setConverterId() en lui passant un id défini dans le fichier de configuration de l'application JSF.

Il faut redéfinir la méthode release() dont les traitements vont permettre de réinitialiser les propriétés de la classe. Ceci est important lorsque, pour améliorer les performances, on souhaite placer ces objets dans un pool. La méthode release() est dans ce cas utilisée pour recycler les instances du pool non utilisées.

Il faut ensuite redéfinir la méthode createConverter() qui va permettre la création d'une instance du converter en utilisant les éventuels valeurs des attributs du tag.

La valeur fournie à un attribut d'un tag pour être soit un littéral soit une expression dont le contenu devra être évalué au moment de son utilisation.

La suite de ce chapitre sera développée dans une version future de ce document

69.12.6.2. L'écriture d'un tag personnalisé pour un validateur

L'écriture d'un tag personnalisé pour un validateur suit les mêmes règles que pour un convertisseur. La grande différence est que la classe handler doit hériter de la classe ValidatorTag. La méthode à appeler dans le constructeur est la méthode setValidatorId() et la méthode à redéfinir pour créer une instance du validateur est la méthode createValidator().

La suite de ce chapitre sera développée dans une version future de ce document

69.13. La sauvegarde et la restauration de l'état

JSF sauvegarde l'état de chaque élément présent dans la vue : les composants, les convertisseurs, les validateurs, ... pourvu que ceux-ci mettent en oeuvre un mécanisme adéquat.

Ces états sont stockés dans un champ de type hidden dans la vue pour permettre leur échange entre deux requêtes si l'application le prévoit dans le fichier de configuration.

Ce mécanisme peut prendre deux formes selon que :

- la classe qui encapsule l'élément implémente l'interface Serializable
- la classe qui encapsule l'élément implémente l'interface StateHolder

Dans le premier cas, c'est le mécanisme standard de la sérialisation qui sera utilisé. Il nécessite donc très peu voire aucun code particulier si les champs de la classe sont tous d'un type qui est sérialisable.

L'implémentation de l'interface StateHolder nécessite la définition des deux méthodes définies dans l'interface (saveState() et restoreState()) et la présence d'un constructeur par défaut. Cette approche peut être intéressante pour obtenir un contrôle très fin de la sauvegarde et de la restauration de l'état.

La méthode saveState(FacesContext) renvoie un objet sérialisable qui va contenir les données de l'état à sauvegarder. La méthode restoreState(FacesContext, Object) effectue l'opération inverse.

Il est aussi nécessaire de définir une propriété nommée transient de type booléen qui précise si l'état doit être sauvegardé ou non.

Si l'élément n'implémente pas l'interface Serializable ou StateHolder alors son état n'est pas sauvegardé entre deux échanges de la vue.

69.14. Le système de navigation

Une application de type web se compose d'un ensemble de pages dans lequel l'utilisateur navigue en fonction de ses actions.

Un système de navigation standard peut être facilement mis en oeuvre avec JSF grâce à un paramétrage au format XML dans le fichier de configuration

de l'application.

Le système de navigation assure la gestion de l'enchaînement des pages en utilisant des actions. Les règles de navigation sont des chaînes de caractères qui sont associées à une page d'origine et qui permettent de déterminer la page de résultat. Toutes ces règles sont contenues dans le fichier de configuration face-config.xml.

La déclaration de ce système de navigation ressemble à celle utilisée dans le framework Struts.

Le système de navigation peut être statique ou dynamique. Dans ce dernier cas, des traitements particuliers doivent être mis en place pour déterminer la cible de la navigation.

Exemple :

```
01. ...
02.   <navigation-rule>
03.     <from-view-id>/login.jsp</from-view-id>
04.     <navigation-case>
05.       <from-outcome>login</from-outcome>
06.       <to-view-id>/accueil.jsp</to-view-id>
07.     </navigation-case>
08.   </navigation-rule>
09. ...
```

La tag <navigation-rule> permet de préciser des règles de navigation.

La tag <from-view-id> permet de préciser la page concernée. Ce tag n'est pas obligatoire : sans sa présence, il est possible de définir une règle de navigation applicable à toutes les pages JSF de l'application.

Exemple :

```
1. <navigation-rule>
2.   <navigation-case>
3.     <from-outcome>logout</from-outcome>
4.     <to-view-id>/logout.jsp</to-view-id>
5.   </navigation-case>
6. </navigation-rule>
```

Il est aussi possible de désigner un ensemble de pages dans le tag <from-view-id> en utilisant le caractère * dans la valeur du tag. Ce caractère * ne peut être utilisé qu'une seule fois dans la valeur du tag et il doit être en dernière position.

Exemple :

```
1. <from-view-id>/admin/*</from-view-id>
```

Le tag <navigation-case> permet de définir les différents cas.

La valeur du tag <from-outcome> doit correspondre au nom d'une action.

Le tag <to-view-id> permet de préciser la page qui sera affichée. L'URL fournie comme valeur doit commencer par un slash et doit préciser une page possédant une extension brute (ne surtout pas mettre une URL utilisée par la servlet faisant office de contrôleur).

Le tag <redirect/> inséré juste après le tag <to-view-id> permet au navigateur de l'utilisateur d'effectuer la redirection vers la page indiquée.

La gestion de la navigation est assurée par une instance de la classe NavigationHandler, gérée au niveau de l'application. Ce gestionnaire utilise la valeur d'un attribut action d'un composant pour déterminer la page suivante et faire la redirection vers la page adéquate en fonction des informations fournies dans le fichier de configuration.

La valeur de l'attribut action peut être statique : dans ce cas la valeur est en dur dans le code de la vue

Exemple :

```
1. <h:commandButton action="login"/>
```

La valeur de l'attribut action peut être dynamique : dans ce cas la valeur est déterminée par l'appel d'une méthode d'un bean

Exemple :

```
1. <h:commandButton action="#{login.verifierMotDePasse}"/>
```

Dans ce cas, la méthode appelée ne doit pas avoir de paramètre et doit retourner une chaîne de caractères définie dans le fichier de configuration.

Lors des traitements par le NavigationHandler, si aucune action ne trouve de correspondance dans le fichier de configuration pour la page alors la page

est simplement réaffichée.

69.15. La gestion des événements

Le modèle de gestion des événements de JSF est similaire à celui utilisé dans les JavaBeans : il repose sur les Listener et les Event pour traiter les événements générés dans les composants graphiques suite aux actions de l'utilisateur.

Un objet de type Event encapsule le composant à l'origine de l'événement et des données relatives à cet événement.

Pour être notifié d'un événement particulier, il est nécessaire d'enregistrer un objet qui implémente l'interface Listener auprès du composant concerné.

Lors de certaines actions de l'utilisateur, un événement est émis.

L'implémentation JSF propose deux types d'événements :

- Value changed : ces événements sont émis lors du changement de la valeur d'un composant de type UIInput, UISelectOne, UISelectMany, et UISelectBoolean
- Action : ces événements sont émis lors d'un clic sur un hyperlien ou un bouton qui sont des composants de type UICommand

JSF propose de transposer le modèle de gestion des événements des interfaces graphiques des applications standalone aux applications de type web utilisant JSF.

La gestion des événements repose donc sur deux types d'objets

- Event : classe qui encapsule l'événement lui-même
- Listener : classe qui va encapsuler les traitements à réaliser pour un type d'événement

Comme pour les interfaces graphiques des applications standalone, la classe de type Listener doit s'enregistrer auprès du composant concerné. Lorsque celui-ci émet un événement suite à une action de l'utilisateur, il appelle le Listener enregistré en lui fournissant en paramètre un objet de type Event.

Exemple :

```
1. <h:selectOneMenu ... valueChangeListener="#{choixLangue.langueChangement}">
2.   ...
3. </h:selectOneMenu>
```

JSF supporte trois types d'événements :

- les changements de valeurs : concernent les composants qui permettent la saisie ou la sélection d'une valeur lorsque cette valeur change
- les actions : concernent un clic sur un bouton (commandButton) ou un lien (commandLink)
- les événements liés au cycle de vie : ils sont émis par le framework JSF durant le cyle de vie des traitements

Les traitements des listeners peuvent affecter la suite des traitements du cycle de vie de plusieurs manières :

- par défaut, laisser ces traitements se poursuivre
- demander l'exécution immédiate de la dernière étape en utilisant la méthode FacesContext.renderResponse()
- arrêter les traitements du cyle de vie en utilisant la méthode FacesContext.responseComplete()

69.15.1. Les événements liés à des changements de valeur

Il y a deux façons de préciser un listener de type valueChangeListener sur un composant :

- utiliser l'attribut valueChangeListener
- utiliser le tag valueChangeListener

L'attribut valueChangeListener permet de préciser, par une expression, la méthode exécutée durant les traitements du cyle de vie de la requête. Pour que ces traitements puissent être déclenchés, il faut soumettre la page.

Exemple :

```
1. <h:selectOneMenu value="#{choixLangue.langue}" onchange="submit()"
2.   valueChangeListener="#{choixLangue.langueChangement}">
3.   <f:selectItems value="#{choixLangue.langues}"/>
4. </h:selectOneMenu>
```

La méthode ne renvoie aucune valeur et attend en paramètre un objet de type ValueChangeEvent.

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. import java.util.Locale;
04. import javax.faces.context.FacesContext;
05. import javax.faces.event.ValueChangeEvent;
06. import javax.faces.model.SelectItem;
07.
08. public class ChoixLangue {
09.   private static final String LANGUE_FR = "Français";
10.   private static final String LANGUE_EN = "Anglais";
11.   private String langue = LANGUE_FR;
12.
13.   private SelectItem[] langueItems = {
14.     new SelectItem(LANGUE_FR, "Français"),
15.     new SelectItem(LANGUE_EN, "Anglais") };
16.
17.   public SelectItem[] getLangues() {
18.     return langueItems;
19.   }
20.
21.   public String getLangue() {
22.     return langue;
23.   }
24.
25.   public void setLangue(String langue) {
26.     this.langue = langue;
27.   }
28.
29.   public void langueChangement(ValueChangeEvent event) {
30.     FacesContext context = FacesContext.getCurrentInstance();
31.     System.out.println("Changement de la langue : "+event.getNewValue());
32.     if (LANGUE_FR.equals((String) event.getNewValue()))
33.       context.getViewRoot().setLocale(Locale.FRENCH);
34.     else
35.       context.getViewRoot().setLocale(Locale.ENGLISH);
36.     }
37.   }
38. }
```

La classe ValueChangeEvent possède plusieurs méthodes utiles :

Méthode	Rôle
UIComponent getComponent()	renvoie le composant qui a généré l'événement
Object getNewValue()	renvoie la nouvelle valeur (convertie et validée)
Object getOldValue()	renvoie la valeur précédente

Le tag valueChangeListener permet aussi de préciser un listener. Son attribut type permet de préciser une classe implémentant l'interface ValueChangeListener.

Exemple :

```
1. <h:selectOneMenu value="#{choixLangue.langue}" onchange="submit()">
2.   <f:valueChangeListener type="com.jmd.test.jsf.ChoixLangueListener"/>
3.   <f:selectItems value="#{choixLangue.langues}"/>
4. </h:selectOneMenu>
```

Une telle classe doit définir une méthode processValueChange() qui va contenir les traitements exécutés en réponse à l'événement.

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. import java.util.Locale;
04.
05. import javax.faces.context.FacesContext;
06. import javax.faces.event.AbortProcessingException;
07. import javax.faces.event.ValueChangeEvent;
08. import javax.faces.event.ValueChangeListener;
09.
10. public class ChoixLangueListener implements ValueChangeListener {
11.
12.   private static final String LANGUE_FR = "Français";
13.
14.   private static final String LANGUE_EN = "Anglais";
15.
16.   public void processValueChange(ValueChangeEvent event)
17.     throws AbortProcessingException {
18.     FacesContext context = FacesContext.getCurrentInstance();
19.     System.out.println("Changement de la langue : " + event.getNewValue());
20.     if (LANGUE_FR.equals((String) event.getNewValue()))
21.       context.getViewRoot().setLocale(Locale.FRENCH);
22.     else
23.       context.getViewRoot().setLocale(Locale.ENGLISH);
```

```
24.     }
25.   }
26. }
```

69.15.2. Les événements liés à des actions

Les actions sont des clics sur des boutons ou des liens. Le clic sur un composant de type commandLink ou commandButton déclenche automatiquement la soumission de la page.

Il y a deux façons de préciser un listener de type actionListener sur un composant :

- utiliser l'attribut actionListener
- utiliser le tag actionListener

L'attribut actionListener permet de préciser, par une expression, la méthode exécutée durant les traitements du cyle de vie de la requête.

Exemple :

```
01. <table align="center" width="50%">
02.   <tr>
03.     <td width="50%"><h:commandButton image="images/bouton_valider.gif"
04.       actionListener="#{saisieDonnees.traiterAction}"
05.       id="Valider" />
06.     </td>
07.     <td><h:commandButton image="images/bouton_annuler.gif"
08.       actionListener="#{saisieDonnees.traiterAction}"
09.       id="Annuler"/>
10.     </td>
11.   </tr>
12. </table>
```

Cette méthode attend en paramètre un objet de type ActionEvent.

Exemple :

```
01. package com.jmd.test.jsf;
02. 
03. import javax.faces.context.FacesContext;
04. import javax.faces.event.ActionEvent;
05. 
06. public class SaisieDonnees {
07. 
08.   public void traiterAction(ActionEvent e) {
09. 
10.     FacesContext context = FacesContext.getCurrentInstance();
11. 
12.     String clientId = e.getComponent().getClientId(context);
13.     System.out.println("traiterAction : clientId=" + clientId);
14. 
15.   }
16. }
```

Le tag valueChangeListener permet aussi de préciser un listener. Son attribut type permet de préciser une classe implémentant l'interface ValueChangeListener.

Exemple :

```
01. <table align="center" width="50%">
02.   <tr>
03.     <td width="50%"><h:commandButton image="images/bouton_valider.gif" id="Valider" >
04.         <f:actionListener type="com.jmd.test.jsf.SaisieDonneesListener"/>
05.       </h:commandButton>
06.       </td>
07.     <td><h:commandButton image="images/bouton_annuler.gif" id="Annuler">
08.         <f:actionListener type="com.jmd.test.jsf.SaisieDonneesListener"/>
09.       </h:commandButton>
10.     </td>
11.   </tr>
12. </table>
```

Une telle classe doit définir la méthode processAction() définie dans l'interface.

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. import javax.faces.context.FacesContext;
04. import javax.faces.event.AbortProcessingException;
05. import javax.faces.event.ActionEvent;
06. import javax.faces.event.ActionListener;
07.
08. public class SaisieDonneesListener implements ActionListener {
09.
10.   public void processAction(ActionEvent e) throws AbortProcessingException {
11.     FacesContext context = FacesContext.getCurrentInstance();
12.
13.     String clientId = e.getComponent().getClientId(context);
14.     System.out.println("processAction : clientId=" + clientId);
15.
16.   }
17. }
```

69.15.3. L'attribut immediate

L'attribut immediate permet de demander les traitements immédiats des listeners.

Par exemple, sur une page un composant possède un attribut required et un second possède un listener. Les traitements du second doivent pouvoir être réalisés sans que le premier composant n'affiche un message d'erreur lié à sa validation.

Le cycle de traitement de la requête est modifié lorsque l'attribut immediate est positionné dans un composant. Dans ce cas, les données du composant sont converties et validées si nécessaire puis les traitements du listener sont exécutés à la place de l'étape « Process validations » (juste après l'étape Apply Request Value).

Exemple :

```
1. <h:selectOneMenu value="#{choixLangue.langue}" onchange="submit()" immediate="true">
2.   <f:valueChangeListener type="com.jmd.test.jsf.ChoixLangueListener"/>
3.   <f:selectItems value="#{choixLangue.langues}"/>
4. </h:selectOneMenu>
```

Par défaut, ceci modifie l'ordre d'exécution des traitements du cycle de vie mais n'empêche pour les traitements prévus de s'exécuter. Pour les inhiber, il est nécessaire de demande au framework JSF d'interrompre les traitements du cycle de vie en utilisant la méthode renderResponse() du FaceCcontext.

Exemple :

```
01. public void langueChangement(ValueChangeEvent event) {
02.   FacesContext context = FacesContext.getCurrentInstance();
03.   System.out.println("Changement de la langue : " + event.getNewValue());
04.   if (LANGUE_FR.equals((String) event.getNewValue())) {
05.     context.getViewRoot().setLocale(Locale.FRENCH);
06.   } else {
07.     context.getViewRoot().setLocale(Locale.ENGLISH);
08.   }
09.   context.renderResponse();
10. }
```

Le mode de fonctionnement est le même avec les actionListener hormis le fait que l'appel à la méthode renderResponse() est inutile puisqu'il est automatiquement fait par le framework.

69.15.4. Les événements liés au cycle de vie

Le framework émet des événements avant et après chaque étape du cycle de vie des requêtes. Ils sont traités par des phaseListeners.

L'enregistrement d'un phaseListener se fait dans le fichier de configuration dans un tag fils <phase-listener> fils du tag <lifecycle> qui doit contenir le nom pleinement qualifié d'une classe.

Exemple :

```
1. <faces-config>
2. ...
3.
4.   <lifecycle>
5.     <phase-listener>com.jmd.test.jsf.PhasesEcouteur</phase-listener>
6.   </lifecycle>
```

```
7.
8. </faces-config>
```

La classe précisée doit implémenter l'interface javax.faces.event.PhaseListener qui définit trois méthodes :

- getPhaseId() : renvoie un objet de type PhaseId qui permet de préciser à quelle phase ce listener correspond
- beforePhase() : traitements à exécuter avant l'exécution de la phase
- afterPhase() : traitements à exécuter après l'exécution de la phase

La classe PhaseId définit des constantes permettant d'identifier chacune des phases : PhaseId.RESTORE_VIEW, PhaseId.APPLY_REQUEST_VALUES, PhaseId.PROCESS_VALIDATIONS, PhaseId.UPDATE_MODEL_VALUES, PhaseId.INVOKE_APPLICATION et PhaseId.RENDER_RESPONSE

Elle définit aussi la constante PhaseId.ANY_PHASE qui permet de demander l'application du listener à toutes les phases. Cela peut être très utile lors du débogage.

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. import javax.faces.event.PhaseEvent;
04. import javax.faces.event.PhaseId;
05. import javax.faces.event.PhaseListener;
06.
07. public class PhasesEcouteur implements PhaseListener {
08.
09.   public void afterPhase(PhaseEvent pe) {
10.     System.out.println("Apres " + pe.getPhaseId());
11.   }
12.
13.   public void beforePhase(PhaseEvent pe) {
14.     System.out.println("Avant " + pe.getPhaseId());
15.   }
16.
17.   public PhaseId getPhaseId() {
18.     return PhaseId.ANY_PHASE;
19.   }
20. }
```

Lors de l'appel de la première page de l'application, les informations suivantes sont affichées dans la sortie standard

Avant RESTORE_VIEW 1
Apres RESTORE_VIEW 1
Avant RENDER_RESPONSE 6
Apres RENDER_RESPONSE 6

Lors d'une soumission de cette page avec une erreur de validation des données, les informations suivantes sont affichées dans la sortie standard

Avant RESTORE_VIEW 1
Apres RESTORE_VIEW 1
Avant APPLY_REQUEST_VALUES 2
Apres APPLY_REQUEST_VALUES 2
Avant PROCESS_VALIDATIONS 3
Apres PROCESS_VALIDATIONS 3
Avant RENDER_RESPONSE 6
Apres RENDER_RESPONSE 6

Lors d'une soumission de cette page sans erreur de validation des données, les informations suivantes sont affichées dans la sortie standard

Avant RESTORE_VIEW 1
Apres RESTORE_VIEW 1
Avant APPLY_REQUEST_VALUES 2
Apres APPLY_REQUEST_VALUES 2
Avant PROCESS_VALIDATIONS 3
Apres PROCESS_VALIDATIONS 3
Avant UPDATE_MODEL_VALUES 4
Apres UPDATE_MODEL_VALUES 4
Avant INVOKE_APPLICATION 5
Apres INVOKE_APPLICATION 5
Avant RENDER_RESPONSE 6
Apres RENDER_RESPONSE 6

69.16. Le déploiement d'une application

Une application utilisant JSF s'exécute dans un serveur d'applications contenant un conteneur web implémentant les spécifications servlet 1.3 et JSP 1.2 minimum. Une telle application doit être packagée dans un fichier .war.

La compilation des différentes classes de l'application nécessite l'ajout dans le classpath de la bibliothèque servlet.

Elle nécessite aussi l'ajout dans le classpath de la bibliothèque jsf-api.jar de la ou des bibliothèques requises par l'implémentation JSF utilisées.

Ces bibliothèques doivent aussi être disponibles pour le conteneur web qui va exécuter l'application. Le plus simple est de mettre ces fichiers dans le répertoire WEB-INF/lib

69.17. Un exemple d'application simple

Cette section va développer une petite application constituée de deux pages. La première va demander le nom de l'utilisateur et la seconde afficher un message de bienvenue.

Il faut créer un répertoire, par exemple nommé Test_JSF et créer à l'intérieur la structure de l'application qui correspond à la structure de toute application Web selon les spécifications J2EE, notamment le répertoire WEB-INF avec ses sous-répertoires lib et classes.

Il faut ensuite copier les fichiers nécessaires à une utilisation de JSF dans l'application web.

Il suffit de copier les fichiers *.jar du répertoire lib de l'implémentation de référence vers le répertoire WEB-INF/lib du projet.

Il faut créer un fichier à la racine du projet et le nommer index.htm

Exemple :

```
01. <html>
02.   <head>
03.     <meta http-equiv="Refresh" content= "0; URL=login.faces"/>
04.     <title>Demarrage de l'application</title>
05.   </head>
06.   <body>
07.     <p>D&eacute;marrage de l'application ...</p>
08.   </body>
09. </html>
```

Il faut créer un fichier à la racine du projet et le nommer login.jsp

Exemple :

```
01. <html>
02. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
03. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
04. <f:view>
05. <head>
06.   <title>Application de tests avec JSF</title>
07. </head>
08. <body>
09.   <h:form>
10.     <h3>Identification</h3>
11.     <table>
12.       <tr>
13.         <td>Nom : </td>
14.         <td><h:inputText value="#{login.nom}"/></td>
15.       </tr>
16.       <tr>
17.         <td>Mot de passe :</td>
18.         <td><h:inputSecret value="#{login.mdp}"/></td>
19.       </tr>
20.       <tr>
21.         <td colspan="2"><h:commandButton value="Login" action="login"/></td>
22.       </tr>
23.     </table>
24.   </h:form>
25. </body>
26. </f:view>
27. </html>
```

Il faut créer une nouvelle classe nommée com.jmd.test.jsf.LoginBean et la compiler dans le répertoire WEB-INF/classes.

Exemple :

```
01. package com.jmd.test.jsf;
02.
03. public class LoginBean {
04.
05.   private String nom;
06.   private String mdp;
07.
08.   public String getMdp() {
09.     return mdp;
10.   }
```

```
11.
12.   public String getNom() {
13.     return nom;
14.   }
15.
16.   public void setMdp(String string) {
17.     mdp = string;
18.   }
19.
20.   public void setNom(String string) {
21.     nom = string;
22.   }
23. }
```

Il faut créer un fichier à la racine du projet et le nommer accueil.jsp : cette page contiendra la page d'accueil de l'application.

Il faut créer un fichier dans le répertoire /WEB-INF et le nommer faces-config.xml

Exemple :

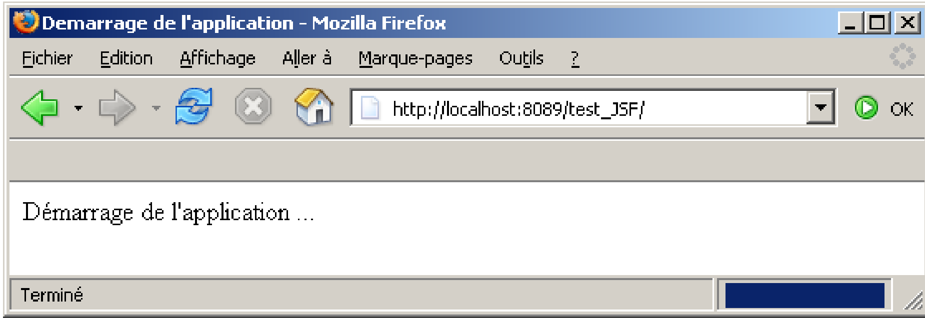
```
01. <?xml version="1.0"?>
02.
03. <!DOCTYPE faces-config PUBLIC
04. "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
05. "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
06.
07. <faces-config>
08.   <navigation-rule>
09.     <from-view-id>/login.jsp</from-view-id>
10.     <navigation-case>
11.       <from-outcome>login</from-outcome>
12.       <to-view-id>/accueil.jsp</to-view-id>
13.     </navigation-case>
14.   </navigation-rule>
15.
16.   <managed-bean>
17.     <managed-bean-name>login</managed-bean-name>
18.     <managed-bean-class>com.jmd.test.jsf.LoginBean</managed-bean-class>
19.     <managed-bean-scope>session</managed-bean-scope>
20.   </managed-bean>
21.
22. </faces-config>
```

Il faut créer un fichier dans le répertoire /WEB-INF et le nommer web.xml

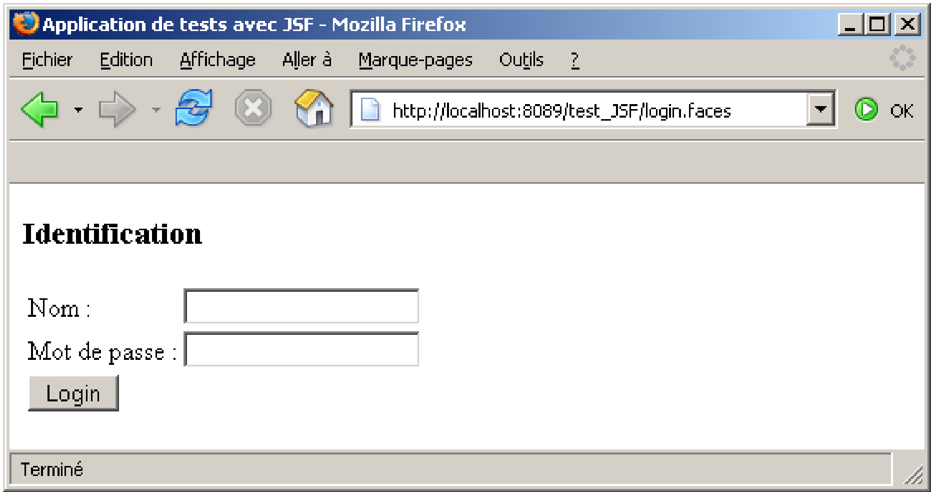
Exemple :

```
01. <?xml version="1.0"?>
02.
03. <!DOCTYPE web-app PUBLIC
04. "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
05. "http://java.sun.com/dtd/web-app_2_3.dtd">
06.
07. <web-app>
08.   <servlet>
09.     <servlet-name>Faces Servlet</servlet-name>
10.     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
11.     <load-on-startup>1</load-on-startup>
12.   </servlet>
13.
14.   <servlet-mapping>
15.     <servlet-name>Faces Servlet</servlet-name>
16.     <url-pattern>*.faces</url-pattern>
17.   </servlet-mapping>
18.
19.   <welcome-file-list>
20.     <welcome-file>index.htm</welcome-file>
21.   </welcome-file-list>
22.
23. </web-app>
```

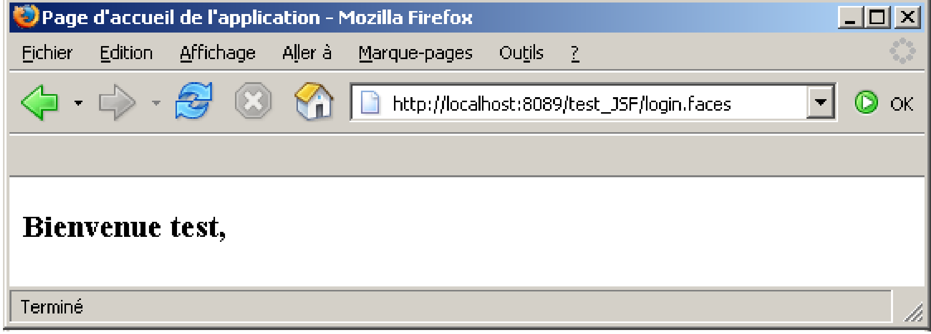
Il suffit alors de démarrer Tomcat, puis d'ouvrir un navigateur et taper l'URL http://localhost:8089/test_JSF/ (en remplaçant le port 8089 par celui défini dans Tomcat).



Une fois l'application démarrée, la page de login s'affiche



Il faut saisir un nom par exemple test et cliquer sur le bouton « Login ».



Cette exemple ne met en aucune façon en valeur la puissance de JSF mais permet simplement de mettre en place les éléments minimum pour une application utilisant JSF.

69.18. L'internationalisation

JSF propose des fonctionnalités qui facilitent l'internationalisation d'une application.

Il faut définir un fichier au format properties qui va contenir la définition des chaînes de caractères. Un tel fichier possède les caractéristiques suivantes :

- le fichier doit avoir l'extension .properties
- il doit être dans le classpath de l'application
- il est composé d'une paire clé=valeur par ligne. La clé permet d'identifier de façon unique la chaîne de caractères

Exemple : le fichier msg.properties

```
1. login_titre=Application de tests avec JSF
2. login_identification=Identification
3. login_nom=Nom
4. login_mdp=Mot de passe
5. login_Login=Valider
```

Ce fichier correspond à la langue par défaut. Il est possible de définir d'autres fichiers pour d'autres langues. Ces fichiers doivent avoir le même nom suivi d'un underscore et du code langue défini par le standard ISO 639 avec toujours l'extension .properties.

Exemple :

```
1. msg.properties
2. msg_en.properties
3. msg_de.properties
```

Il faut bien sûr remplacer les valeurs de chaque chaîne par leur traduction correspondante.

Exemple :

```
1. login_titre=Tests of JSF
2. login_identification=Login
3. login_nom=Name
4. login_mdp=Password
5. login_Login=Login
```

Les langues disponibles doivent être précisées dans le fichier de configuration.

Exemple :

```
01. <faces-config>
02. ...
03.   <application>
04.     <locale-config>
05.       <default-locale>fr</default-locale>
06.       <supported-locale>en</supported-locale>
07.     </locale-config>
08.   </application>
09. ...
10. </faces-config>
```

Pour utiliser l'internationalisation dans les vues, il faut utiliser le tag <f:loadBundle> pour charger le fichier .properties nécessaire. Deux attributs de ce tags sont requis :

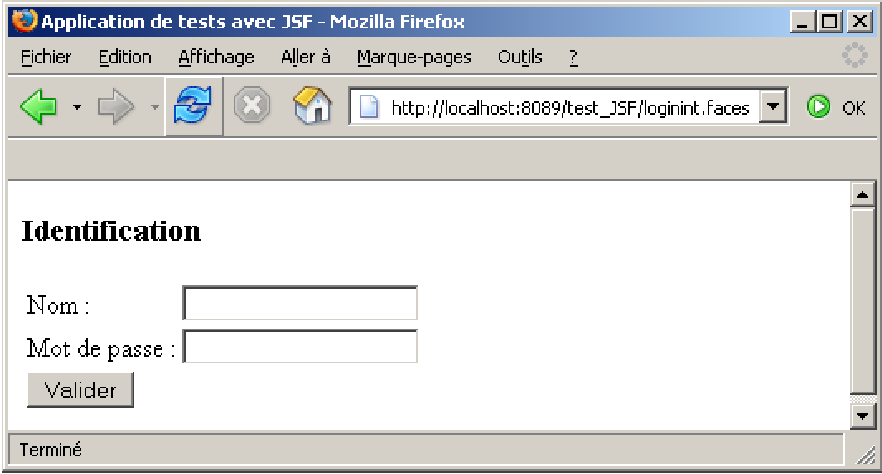
- basename : précise la localisation et le nom de base des fichiers .properties. La notation de la localisation est similaire à celle utilisée pour les packages
- var : précise le nom de la variable qui va contenir les chaînes de caractères

Il ne reste plus qu'à utiliser la variable définie en utilisant la notation avec un point pour la clé de la chaîne dont on souhaite utiliser la valeur.

Exemple :

```
01. <html>
02. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
03. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
04. <f:view>
05. <f:loadBundle basename="com.jmd.test.jsf.msg" var="msg"/>
06.
07. <head>
08.   <title><h:outputText value="#{msg.login_titre}"/></title>
09. </head>
10. <body>
11.   <h:form>
12.     <h3><h:outputText value="#{msg.login_identification}"/></h3>
13.     <table>
14.       <tr>
15.         <td><h:outputText value="#{msg.login_nom}"/> : </td>
16.         <td><h:inputText value="#{login.nom}"/></td>
17.       </tr>
18.       <tr>
19.         <td><h:outputText value="#{msg.login_mdp}"/> :</td>
20.         <td><h:inputSecret value="#{login.mdp}"/></td>
21.       </tr>
22.       <tr>
23.         <td colspan="2"><h:commandButton value="#{msg.login_Login}" action="login"/></td>
24.       </tr>
25.     </table>
26.   </h:form>
27. </body>
28. </f:view>
29. </html>
```

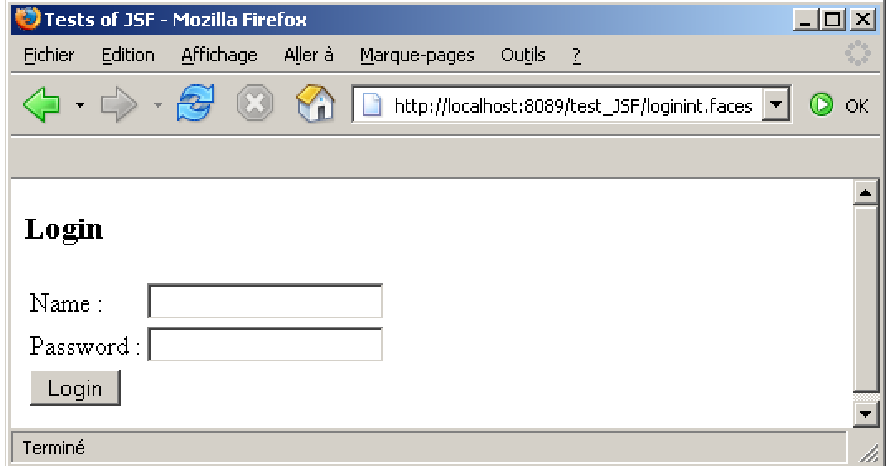
La langue à utiliser est déterminée automatiquement par JSF en fonction des informations contenues dans la propriété Accept-Language de l'en-tête de la requête et du fichier de configuration.



La langue peut aussi être forcée dans l'objet de type view en précisant le code langue dans l'attribut locale.

Exemple :

```
1. ...
2.     <f:view locale="en">
3. ...
```



Elle peut aussi être déterminée dans le code des traitements. L'exemple suivant va permettre à l'utilisateur de sélectionner la langue utilisée entre français et anglais grâce à deux petites icônes cliquables.

Exemple :

```
01. <html>
02. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
03. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
04. <f:view>
05. <f:loadBundle basename="com.jmd.test.jsf.Messages" var="msg"/>
06. <head>
07. <title>Application de tests avec JSF</title>
08. </head>
09. <body>
10.   <h:form>
11.
12.   <table>
13.     <tr>
14.       <td>
15.         <h:commandLink action="#{langueApp.activerFR}" immediate="true">
16.           <h:graphicImage value="images/francais.jpg" style="border: 0px"/>
17.         </h:commandLink>
18.       </td>
19.       <td>
20.         <h:commandLink action="#{langueApp.activerEN}" immediate="true">
21.           <h:graphicImage value="images/anglais.jpg" style="border: 0px"/>
22.         </h:commandLink>
23.       </td>
24.       <td width="100%"> </td>
25.     </tr>
26.   </table>
27.
28.   <h3><h:outputText value="#{msg.login_titre}" /></h3>
29.   <p> </p>
30.   <h:panelGrid columns="2">
31.     <h:outputText value="#{msg.login_nom}" />
32.     <h:panelGroup>
```

```
33.         <h:inputText value="#{login.nom}" id="nom" required="true"
34.           binding="#{login.inputTextNom}"/>
35.         <h:message for="nom"/>
36.       </h:panelGroup>
37.       <h:outputText value="#{msg.login_mdp}" />
38.       <h:inputSecret value="#{login.mdp}"/>
39.       <h:commandButton value="#{msg.login_valider}" action="login"/>
40.     </h:panelGrid>
41.
42.     </h:form>
43.     </body>
44.   </f:view>
45.   </html>
```

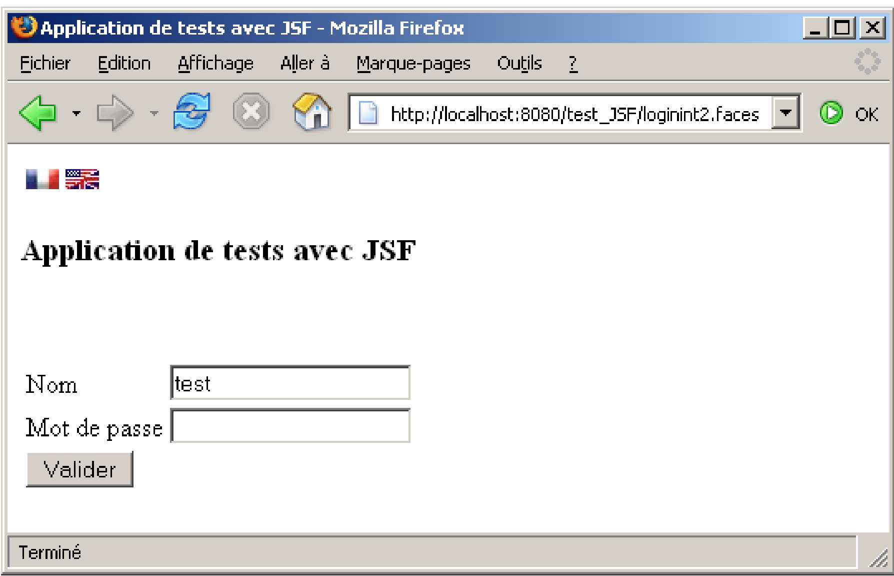
Ce code n'a rien de particulier si ce n'est l'utilisation de l'attribut immediate sur les liens sur le choix de la langue pour empêcher la validation des données lors d'un changement de la langue d'affichage.

Ce sont les deux méthodes du bean qui se chargent de modifier la Locale par défaut du contexte de l'application

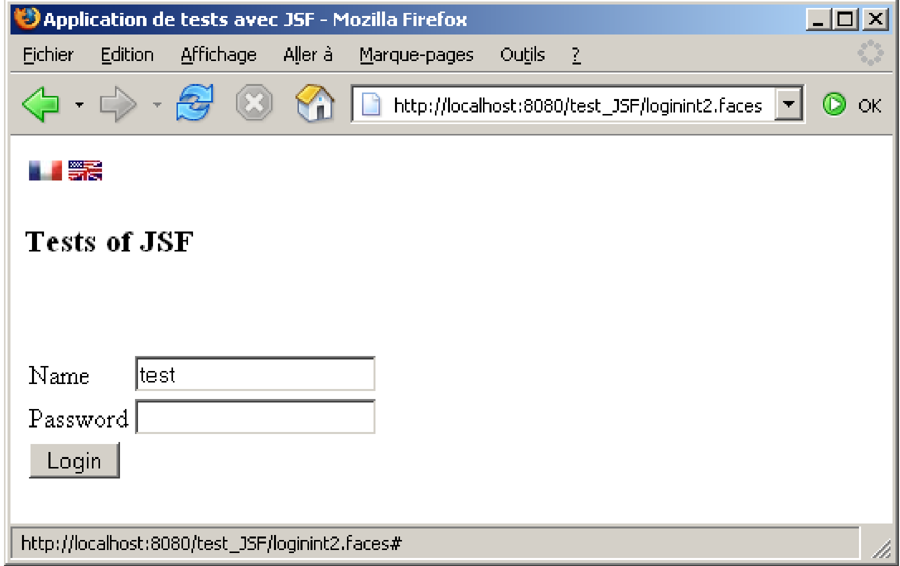
Exemple :

```
01. package com.jmd.test.jsf;
02.
03. import java.util.Locale;
04.
05. import javax.faces.context.FacesContext;
06.
07. public class LangueApp {
08.
09.   public String activerFR() {
10.     FacesContext context = FacesContext.getCurrentInstance();
11.     context.getViewRoot().setLocale(Locale.FRENCH);
12.     return null;
13.   }
14.
15.   public String activerEN() {
16.     FacesContext context = FacesContext.getCurrentInstance();
17.     context.getViewRoot().setLocale(Locale.ENGLISH);
18.     return null;
19.   }
20. }
```

Lors de l'exécution, la page s'affiche en français par défaut.



Lors d'un clic sur la petite icône indiquant la langue anglaise, la page est réaffichée en anglais.



69.19. Les points faibles de JSF

Malgré ses nombreux points forts, JSF possède aussi quelques points faibles :

- Maturité de la technologie

  JSF est une technologie récente qui nécessite l'écriture de beaucoup de code. Bien que prévu pour être utilisé avec des utilitaires facilitant la rédaction de la majeure partie de ce code, à l'heure actuelle, seuls quelques outils supportent JSF.

- Manque de composants évolués en standard

  L'implémentation standard ne propose que des composants simples dont la plupart ont une correspondance directe en HTML. Hormis le composant dataTable aucun composant évolué n'est proposé en standard dans la version 1.0. Il est donc nécessaire de développer ses propres composants ou d'acquérir les composants nécessaires auprès de tiers.

- Consommation en ressources d'une application JSF

  L'exécution d'une application JSF est assez gourmande en ressource notamment mémoire à cause du mode de fonctionnement du cycle de traitement d'une page. Ce cycle de vie inclut la création en mémoire d'une arborescence des composants utilisés lors des différentes étapes des traitements.

- Le rendu des composants uniquement en HTML en standard

  Dans l'implémentation de référence le rendu des composants est uniquement possible en HTML alors que JSF intègre un système de rendu (Renderer) découplé des traitements des composants. Pour un rendu différent de HTML, il est nécessaire de développer ses propres Renderers ou d'acquérir un système de rendu auprès de tiers.

La suite de ce chapitre sera développée dans une version future de ce document

Copyright (C) 1999-2014 Jean-Michel DOUDOUX.